

Edge Detection in Color Images

Edge information is essential in many image analysis and computer vision applications and thus the ability to locate and characterize edges robustly and accurately is an important task. Basic techniques for edge detection in *grayscale* images are discussed in Chapter 6. *Color* images contain richer information than grayscale images and it appears natural to assume that edge detection methods based on color should outperform their monochromatic counterparts. For example, locating an edge between two image regions of different hue but similar brightness is difficult with an edge detector that only looks for changes in image intensity. In this chapter, we first look at the use of “ordinary” (i.e., monochromatic) edge detectors for color images and then discuss dedicated detectors that are specifically designed for color images.

Although the problem of color edge detection has been pursued for a long time (see [140,266] for a good overview), most image processing texts do not treat this subject in much detail. One reason could be that, in practice, edge detection in color images is often accomplished by using “monochromatic” techniques on the intensity channel or the individual color components. We discuss these simple methods—which nevertheless give satisfactory results in many situations—in Sec. 16.1.

Unfortunately, monochromatic techniques do not extend naturally to color images and other “multi-channel” data, since edge information in the different color channels may be ambiguous or even contradictory. For example, multiple edges running in different directions may coincide at a given image location, edge gradients may cancel out, or edges in different channels may be slightly displaced. In Sec. 16.2, we describe how local gradients can be calculated for edge detection by treating the color image as a 2D *vector field*. In Sec. 16.3, we show how the popular Canny edge detector, originally designed for monochromatic images, can be adapted for color images, and Sec. 16.4 goes on to look at other color edge operators. Implementations of the discussed algorithms are described in Sec. 16.5, with complete source code available on the book’s website.

16.1 Monochromatic Techniques

Linear filters are the basis of most edge enhancement and edge detection operators for scalar-valued grayscale images, particularly the gradient filters described in Chapter 15, Sec. 6.3. Again, it is quite common to apply these scalar filters separately to the individual color channels of RGB images. A popular example is the Sobel operator with the filter kernels

$$H_x^S = \frac{1}{8} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^S = \frac{1}{8} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (16.1)$$

for the x - and y -direction, respectively. Applied to a grayscale image I , with $I_x = I * H_x^S$ and $I_y = I * H_y^S$, these filters give a reasonably good estimate of the local gradient vector,

$$\nabla I(\mathbf{u}) = \begin{pmatrix} I_x(\mathbf{u}) \\ I_y(\mathbf{u}) \end{pmatrix}, \quad (16.2)$$

at position $\mathbf{u} = (u, v)$. The local edge *strength* of the grayscale image is then taken as

$$E_{\text{gray}}(\mathbf{u}) = \|\nabla I(\mathbf{u})\| = \sqrt{I_x^2(\mathbf{u}) + I_y^2(\mathbf{u})}, \quad (16.3)$$

and the corresponding edge *orientation* is calculated as

$$\Phi(\mathbf{u}) = \angle \nabla I(\mathbf{u}) = \tan^{-1} \left(\frac{I_y(\mathbf{u})}{I_x(\mathbf{u})} \right). \quad (16.4)$$

The angle $\Phi(\mathbf{u})$ gives the direction of maximum intensity change on the 2D image surface at position (\mathbf{u}) , which is the normal to the edge tangent.

Analogously, to apply this technique to a color image $\mathbf{I} = (I_R, I_G, I_B)$, each color plane is first filtered individually with the two gradient kernels given in Eqn. (16.1), resulting in

$$\begin{aligned} \nabla I_R &= \begin{pmatrix} I_{R,x} \\ I_{R,y} \end{pmatrix} = \begin{pmatrix} I_R * H_x^S \\ I_R * H_y^S \end{pmatrix}, \\ \nabla I_G &= \begin{pmatrix} I_{G,x} \\ I_{G,y} \end{pmatrix} = \begin{pmatrix} I_G * H_x^S \\ I_G * H_y^S \end{pmatrix}, \\ \nabla I_B &= \begin{pmatrix} I_{B,x} \\ I_{B,y} \end{pmatrix} = \begin{pmatrix} I_B * H_x^S \\ I_B * H_y^S \end{pmatrix}. \end{aligned} \quad (16.5)$$

The local edge strength is calculated separately for each color channel which yields a vector

$$\mathbf{E}(\mathbf{u}) = \begin{pmatrix} E_R(\mathbf{u}) \\ E_G(\mathbf{u}) \\ E_B(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \|\nabla I_R(\mathbf{u})\| \\ \|\nabla I_G(\mathbf{u})\| \\ \|\nabla I_B(\mathbf{u})\| \end{pmatrix} \quad (16.6)$$

$$= \begin{pmatrix} [I_{R,x}^2(\mathbf{u}) + I_{R,y}^2(\mathbf{u})]^{1/2} \\ [I_{G,x}^2(\mathbf{u}) + I_{G,y}^2(\mathbf{u})]^{1/2} \\ [I_{B,x}^2(\mathbf{u}) + I_{B,y}^2(\mathbf{u})]^{1/2} \end{pmatrix} \quad (16.7)$$

for each image position \mathbf{u} . These vectors could be combined into a new color image $\mathbf{E} = (E_R, E_G, E_B)$, although such a “color edge image” has no particularly useful interpretation.¹ Finally, a scalar quantity of *combined edge strength* (C) over all color planes can be obtained, for example, by calculating the Euclidean (L_2) norm of \mathbf{E} as

$$\begin{aligned} C_2(\mathbf{u}) &= \|\mathbf{E}(\mathbf{u})\|_2 = [E_R^2(\mathbf{u}) + E_G^2(\mathbf{u}) + E_B^2(\mathbf{u})]^{1/2} \\ &= [I_{R,x}^2 + I_{R,y}^2 + I_{G,x}^2 + I_{G,y}^2 + I_{B,x}^2 + I_{B,y}^2]^{1/2} \end{aligned} \quad (16.8)$$

(coordinates (\mathbf{u}) are omitted in the second line) or, using the L_1 norm,

$$C_1(\mathbf{u}) = \|\mathbf{E}(\mathbf{u})\|_1 = |E_R(\mathbf{u})| + |E_G(\mathbf{u})| + |E_B(\mathbf{u})|. \quad (16.9)$$

Another alternative for calculating a combined edge strength is to take the *maximum* magnitude of the RGB gradients (i.e., the L_∞ norm),

$$C_\infty(\mathbf{u}) = \|\mathbf{E}(\mathbf{u})\|_\infty = \max(|E_R(\mathbf{u})|, |E_G(\mathbf{u})|, |E_B(\mathbf{u})|). \quad (16.10)$$

An example using the test image from Chapter 15 is given in Fig. 16.1. It shows the edge magnitude of the corresponding grayscale image and the combined color edge magnitude calculated with the different norms defined in Eqns. (16.8)–(16.10).²

As far as edge *orientation* is concerned, there is no simple extension of the grayscale case. While edge orientation can easily be calculated for each individual color component (using Eqn. (16.4)), the gradients, three color channels are generally different (or even contradictory) and there is no obvious way of combining them.

A simple ad hoc approach is to choose, at each image position \mathbf{u} , the gradient direction from the color channel of maximum edge strength, that is,

$$\varPhi_{\text{col}}(\mathbf{u}) = \tan^{-1}\left(\frac{I_{m,y}(\mathbf{u})}{I_{m,x}(\mathbf{u})}\right), \quad (16.11)$$

with $m = \underset{k=R,G,B}{\operatorname{argmax}} E_k(\mathbf{u})$.

This simple (monochromatic) method for calculating edge strength and orientation in color images is summarized in Alg. 16.1 (see Sec. 16.5 for the corresponding Java implementation). Two sample results are shown in Fig. 16.2. For comparison, these figures also show the edge maps obtained by first converting the color image to a grayscale

¹ Such images are nevertheless produced by the “Find Edges” command in ImageJ and the filter of the same name in Photoshop (showing inverted components).

² In this case, the grayscale image in (c) was calculated with the *direct* conversion method (see Chapter 14, Eqn. (14.39)) from nonlinear sRGB components. With *linear* grayscale conversion (Ch. 14, Eqn. (14.37)), the desaturated bar at the center would exhibit no grayscale edges along its borders, since the luminance is the same inside and outside.

16 EDGE DETECTION IN COLOR IMAGES

Fig. 16.1

Color edge enhancement with monochromatic methods. Original color image (a) and corresponding grayscale image (b); edge magnitude from the grayscale image (c). Color edge magnitude calculated with different norms: L_1 (d), L_2 (e), and L_∞ (f). The images in (c–f) are inverted for better viewing.

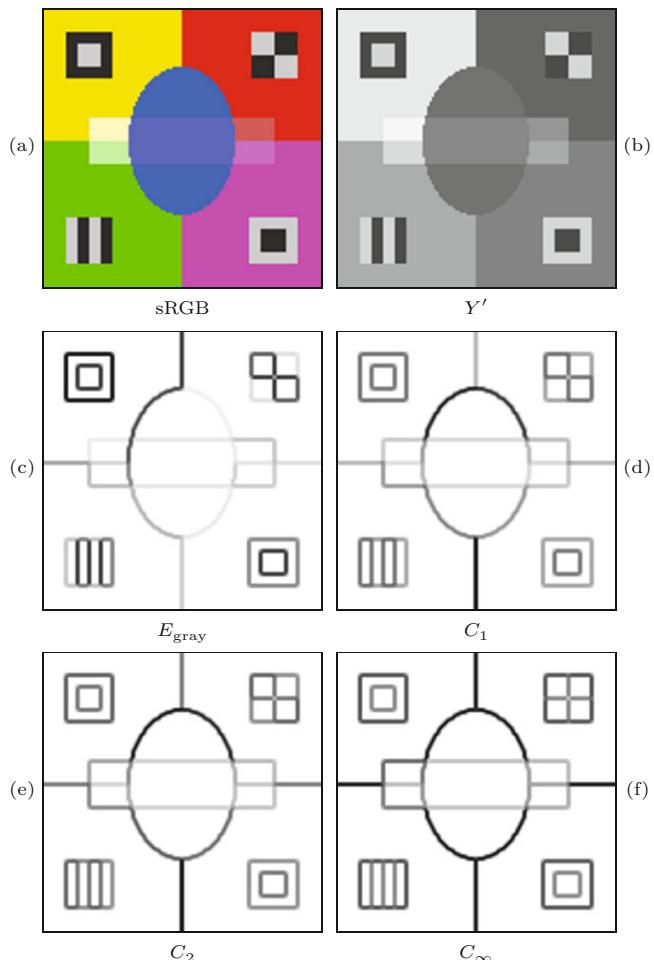


image and then applying the Sobel operator³ (Fig. 16.2(b)). The edge magnitude in all examples is normalized; it is shown inverted and contrast-enhanced to increase the visibility of low-contrast edges. As expected and apparent from the examples, even simple monochromatic techniques applied to color images perform better than edge detection on the corresponding grayscale images. In particular, edges between color regions of similar brightness are not detectable in this way, so using color information for edge detection is generally more powerful than relying on intensity alone. Among the simple color techniques, the maximum channel edge strength C_∞ (Eqn. (16.10)) seems to give the most consistent results with the fewest edges getting lost.

However, none of the monochromatic detection techniques can be expected to work reliably under these circumstances. While the threshold for binarizing the edge magnitude could be tuned manually to give more pleasing results on specific images, it is difficult in practice to achieve consistently good results over a wide range of images. Methods for determining the optimal edge threshold dynam-

³ See Chapter 6, Sec. 6.3.1.

1: MonochromaticColorEdge(\mathbf{I})

Input: $\mathbf{I} = (I_R, I_G, I_B)$, an RGB color image of size $M \times N$. Returns a pair of maps (E_2, Φ) for edge magnitude and orientation.

```

2:    $H_x^S \leftarrow \frac{1}{8} \cdot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 
3:    $H_y^S \leftarrow \frac{1}{8} \cdot \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$   $\triangleright x/y$  gradient kernels
4:    $(M, N) \leftarrow \text{Size}(\mathbf{I})$ 
5:   Create maps  $E, \Phi : M \times N \rightarrow \mathbb{R}$   $\triangleright$  edge magnitude/orientation
6:    $I_{R,x} \leftarrow I_R * H_x^S, \quad I_{R,y} \leftarrow I_R * H_y^S$   $\triangleright$  apply gradient filters
7:    $I_{G,x} \leftarrow I_G * H_x^S, \quad I_{G,y} \leftarrow I_G * H_y^S$ 
8:    $I_{B,x} \leftarrow I_B * H_x^S, \quad I_{B,y} \leftarrow I_B * H_y^S$ 
9:   for all image coordinates  $\mathbf{u} \in M \times N$  do
10:     $(r_x, g_x, b_x) \leftarrow (I_{R,x}(\mathbf{u}), I_{G,x}(\mathbf{u}), I_{B,x}(\mathbf{u}))$ 
11:     $(r_y, g_y, b_y) \leftarrow (I_{R,y}(\mathbf{u}), I_{G,y}(\mathbf{u}), I_{B,y}(\mathbf{u}))$ 
12:     $e_R^2 \leftarrow r_x^2 + r_y^2$ 
13:     $e_G^2 \leftarrow g_x^2 + g_y^2$ 
14:     $e_B^2 \leftarrow b_x^2 + b_y^2$ 
15:     $e_{\max}^2 \leftarrow e_R^2$   $\triangleright$  find maximum gradient channel
16:     $c_x \leftarrow r_x, \quad c_y \leftarrow r_y$ 
17:    if  $e_G^2 > e_{\max}^2$  then
18:       $e_{\max}^2 \leftarrow e_G^2, \quad c_x \leftarrow g_x, \quad c_y \leftarrow g_y$ 
19:    if  $e_B^2 > e_{\max}^2$  then
20:       $e_{\max}^2 \leftarrow e_B^2, \quad c_x \leftarrow b_x, \quad c_y \leftarrow b_y$ 
21:     $E(\mathbf{u}) \leftarrow \sqrt{e_R^2 + e_G^2 + e_B^2}$   $\triangleright$  edge magnitude ( $L_2$  norm)
22:     $\Phi(\mathbf{u}) \leftarrow \text{ArcTan}(c_x, c_y)$   $\triangleright$  edge orientation
23:   return  $(E, \Phi)$ .

```

16.2 EDGES IN VECTOR-VALUED IMAGES
Alg. 16.1

Monochromatic color edge operator. A pair of Sobel-type filter kernels (H_x^S, H_y^S) is used to estimate the local x/y gradients of each component of the RGB input image \mathbf{I} . Color edge magnitude is calculated as the L_2 norm of the color gradient vector (see Eqn. (16.8)). The procedure returns a pair of maps, holding the edge magnitude E_2 and the edge orientation Φ , respectively.

ically, that is, depending on the image content, have been proposed, typically based on the statistical variability of the color gradients. Additional details can be found in [84, 171, 192].

16.2 Edges in Vector-Valued Images

In the “monochromatic” scheme described in Sec. 16.1, the edge magnitude in each color channel is calculated separately and thus no use is made of the potential coupling between color channels. Only in a subsequent step are the individual edge responses in the color channels combined, albeit in an ad hoc fashion. In other words, the color data are not treated as vectors, but merely as separate and unrelated scalar values.

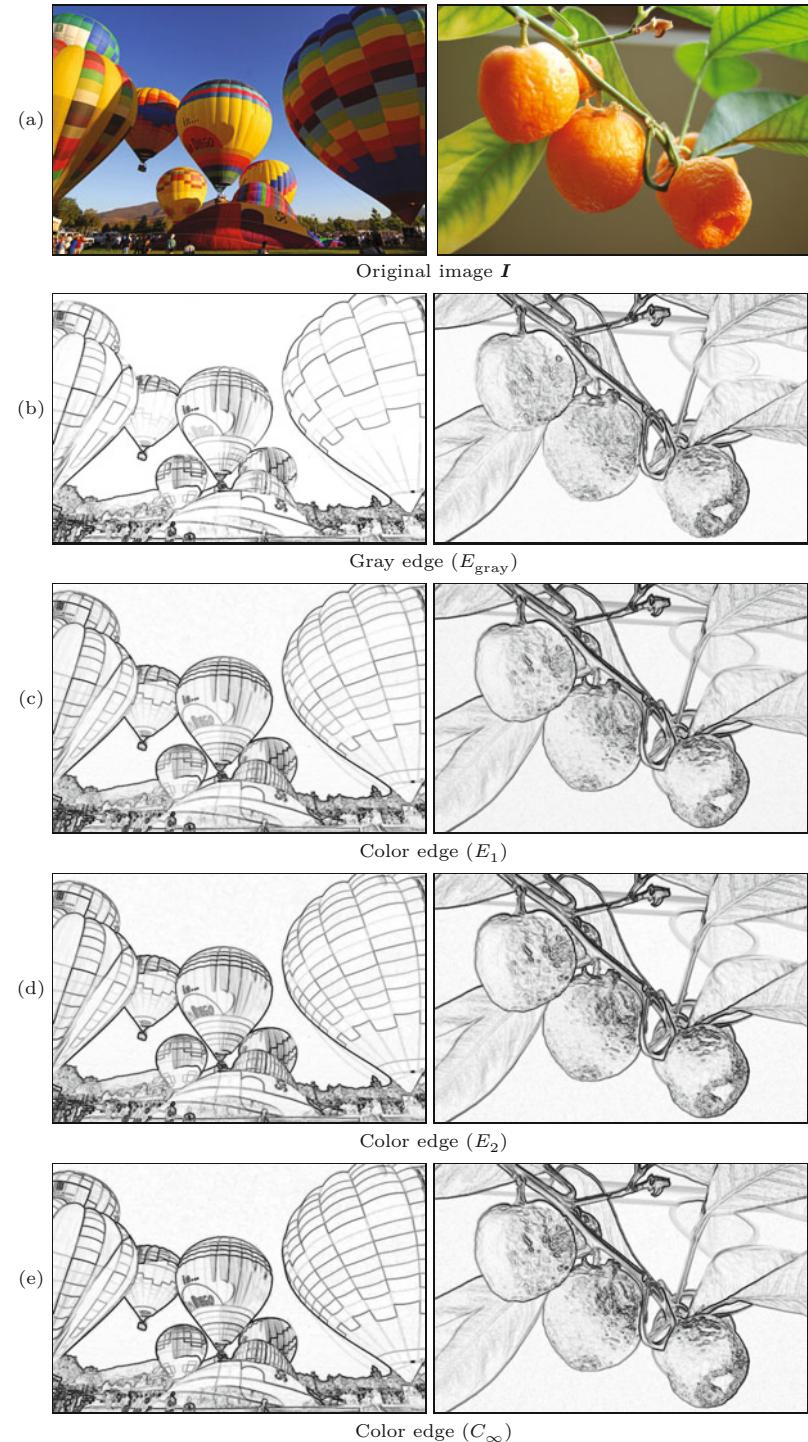
To obtain better insight into this problem it is helpful to treat the color image as a *vector field*, a standard construct in vector calculus [32, 223].⁴ A three-channel RGB color image $\mathbf{I}(\mathbf{u}) = (I_R(\mathbf{u}), I_G(\mathbf{u}), I_B(\mathbf{u}))$ can be modeled as a discrete 2D vector field, that is, a function whose coordinates $\mathbf{u} = (u, v)$ are 2D and whose values are 3D vectors.

⁴ See Sec. C.2 in the Appendix for some general properties of vector fields.

16 EDGE DETECTION IN COLOR IMAGES

Fig. 16.2

Example of color edge enhancement with monochromatic techniques (balloons image). Original color image and corresponding grayscale image (a), edge magnitude obtained from the grayscale image (b), color edge magnitude calculated with the L_2 norm (c), and the L_∞ norm (d). Differences between the grayscale edge detector (b) and the color-based detector (c–e) are particularly visible inside the right balloon and at the lower borders of the tangerines.



Similarly, a grayscale image can be described as a discrete *scalar field*, since its pixel values are only 1D.

16.2.1 Multi-Dimensional Gradients

As noted in the previous section, the gradient of a scalar image I at a specific position \mathbf{u} is defined as

$$\nabla I(\mathbf{u}) = \begin{pmatrix} \frac{\partial I}{\partial x}(\mathbf{u}) \\ \frac{\partial I}{\partial y}(\mathbf{u}) \end{pmatrix}, \quad (16.12)$$

that is, the vector of the partial derivatives of the function I in the x - and y -direction, respectively.⁵ Obviously, the gradient of a scalar image is a 2D vector field.

In the case of a color image $\mathbf{I} = (I_R, I_G, I_B)$, we can treat the three color channels as separate scalar images and obtain their gradients analogously as

$$\nabla I_R(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_R}{\partial x}(\mathbf{u}) \\ \frac{\partial I_R}{\partial y}(\mathbf{u}) \end{pmatrix}, \quad \nabla I_G(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_G}{\partial x}(\mathbf{u}) \\ \frac{\partial I_G}{\partial y}(\mathbf{u}) \end{pmatrix}, \quad \nabla I_B(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_B}{\partial x}(\mathbf{u}) \\ \frac{\partial I_B}{\partial y}(\mathbf{u}) \end{pmatrix}, \quad (16.13)$$

which is equivalent to what we did in Eqn. (16.5). Before we can take the next steps, we need to introduce a standard tool for the analysis of vector fields.

16.2.2 The Jacobian Matrix

The *Jacobian* matrix⁶ $\mathbf{J}_I(\mathbf{u})$ combines all first partial derivatives of a vector field \mathbf{I} at a given position \mathbf{u} , its row vectors being the gradients of the scalar component functions. In particular, for an RGB color image \mathbf{I} , the Jacobian matrix is defined as

$$\mathbf{J}_I(\mathbf{u}) = \begin{pmatrix} (\nabla I_R)^\top(\mathbf{u}) \\ (\nabla I_G)^\top(\mathbf{u}) \\ (\nabla I_B)^\top(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \frac{\partial I_R}{\partial x}(\mathbf{u}) & \frac{\partial I_R}{\partial y}(\mathbf{u}) \\ \frac{\partial I_G}{\partial x}(\mathbf{u}) & \frac{\partial I_G}{\partial y}(\mathbf{u}) \\ \frac{\partial I_B}{\partial x}(\mathbf{u}) & \frac{\partial I_B}{\partial y}(\mathbf{u}) \end{pmatrix} = (\mathbf{I}_x(\mathbf{u})^\top, \mathbf{I}_y(\mathbf{u})^\top), \quad (16.14)$$

with $\nabla I_R, \nabla I_G, \nabla I_B$ as defined in Eqn. (16.13). We see that the 2D gradient vectors $(\nabla I_R)^\top, (\nabla I_G)^\top, (\nabla I_B)^\top$ constitute the rows of the resulting 3×2 matrix \mathbf{J}_I . The two 3D column vectors of this matrix,

$$\mathbf{I}_x(\mathbf{u}) = \frac{\partial \mathbf{I}}{\partial x}(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_R}{\partial x}(\mathbf{u}) \\ \frac{\partial I_G}{\partial x}(\mathbf{u}) \\ \frac{\partial I_B}{\partial x}(\mathbf{u}) \end{pmatrix}, \quad \mathbf{I}_y(\mathbf{u}) = \frac{\partial \mathbf{I}}{\partial y}(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_R}{\partial y}(\mathbf{u}) \\ \frac{\partial I_G}{\partial y}(\mathbf{u}) \\ \frac{\partial I_B}{\partial y}(\mathbf{u}) \end{pmatrix}, \quad (16.15)$$

are the partial derivatives of the color components along the x - and y -axes, respectively. At a given position \mathbf{u} , the total amount of change over all three color channels in the horizontal direction can be quantified by the norm of the corresponding column vector $\|\mathbf{I}_x(\mathbf{u})\|$. Analogously, $\|\mathbf{I}_y(\mathbf{u})\|$ gives the total amount of change over all three color channels along the vertical axis.

⁵ Of course, images are discrete functions and the partial derivatives are estimated from finite differences (see Sec. C.3.1 in the Appendix).

⁶ See also Sec. C.2.1 in the Appendix.

16.2.3 Squared Local Contrast

Now that we can quantify the change along the horizontal and vertical axes at any position \mathbf{u} , the next task is to find out the direction of the *maximum* change to find the angle of the edge normal, which we then use to derive the local edge strength. How can we calculate the gradient in some direction θ other than horizontal and vertical? For this purpose, we use the product of the unit vector oriented at angle θ ,

$$\mathbf{e}_\theta = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}, \quad (16.16)$$

and the Jacobian matrix \mathbf{J}_I (Eqn. (16.14)) in the form

$$\begin{aligned} (\text{grad}_\theta \mathbf{I})(\mathbf{u}) &= \mathbf{J}_I(\mathbf{u}) \cdot \mathbf{e}_\theta = \left(\mathbf{I}_x(\mathbf{u}) \cdot \mathbf{I}_y(\mathbf{u}) \right) \cdot \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \\ &= \mathbf{I}_x(\mathbf{u}) \cdot \cos(\theta) + \mathbf{I}_y(\mathbf{u}) \cdot \sin(\theta). \end{aligned} \quad (16.17)$$

The resulting 3D vector $(\text{grad}_\theta \mathbf{I})(\mathbf{u})$ is called the *directional gradient*⁷ of the color image \mathbf{I} in the direction θ at position \mathbf{u} . By taking the squared norm of this vector,

$$\begin{aligned} S_\theta(\mathbf{I}, \mathbf{u}) &= \|(\text{grad}_\theta \mathbf{I})(\mathbf{u})\|_2^2 \\ &= \|\mathbf{I}_x(\mathbf{u}) \cdot \cos(\theta) + \mathbf{I}_y(\mathbf{u}) \cdot \sin(\theta)\|_2^2 \\ &= \mathbf{I}_x^2(\mathbf{u}) \cdot \cos^2(\theta) + 2 \cdot \mathbf{I}_x(\mathbf{u}) \cdot \mathbf{I}_y(\mathbf{u}) \cdot \cos(\theta) \cdot \sin(\theta) + \mathbf{I}_y^2(\mathbf{u}) \cdot \sin^2(\theta), \end{aligned} \quad (16.18)$$

we obtain what is called the *squared local contrast* of the vector-valued image \mathbf{I} at position \mathbf{u} in direction θ .⁸ For an RGB image $\mathbf{I} = (I_R, I_G, I_B)$, the squared local contrast in Eqn. (16.18) is, explicitly written,

$$S_\theta(\mathbf{I}, \mathbf{u}) = \left\| \begin{pmatrix} I_{R,x}(\mathbf{u}) \\ I_{G,x}(\mathbf{u}) \\ I_{B,x}(\mathbf{u}) \end{pmatrix} \cdot \cos(\theta) + \begin{pmatrix} I_{R,y}(\mathbf{u}) \\ I_{G,y}(\mathbf{u}) \\ I_{B,y}(\mathbf{u}) \end{pmatrix} \cdot \sin(\theta) \right\|_2^2 \quad (16.19)$$

$$\begin{aligned} &= [I_{R,x}^2(\mathbf{u}) + I_{G,x}^2(\mathbf{u}) + I_{B,x}^2(\mathbf{u})] \cdot \cos^2(\theta) \\ &+ [I_{R,y}^2(\mathbf{u}) + I_{G,y}^2(\mathbf{u}) + I_{B,y}^2(\mathbf{u})] \cdot \sin^2(\theta) \\ &+ 2 \cdot \cos(\theta) \cdot \sin(\theta) \cdot [I_{R,x}(\mathbf{u}) \cdot I_{R,y}(\mathbf{u}) + I_{G,x}(\mathbf{u}) \cdot I_{G,y}(\mathbf{u}) + I_{B,x}(\mathbf{u}) \cdot I_{B,y}(\mathbf{u})]. \end{aligned} \quad (16.20)$$

Note that, in the case that I is a *scalar* image, the squared local contrast reduces to

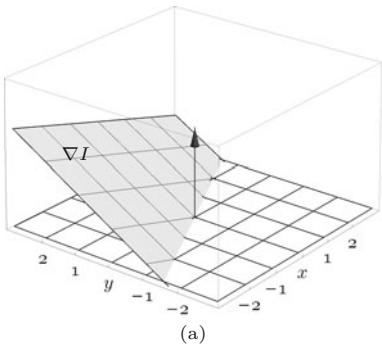
$$S_\theta(I, \mathbf{u}) = \|(\text{grad}_\theta I)(\mathbf{u})\|_2^2 = \left\| \begin{pmatrix} I_x(\mathbf{u}) \\ I_y(\mathbf{u}) \end{pmatrix} \cdot \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \right\|_2^2 \quad (16.21)$$

$$= [I_x(\mathbf{u}) \cdot \cos(\theta) + I_y(\mathbf{u}) \cdot \sin(\theta)]^2. \quad (16.22)$$

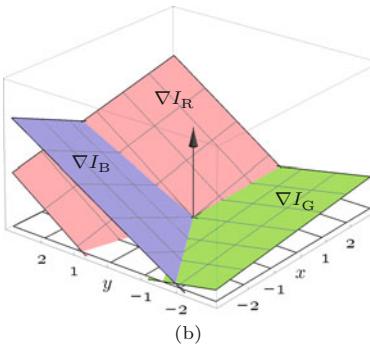
We will return to this result again later in Sec. 16.2.6. In the following, we use the root of the squared local contrast, that is, $\sqrt{S_\theta(I, \mathbf{u})}$, under the term *local contrast*.

⁷ See also Sec. C.2.2 in the Appendix (Eqn. (C.18)).

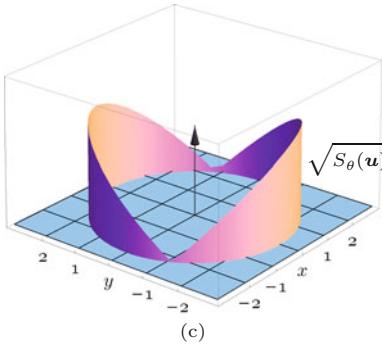
⁸ Note that $\mathbf{I}_x^2 = \mathbf{I}_x \cdot \mathbf{I}_x$, $\mathbf{I}_y^2 = \mathbf{I}_y \cdot \mathbf{I}_y$ and $\mathbf{I}_x \cdot \mathbf{I}_y$ in Eqn. (16.18) are dot products and thus the results are scalar values.

Grayscale image I 

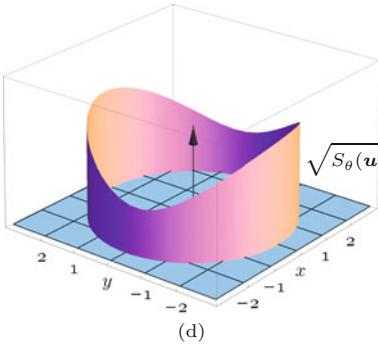
(a)

RGB color image $\mathbf{I} = (I_R, I_G, I_B)$ 

(b)



(c)



(d)

16.2 EDGES IN VECTOR-VALUED IMAGES

Fig. 16.3

Local image gradients and local contrast. In case of a scalar (grayscale) image I (a), the local gradient ∇I defines a single plane that is tangential to the image function I at position $\mathbf{u} = (u, v)$. In case of an RGB color image $\mathbf{I} = (I_R, I_G, I_B)$ (b), the local gradients ∇I_R , ∇I_G , ∇I_B for each color channel define three tangent planes. The vertical axes in graphs (c, d) show the corresponding local contrast values $\sqrt{S_\theta(\mathbf{I}, \mathbf{u})}$ (see Eqns. (16.18) and (16.19)) for all possible directions $\theta = 0, \dots, 2\pi$.

Figure 16.3 illustrates the meaning of the squared local contrast in relation to the local image gradients. At a given image position \mathbf{u} , the local gradient $\nabla I(\mathbf{u})$ in a grayscale image (**Fig. 16.3(a)**) defines a single plane that is tangential to the image function I at position \mathbf{u} . In case of a *color* image (**Fig. 16.3(b)**), each color channel defines an individual tangent plane. In **Fig. 16.3(c, d)** the *local contrast* values are shown as the height of cylindrical surfaces for all directions θ . For a *grayscale* image (**Fig. 16.3(c)**), the local contrast changes *linearly* with the orientation θ , while the relation is *quadratic* for a color image (**Fig. 16.3(d)**). To calculate the strength and orientation of edges we need to determine the direction of the *maximum* local contrast, which is described in the following.

16.2.4 Color Edge Magnitude

The directions that *maximize* $S_\theta(\mathbf{I}, \mathbf{u})$ in Eqn. (16.18) can be found analytically as the roots of the first partial derivative of S with respect to the angle θ , as originally suggested by Di Zenzo [63], and the resulting quantity is called *maximum local contrast*. As shown in [59], the maximum local contrast can also be found from the Jacobian matrix \mathbf{J}_I (Eqn. (16.14)) as the largest eigenvalue of the (symmetric) 2×2 matrix

$$\mathbf{M}(\mathbf{u}) = \mathbf{J}_I^\top(\mathbf{u}) \cdot \mathbf{J}_I(\mathbf{u}) = \begin{pmatrix} \mathbf{I}_x^\top(\mathbf{u}) \\ \mathbf{I}_y^\top(\mathbf{u}) \end{pmatrix} \cdot (\mathbf{I}_x(\mathbf{u}) \mathbf{I}_y^\top(\mathbf{u})) \quad (16.23)$$

$$= \begin{pmatrix} \mathbf{I}_x^2(\mathbf{u}) & \mathbf{I}_x(\mathbf{u}) \cdot \mathbf{I}_y(\mathbf{u}) \\ \mathbf{I}_y(\mathbf{u}) \cdot \mathbf{I}_x(\mathbf{u}) & \mathbf{I}_y^2(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} A(\mathbf{u}) & C(\mathbf{u}) \\ C(\mathbf{u}) & B(\mathbf{u}) \end{pmatrix}, \quad (16.24)$$

with the elements

$$\begin{aligned} A(\mathbf{u}) &= \mathbf{I}_x^2(\mathbf{u}) = \mathbf{I}_x(\mathbf{u}) \cdot \mathbf{I}_x(\mathbf{u}), \\ B(\mathbf{u}) &= \mathbf{I}_y^2(\mathbf{u}) = \mathbf{I}_y(\mathbf{u}) \cdot \mathbf{I}_y(\mathbf{u}), \\ C(\mathbf{u}) &= \mathbf{I}_x(\mathbf{u}) \cdot \mathbf{I}_y(\mathbf{u}) = \mathbf{I}_y(\mathbf{u}) \cdot \mathbf{I}_x(\mathbf{u}). \end{aligned} \quad (16.25)$$

The matrix $\mathbf{M}(\mathbf{u})$ could be considered as the color equivalent to the *local structure matrix* used for corner detection on grayscale images in Chapter 7, Sec. 7.2.1. The two eigenvalues λ_1, λ_2 of \mathbf{M} can be found in closed form as⁹

$$\begin{aligned} \lambda_1(\mathbf{u}) &= (A + B + \sqrt{(A - B)^2 + 4 \cdot C^2})/2, \\ \lambda_2(\mathbf{u}) &= (A + B - \sqrt{(A - B)^2 + 4 \cdot C^2})/2. \end{aligned} \quad (16.26)$$

Since \mathbf{M} is symmetric, the expression under the square root in Eqn. (16.26) is positive and thus all eigenvalues are real. In addition, A, B are both positive and therefore λ_1 is always the *larger* of the two eigenvalues. It is equivalent to the maximum squared local contrast (Eqn. (16.18)), that is,

$$\lambda_1(\mathbf{u}) \equiv \max_{0 \leq \theta < 2\pi} S_\theta(\mathbf{I}, \mathbf{u}), \quad (16.27)$$

and thus $\sqrt{\lambda_1}$ can be used directly to quantify the local edge strength. The eigenvector associated with $\lambda_1(\mathbf{u})$ is

$$\mathbf{q}_1(\mathbf{u}) = \left(\frac{A - B + \sqrt{(A - B)^2 + 4 \cdot C^2}}{2 \cdot C} \right), \quad (16.28)$$

or, equivalently, any multiple of \mathbf{q}_1 .¹⁰ Thus the rate of change along the vector \mathbf{q}_1 is the same as in the opposite direction $-\mathbf{q}_1$, and it follows that the local contrast $S_\theta(\mathbf{I}, \mathbf{u})$ at orientation θ is the same at orientation $\theta + k\pi$ (for any $k \in \mathbb{Z}$).¹¹ As usual, the *unit vector* corresponding to \mathbf{q}_1 is obtained by scaling \mathbf{q}_1 by its magnitude, that is,

$$\hat{\mathbf{q}}_1 = \frac{1}{\|\mathbf{q}_1\|} \cdot \mathbf{q}_1. \quad (16.29)$$

An alternative method, proposed in [60], is to calculate the unit eigenvector $\hat{\mathbf{q}}_1 = (\hat{x}_1, \hat{y}_1)^\top$ in the form

$$\hat{\mathbf{q}}_1 = \left(\sqrt{\frac{1+\alpha}{2}}, \operatorname{sgn}(C) \cdot \sqrt{\frac{1-\alpha}{2}} \right)^\top, \quad (16.30)$$

with $\alpha = (A - B)/\sqrt{(A - B)^2 + 4C^2}$, directly from the matrix elements A, B, C defined in Eqn. (16.25).

While \mathbf{q}_1 (the eigenvector associated with the greater eigenvalue of \mathbf{M}) points in the direction of maximum change, the second eigenvector \mathbf{q}_2 (associated with λ_2) is *orthogonal* to \mathbf{q}_1 , that is, has the same direction as the local edge tangent.

⁹ See Sec. B.4 in the Appendix for details.

¹⁰ The eigenvalues of a matrix are unique, but the corresponding eigenvectors are not.

¹¹ Thus the orientation of maximum change is inherently ambiguous [60].

16.2.5 Color Edge Orientation

The local orientation of the edge (i.e., the *normal* to the edge tangent) at a given position \mathbf{u} can be obtained directly from the associated eigenvector $\mathbf{q}_1(\mathbf{u}) = (q_x(\mathbf{u}), q_y(\mathbf{u}))^\top$ using the relation

$$\tan(\theta_1(\mathbf{u})) = \frac{q_x(\mathbf{u})}{q_y(\mathbf{u})} = \frac{2 \cdot C}{A - B + \sqrt{(A - B)^2 + 4 \cdot C^2}}, \quad (16.31)$$

which can be simplified¹² to

$$\tan(2 \cdot \theta_1(\mathbf{u})) = \frac{2 \cdot C}{A - B}. \quad (16.32)$$

Unless both $A = B$ and $C = 0$ (in which case the edge orientation is undetermined) the angle of maximum local contrast or color edge orientation can be calculated as

$$\theta_1(\mathbf{u}) = \frac{1}{2} \cdot \tan^{-1}\left(\frac{2 \cdot C}{A - B}\right) = \frac{1}{2} \cdot \text{ArcTan}(A - B, 2 \cdot C). \quad (16.33)$$

The above steps are summarized in Alg. 16.2, which is a color edge operator based on the first derivatives of the image function (see Sec. 16.5 for the corresponding Java implementation). It is similar to the algorithm proposed by Di Zenzo [63] but uses the eigenvalues of the local structure matrix for calculating edge magnitude and orientation, as suggested in [59] (see Eqn. (16.24)).

Results of the monochromatic edge operator in Alg. 16.1 and the Di Zenzo-Cumani multi-gradient operator in Alg. 16.2 are compared in Fig. 16.4. The synthetic test image in Fig. 16.4(a) has constant luminance (brightness) and thus no gray-value operator should be able to detect edges in this image. The local edge strength $E(\mathbf{u})$ produced by the two operators is very similar (Fig. 16.4(b)). The vectors in Fig. 16.4(c–f) show the orientation of the edge tangents that are normals to the direction of maximum color contrast, $\Phi(\mathbf{u})$. The length of each tangent vector is proportional to the local edge strength $E(\mathbf{u})$.

Figure 16.5 shows two examples of applying the Di Zenzo-Cumani-style color edge operator (Alg. 16.2) to real images. Note that the multi-gradient edge magnitude (calculated from the eigenvalue λ_1 in Eqn. (16.27)) in Fig. 16.5(b) is virtually identical to the monochromatic edge magnitude E_{mag} under the L_2 norm in Fig. 16.2(d). The larger difference to the result for the L_∞ norm in Fig. 16.2(e) is shown in Fig. 16.5(c).

Thus, considering only edge *magnitude*, the Di Zenzo-Cumani operator has no significant advantage over the simpler, monochromatic operator in Sec. 16.1. However, if edge *orientation* is important (as in the color version of the Canny operator described in Sec. 16.3), the Di Zenzo-Cumani technique is certainly more reliable and consistent.

16.2.6 Grayscale Gradients Revisited

As one might have guessed, the usual gradient-based calculation of the edge orientation (see Ch. 6, Sec. 6.2) is only a special case of the

¹² Using the relation $\tan(2\theta) = [2 \cdot \tan(\theta)] / [1 - \tan^2(\theta)]$.

16 EDGE DETECTION IN COLOR IMAGES

Alg. 16.2

Di Zenzo/Cumani-style multi-gradient color edge operator.

A pair of Sobel-type filters (H_x^S, H_y^S) is used for estimating the local x/y gradients in each component of the RGB input image \mathbf{I} . The procedure returns a pair of maps, holding the edge magnitude $E(\mathbf{u})$ and the edge orientation $\Phi(\mathbf{u})$, respectively.

1: MultiGradientColorEdge(\mathbf{I})

Input: $\mathbf{I} = (I_R, I_G, I_B)$, an RGB color image of size $M \times N$. Returns a pair of maps (E, Φ) for edge magnitude and orientation.

```

2:    $H_x^S := \frac{1}{8} \cdot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 
3:    $H_y^S := \frac{1}{8} \cdot \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$   $\triangleright x/y$  gradient kernels
4:    $(M, N) \leftarrow \text{Size}(\mathbf{I})$ 
5:   Create maps  $E, \Phi : M \times N \mapsto \mathbb{R}$   $\triangleright$  edge magnitude/orientation
6:    $I_{R,x} \leftarrow I_R * H_x^S, \quad I_{R,y} \leftarrow I_R * H_y^S$   $\triangleright$  apply gradient filters
7:    $I_{G,x} \leftarrow I_G * H_x^S, \quad I_{G,y} \leftarrow I_G * H_y^S$ 
8:    $I_{B,x} \leftarrow I_B * H_x^S, \quad I_{B,y} \leftarrow I_B * H_y^S$ 
9:   for all  $\mathbf{u} \in M \times N$  do
10:     $(\mathbf{r}_x, \mathbf{g}_x, \mathbf{b}_x) \leftarrow (I_{R,x}(\mathbf{u}), I_{G,x}(\mathbf{u}), I_{B,x}(\mathbf{u}))$ 
11:     $(\mathbf{r}_y, \mathbf{g}_y, \mathbf{b}_y) \leftarrow (I_{R,y}(\mathbf{u}), I_{G,y}(\mathbf{u}), I_{B,y}(\mathbf{u}))$ 
12:     $A \leftarrow \mathbf{r}_x^2 + \mathbf{g}_x^2 + \mathbf{b}_x^2$   $\triangleright A = \mathbf{I}_x \cdot \mathbf{I}_x$ 
13:     $B \leftarrow \mathbf{r}_y^2 + \mathbf{g}_y^2 + \mathbf{b}_y^2$   $\triangleright B = \mathbf{I}_y \cdot \mathbf{I}_y$ 
14:     $C \leftarrow \mathbf{r}_x \cdot \mathbf{r}_y + \mathbf{g}_x \cdot \mathbf{g}_y + \mathbf{b}_x \cdot \mathbf{b}_y$   $\triangleright C = \mathbf{I}_x \cdot \mathbf{I}_y$ 
15:     $\lambda_1 \leftarrow (A+B+\sqrt{(A-B)^2+4C^2})/2$   $\triangleright$  Eq. 16.26
16:     $E(\mathbf{u}) \leftarrow \sqrt{\lambda_1}$   $\triangleright$  Eq. 16.27
17:     $\Phi(\mathbf{u}) \leftarrow \frac{1}{2} \cdot \text{ArcTan}(A-B, 2 \cdot C)$   $\triangleright$  Eq. 16.33
18:   return  $(E, \Phi)$ .
```

multi-dimensional gradient calculation described already. Given a scalar image I , the intensity gradient vector $(\nabla I)(\mathbf{u}) = (I_x(\mathbf{u}), I_y(\mathbf{u}))^\top$ defines a single plane that is tangential to the image function at position \mathbf{u} , as illustrated in Fig. 16.3(a). With

$$A = I_x^2(\mathbf{u}), \quad B = I_y^2(\mathbf{u}), \quad C = I_x(\mathbf{u}) \cdot I_y(\mathbf{u}) \quad (16.34)$$

(analogous to Eqn. (16.25)) the squared local contrast at position \mathbf{u} in direction θ (as defined in Eqn. (16.18)) is

$$S_\theta(I, \mathbf{u}) = (I_x(\mathbf{u}) \cdot \cos(\theta) + I_y(\mathbf{u}) \cdot \sin(\theta))^2. \quad (16.35)$$

From Eqn. (16.26), the eigenvalues of the local structure matrix $\mathbf{M} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$ at position \mathbf{u} are (see Eqn. (16.26))

$$\lambda_{1,2}(\mathbf{u}) = (A+B \pm \sqrt{(A-B)^2+4C^2})/2, \quad (16.36)$$

but here, with I_x, I_y not being vectors but scalar values, we get $C^2 = (I_x \cdot I_y)^2 = I_x^2 \cdot I_y^2$, such that $(A-B)^2+4C^2 = (A+B)^2$, and therefore

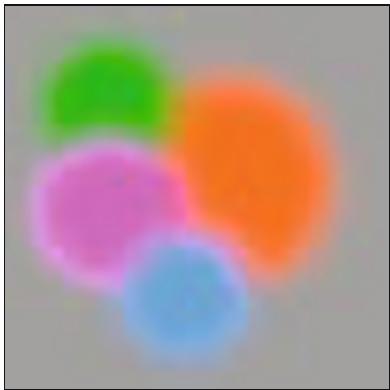
$$\lambda_{1,2}(\mathbf{u}) = (A+B \pm (A+B))/2. \quad (16.37)$$

We see that, for a scalar-valued image, the dominant eigenvalue,

$$\lambda_1(\mathbf{u}) = A+B = I_x^2(\mathbf{u}) + I_y^2(\mathbf{u}) = \|\nabla I(\mathbf{u})\|_2^2, \quad (16.38)$$

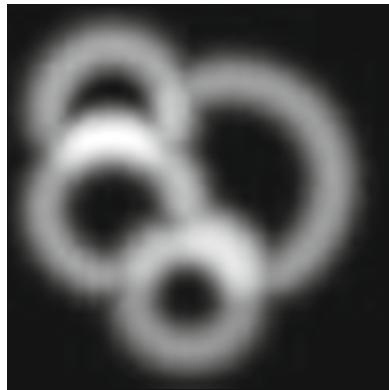
is simply the squared L₂ norm of the local gradient vector, while the smaller eigenvalue λ_2 is always zero. Thus, for a grayscale image, the

Original image



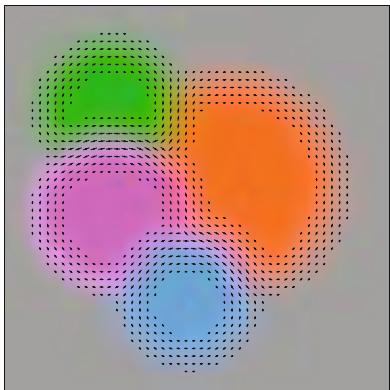
(a)

Color edge strength



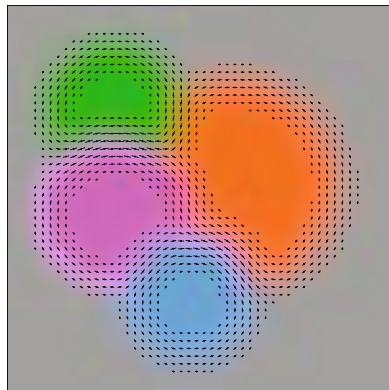
(b)

Monochromatic operator

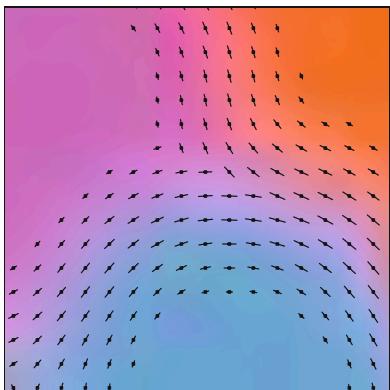


(c)

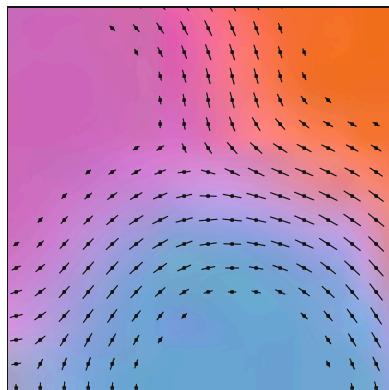
Di Zenzo-Cumani operator



(d)



(e)



(f)

16.2 EDGES IN VECTOR-VALUED IMAGES

Fig. 16.4

Results from the monochromatic (Alg. 16.1) and the Di Zenzo-Cumani color edge operators (Alg. 16.2). The original color image (a) has constant luminance, that is, the intensity gradient is zero and thus a simple grayscale operator would not detect any edges at all. The local edge strength $E(\mathbf{u})$ is almost identical for both color edge operators (b). Edge tangent orientation vectors (normal to $\Phi(\mathbf{u})$) for the monochromatic and multi-gradient operators (c, d); enlarged details in (e, f).

maximum edge strength $\sqrt{\lambda_1(\mathbf{u})} = \|\nabla I(\mathbf{u})\|_2$ is equivalent to the magnitude of the local intensity gradient.¹³ The fact that $\lambda_2 = 0$ indicates that the local contrast in the orthogonal direction (i.e., along the edge tangent) is zero (see Fig. 16.3(c)).

To calculate the local edge *orientation*, at position \mathbf{u} we use Eqn. (16.31) to get

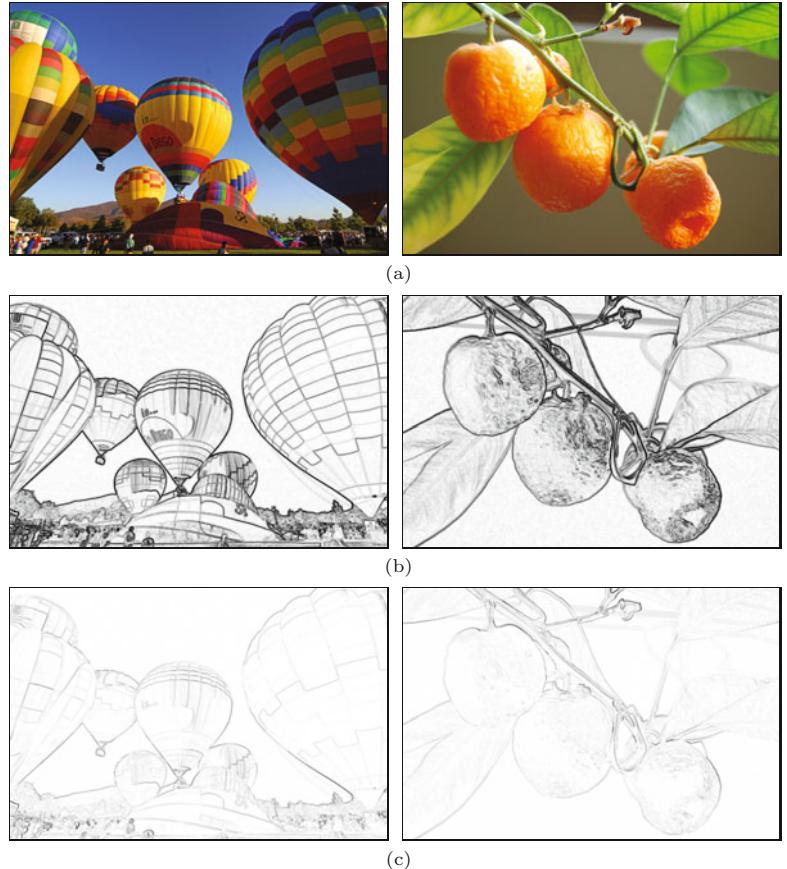
¹³ See Eqns. (6.5) and (6.13) in Chapter 6, Sec. 6.2.

16 EDGE DETECTION IN COLOR IMAGES

Fig. 16.5

Results of Di Zenzo-Cumani color edge operator (Alg. 16.2) on real images. Original image (a) and inverted color edge magnitude (b).

The images in (c) show the differences to the edge magnitude returned by the monochromatic operator (Alg. 16.1, using the L_∞ norm).



$$\tan(\theta_1(\mathbf{u})) = \frac{2C}{A - B + (A + B)} = \frac{2C}{2A} = \frac{I_x(\mathbf{u}) \cdot I_y(\mathbf{u})}{I_x^2(\mathbf{u})} = \frac{I_y(\mathbf{u})}{I_x(\mathbf{u})} \quad (16.39)$$

and the direction of maximum contrast¹⁴ is then found as

$$\theta_1(\mathbf{u}) = \tan^{-1}\left(\frac{I_y(\mathbf{u})}{I_x(\mathbf{u})}\right) = \text{ArcTan}(I_x(\mathbf{u}), I_y(\mathbf{u})). \quad (16.40)$$

Thus, for scalar-valued images, the general (multi-dimensional) technique based on the eigenvalues of the structure matrix leads to exactly the same result as the conventional grayscale edge detection approach described in Chapter 6, Sec. 6.3.

16.3 Canny Edge Detector for Color Images

Like most other edge operators, the Canny detector was originally designed for grayscale (i.e., scalar-valued) images. To use it on color images, a trivial approach is to apply the monochromatic operator separately to each of the color channels and subsequently merge the results into a single edge map. However, since edges within the different color channels rarely occur in the same places, the result will

¹⁴ See Eqn. (6.14) in Chapter 6.

usually contain multiple edge marks and undesirable clutter (see Fig. 16.8 for an example).

Fortunately, the original grayscale version of the Canny edge detector can be easily adapted to color imagery using the multi-gradient concept described in Sec. 16.2.1. The only changes required in Alg. 6.1 are the calculation of the local gradients and the edge magnitude E_{mag} . The modified procedure is shown in Alg. 16.3 (see Sec. 16.5 for the corresponding Java implementation).

```

1: ColorCannyEdgeDetector( $I, \sigma, t_{\text{hi}}, t_{\text{lo}}$ )
   Input:  $I = (I_R, I_G, I_B)$ , an RGB color image of size  $M \times N$ ;
           $\sigma$ , radius of Gaussian filter  $H^{G,\sigma}$ ;  $t_{\text{hi}}, t_{\text{lo}}$ , hysteresis thresholds
          ( $t_{\text{hi}} > t_{\text{lo}}$ ). Returns a binary edge image of size  $M \times N$ .
2:  $\bar{I}_R \leftarrow I_R * H^{G,\sigma}$      $\triangleright$  blur components with Gaussian of width  $\sigma$ 
3:  $\bar{I}_G \leftarrow I_G * H^{G,\sigma}$ 
4:  $\bar{I}_B \leftarrow I_B * H^{G,\sigma}$ 
5:  $H_x^\nabla \leftarrow [-0.5 \ 0 \ 0.5]$      $\triangleright x$  gradient filter
6:  $H_y^\nabla \leftarrow [-0.5 \ 0 \ 0.5]^\top$      $\triangleright y$  gradient filter
7:  $\bar{I}_{R,x} \leftarrow \bar{I}_R * H_x^\nabla, \quad \bar{I}_{R,y} \leftarrow \bar{I}_R * H_y^\nabla$ 
8:  $\bar{I}_{G,x} \leftarrow \bar{I}_G * H_x^\nabla, \quad \bar{I}_{G,y} \leftarrow \bar{I}_G * H_y^\nabla$ 
9:  $\bar{I}_{B,x} \leftarrow \bar{I}_B * H_x^\nabla, \quad \bar{I}_{B,y} \leftarrow \bar{I}_B * H_y^\nabla$ 
10:  $(M, N) \leftarrow \text{Size}(I)$ 
11: Create maps:
12:    $E_{\text{mag}}, E_{\text{nms}}, E_x, E_y : M \times N \rightarrow \mathbb{R}$ 
13:    $E_{\text{bin}} : M \times N \rightarrow \{0, 1\}$ 
14:   for all image coordinates  $\mathbf{u} \in M \times N$  do
15:      $(r_x, g_x, b_x) \leftarrow (I_{R,x}(\mathbf{u}), I_{G,x}(\mathbf{u}), I_{B,x}(\mathbf{u}))$ 
16:      $(r_y, g_y, b_y) \leftarrow (I_{R,y}(\mathbf{u}), I_{G,y}(\mathbf{u}), I_{B,y}(\mathbf{u}))$ 
17:      $A \leftarrow r_x^2 + g_x^2 + b_x^2,$ 
18:      $B \leftarrow r_y^2 + g_y^2 + b_y^2$ 
19:      $C \leftarrow r_x \cdot r_y + g_x \cdot g_y + b_x \cdot b_y$ 
20:      $D \leftarrow [(A-B)^2 + 4C^2]^{1/2}$ 
21:      $E_{\text{mag}}(\mathbf{u}) \leftarrow [0.5 \cdot (A + B + D)]^{1/2}$      $\triangleright \sqrt{\lambda_1}$ , Eq. 16.27
22:      $E_x(\mathbf{u}) \leftarrow A - B + D$      $\triangleright q_1$ , Eq. 16.28
23:      $E_y(\mathbf{u}) \leftarrow 2C$ 
24:      $E_{\text{nms}}(\mathbf{u}) \leftarrow 0$ 
25:      $E_{\text{bin}}(\mathbf{u}) \leftarrow 0$ 
26:   for  $u \leftarrow 1, \dots, M-2$  do
27:     for  $v \leftarrow 1, \dots, N-2$  do
28:        $\mathbf{u} \leftarrow (u, v)$ 
29:        $d_x \leftarrow E_x(\mathbf{u})$ 
30:        $d_y \leftarrow E_y(\mathbf{u})$ 
31:        $s \leftarrow \text{GetOrientationSector}(d_x, d_y)$      $\triangleright$  Alg. 6.2
32:       if  $\text{IsLocalMax}(E_{\text{mag}}, \mathbf{u}, s, t_{\text{lo}})$  then     $\triangleright$  Alg. 6.2
33:          $E_{\text{nms}}(\mathbf{u}) \leftarrow E_{\text{mag}}(\mathbf{u})$ 
34:   for  $u \leftarrow 1, \dots, M-2$  do
35:     for  $v \leftarrow 1, \dots, N-2$  do
36:        $\mathbf{u} \leftarrow (u, v)$ 
37:       if  $(E_{\text{nms}}(\mathbf{u}) \geq t_{\text{hi}} \wedge E_{\text{bin}}(\mathbf{u}) = 0)$  then
38:          $\text{TraceAndThreshold}(E_{\text{nms}}, E_{\text{bin}}, u, v, t_{\text{lo}})$      $\triangleright$  Alg. 6.2
39:   return  $E_{\text{bin}}$ .
```

16.3 CANNY EDGE DETECTOR FOR COLOR IMAGES

Alg. 16.3

Canny edge detector for color images. Structure and parameters are identical to the grayscale version in Alg. 6.1 (p. 135). In the algorithm below, edge magnitude (E_{mag}) and orientation (E_x, E_y) are obtained from the gradients of the individual color channels (as described in Sec. 16.2.1).

In the pre-processing step, each of the three color channels is individually smoothed by a Gaussian filter of width σ , before calculating the gradient vectors (Alg. 16.3, lines 2–9). As in Alg. 16.2, the color edge magnitude is calculated as the squared local contrast, obtained from the dominant eigenvalue of the structure matrix \mathbf{M} (Eqns. (16.24)–(16.27)). The local gradient vector (E_x, E_y) is calculated from the elements A, B, C , of the matrix \mathbf{M} , as given in Eqn. (16.28). The corresponding steps are found in Alg. 16.3, lines 14–22. The remaining steps, including non-maximum suppression, edge tracing and thresholding, are exactly the same as in Alg. 6.1.

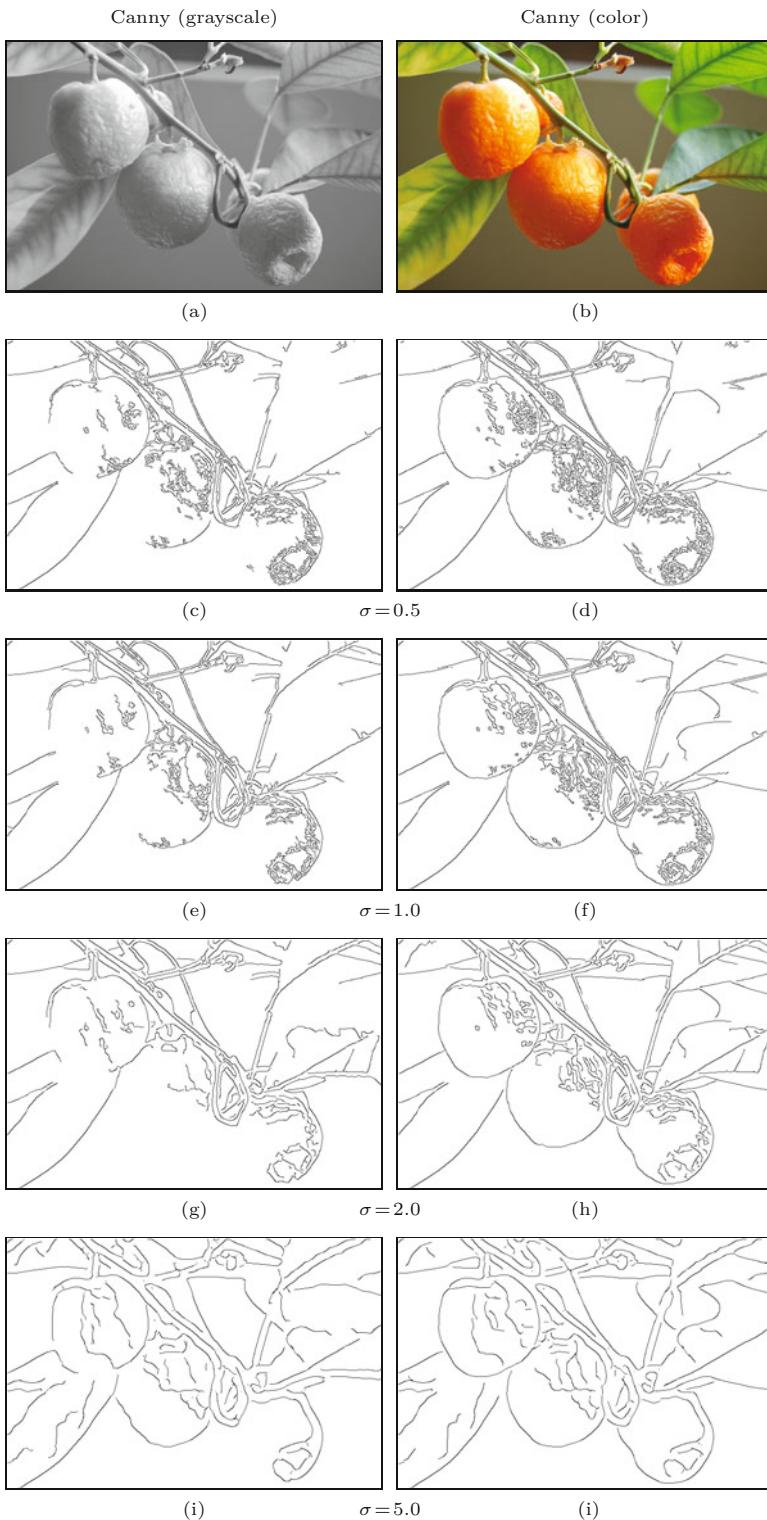
Results from the grayscale and color version of the Canny edge detector are compared in Figs. 16.6 and 16.7 for varying values of σ and t_{hi} , respectively. In all cases, the gradient magnitude was normalized and the threshold values t_{hi}, t_{lo} are given as a percentage of the maximum edge magnitude. Evidently, the color detector gives the more consistent results, particularly at color edges with low intensity difference.

For comparison, Fig. 16.8 shows the results of applying the monochromatic Canny operator separately to each color channel and subsequently merging the edge pixels into a combined edge map, as mentioned at the beginning of this section. We see that this leads to multiple responses and cluttered edges, since maximum gradient positions in the different color channels are generally not collocated.

In summary, the Canny edge detector is superior to simpler schemes based on first-order gradients and global thresholding, in terms of extracting clean and well-located edges that are immediately useful for subsequent processing. The results in Figs. 16.6 and 16.7 demonstrate that the use of color gives additional improvements over the grayscale approach, since edges with insufficient brightness gradients can still be detected from local color differences. Essential for the good performance of the color Canny edge detector, however, is the reliable calculation of the gradient direction, based on the multi-dimensional local contrast formulation given in Sec. 16.2.3. Quite a few variations of Canny detectors for color images have been proposed in the literature, including the one attributed to Kanade (in [140]), which is similar to the algorithm described here.

16.4 Other Color Edge Operators

The idea of using a vector field model in the context of color edge detection was first presented by Di Zenzo [63], who suggested finding the orientation of maximum change by maximizing $S(\mathbf{u}, \theta)$ in Eqn. (16.18) over the angle θ . Later Cumani [59, 60] proposed directly using the eigenvalues and eigenvectors of the local structure matrix \mathbf{M} (Eqn. (16.24)) for calculating edge strength and orientation. He also proposed using the zero-crossings of the second-order gradients along the direction of maximum contrast to precisely locate edges, which is a general problem with first-order techniques. Both Di Zenzo and Cumani used only the dominant eigenvalue, indicating the edge strength perpendicular to the edge (if an edge existed at all), and then discarded the smaller eigenvalue proportional to the edge strength in



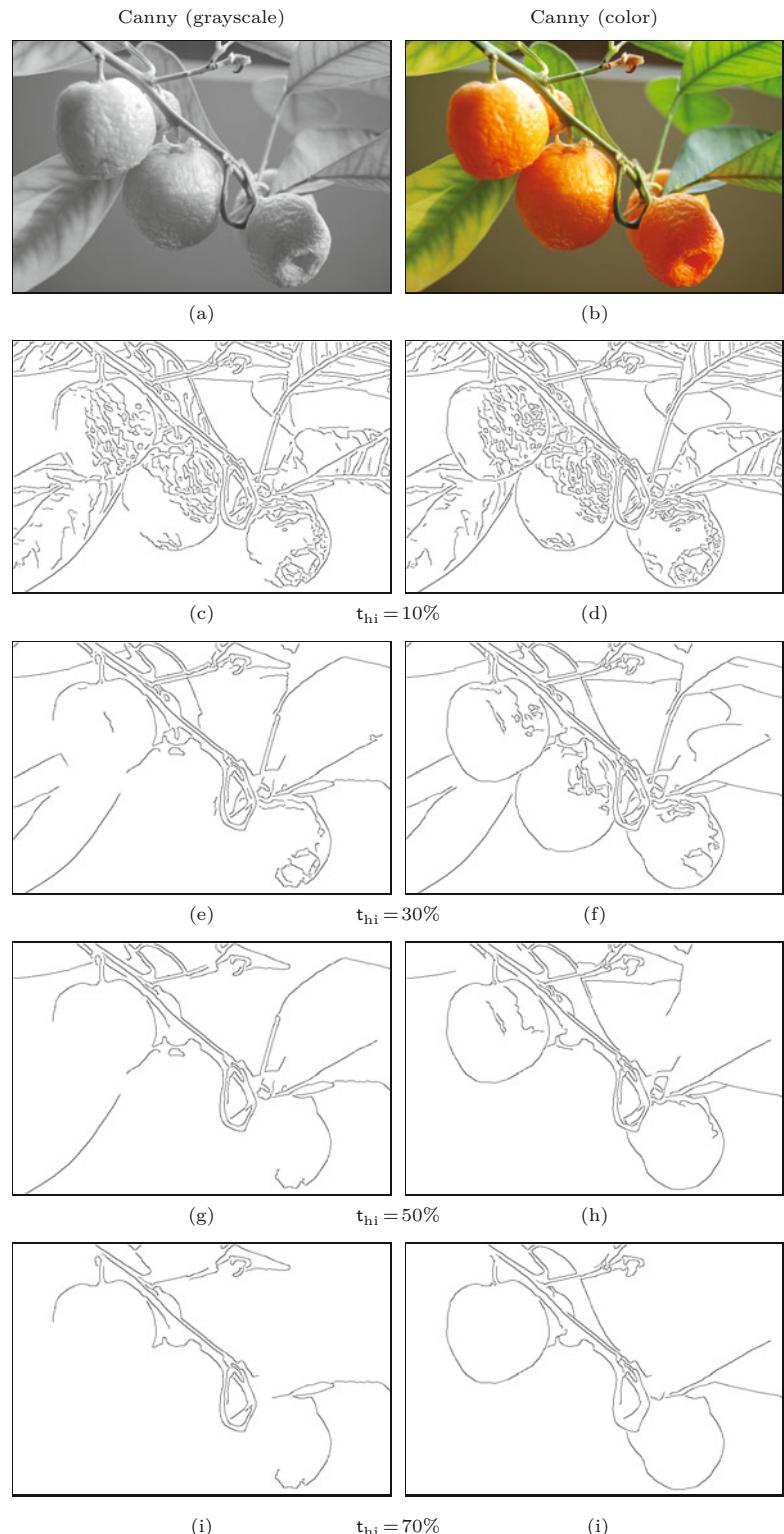
16.4 OTHER COLOR EDGE OPERATORS

Fig. 16.6

Canny grayscale vs. color version. Results from the grayscale (left) and the color version (right) of the Canny operator for different values of σ ($t_{hi} = 20\%$, $t_{lo} = 5\%$ of max. edge magnitude).

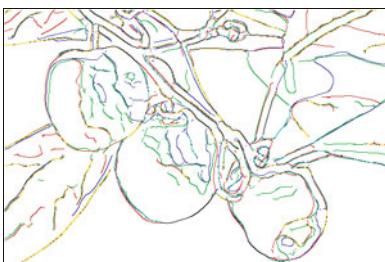
16 EDGE DETECTION IN COLOR IMAGES**Fig. 16.7**

Canny grayscale vs. color version. Results from the grayscale (left) and the color version (right) of the Canny operator for different threshold values t_{hi} , given in % of max. edge magnitude ($t_{lo} = 5\%$, $\sigma = 2.0$).



 $\sigma = 2.0$ 

(a)

 $\sigma = 5.0$ 

(b)



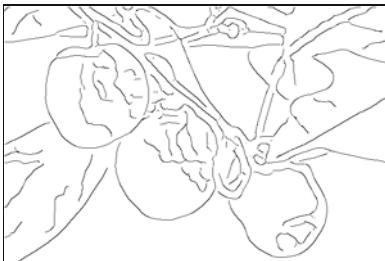
(c)



(d)



(e)



(f)

16.4 OTHER COLOR EDGE OPERATORS

Fig. 16.8

Scalar vs. vector-based color Canny operator. Results from the scalar Canny operator applied separately to each color channel (a, b). Channel edges are shown in corresponding colors, with mixed colors indicating that edge points were detected in multiple channels (e.g., yellow marks overlapping points from the red and the green channel). A black pixel indicates that an edge point was detected in all three color channels. Channel edges combined into a joint edge map (c, d). For comparison, the result of the vector-based color Canny operator (e, f). Common parameter settings are $\sigma = 2.0$ and 5.0 , $t_{hi} = 20\%$, $t_{lo} = 5\%$ of max. edge magnitude.

the perpendicular (i.e., tangential) direction. Real edges only exist where the larger eigenvalue is considerably greater than the smaller one. If both eigenvalues have similar values, this indicates that the local image surface exhibits change in all directions, which is not typically true at an edge but quite characteristic of flat, noisy regions and corners. One solution therefore is to use the difference between the eigenvalues, $\lambda_1 - \lambda_2$, to quantify edge strength [206].

Several color versions of the Canny edge detector can be found in the literature, such as the one proposed by Kanade (in [140]) which is very similar to the algorithm presented here. Other approaches of adapting the Canny detector for color images can be found in [85]. In addition to Canny's scheme, other types of color edge detectors have been used successfully, including techniques based on vector order statistics and color difference vectors. Excellent surveys of the various color edge detection approaches can be found in [266] and [141, Ch. 6].

16.5 Java Implementation

The following Java implementations of the algorithms described in this chapter can be found in the source code section¹⁵ of the book's website. The common (abstract) super-class for all color edge detectors is `ColorEdgeDetector`, which mainly provides the following methods:

`FloatProcessor getEdgeMagnitude()`

Returns the resulting edge magnitude map $E(\mathbf{u})$ as a `FloatProcessor` object.

`FloatProcessor getEdgeOrientation()`

Returns the resulting edge orientation map $\Phi(\mathbf{u})$ as a `FloatProcessor` object, with values in the range $[-\pi, \pi]$.

The following edge detectors are defined as concrete sub-classes of `ColorEdgeDetector`:

`GrayscaleEdgeDetector`: Implements an edge detector that uses only the intensity (brightness) of the supplied color image.

`MonochromaticEdgeDetector`: Implements the monochromatic color edge detector described in Alg. 16.1.

`DiZenzoCumaniEdgeDetector`: Implements the Di Zenzo-Cumani type color edge detector described in Alg. 16.2.

`CannyEdgeDetector`: Implements the canny edge detector for grayscale and color images described in Alg. 16.3. This class defines the additional methods

`ByteProcessor getEdgeBinary()`,

`List<List<java.awt.Point>> getEdgeTraces()`.

Program 16.1 shows a complete example for the use of the class `CannyEdgeDetector` in the context of an ImageJ plugin.

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ByteProcessor;
4 import ij.process.FloatProcessor;
5 import ij.process.ImageProcessor;
6 import imagingbook.pub.coloredge.CannyEdgeDetector;
7
8 import java.awt.Point;
9 import java.util.List;
10
11 public class Canny_Edge_Demo implements PlugInFilter {
12
13     public int setup(String arg0, ImagePlus imp) {
14         return DOES_ALL + NO_CHANGES;
15     }
16
17     public void run(ImageProcessor ip) {
18
19         CannyEdgeDetector.Parameters params =
20             new CannyEdgeDetector.Parameters();
21
22         params.gSigma = 3.0f; //σ of Gaussian
23         params.hiThr = 20.0f; // 20% of max. edge magnitude
24         params.loThr = 5.0f; // 5% of max. edge magnitude
25
26         CannyEdgeDetector detector =
27             new CannyEdgeDetector(ip, params);
28
29         FloatProcessor eMag = detector.getEdgeMagnitude();
30         FloatProcessor eOrt = detector.getEdgeOrientation();
31         ByteProcessor eBin = detector.getEdgeBinary();
32         List<List<Point>> edgeTraces =
33             detector.getEdgeTraces();
34
35         (new ImagePlus("Canny Edges", eBin)).show();
36
37         // process edge detection results ...
38     }
39 }
```

16.5 JAVA IMPLEMENTATION

Prog. 16.1

Use of the `CannyEdgeDetector` class in an ImageJ plugin. A parameter object (`params`) is created in line 20, subsequently configured (in lines 22–24) and finally used to construct a `CannyEdgeDetector` object in line 27. Note that edge detection is performed within the constructor method. Lines 29–33 demonstrate how different types of edge detection results can be retrieved. The binary edge map `eBin` is displayed in line 35. As indicated in the `setup()` method (by returning `DOES_ALL`), this plugin works with any type of image.

Edge-Preserving Smoothing Filters

Noise reduction in images is a common objective in image processing, not only for producing pleasing results for human viewing but also to facilitate easier extraction of meaningful information in subsequent steps, for example, in segmentation or feature detection. Simple smoothing filters, such as the Gaussian filter¹ and the filters discussed in Chapter 15 effectively perform low-pass filtering and thus remove high-frequency noise. However, they also tend to suppress high-rate intensity variations that are part of the original signal, thereby destroying image structures that are visually important. The filters described in this chapter are “edge preserving” in the sense that they change their smoothing behavior adaptively depending upon the local image structure. In general, maximum smoothing is performed over “flat” (uniform) image regions, while smoothing is reduced near or across edge-like structures, typically characterized by high intensity gradients.

In the following, three classical types of edge preserving filters are presented, which are largely based on different strategies. The *Kuwahara-type* filters described in Sec. 17.1 partition the filter kernel into smaller sub-kernels and select the most “homogeneous” of the underlying image regions for calculating the filter’s result. In contrast, the *bilateral* filter in Sec. 17.2 uses the differences between pixel *values* to control how much each individual pixel in the filter region contributes to the local average. Pixels which are similar to the current center pixel contribute strongly, while highly different pixels add little to the result. Thus, in a sense, the bilateral filter is a non-homogeneous linear filter with a convolution kernel that is adaptively controlled by the local image content. Finally, the *anisotropic diffusion* filters in Sec. 17.3 iteratively smooth the image similar to the process of thermal diffusion, using the image gradient to block the local diffusion at edges and similar structures. It should be noted that all filters described in this chapter are nonlinear and can be applied to either grayscale or color images.

¹ See Chapter 5, Sec. 5.2.

17.1 Kuwahara-Type Filters

The filters described in this section are all based on a similar concept that has its early roots in the work of Kuwahara et al. [144]. Although many variations have been proposed by other authors, we summarize them here under the term “Kuwahara-type” to indicate their origin and algorithmic similarities.

In principle, these filters work by calculating the mean and variance within neighboring image regions and selecting the mean value of the most “homogeneous” region, that is, the one with the smallest variance, to replace the original (center) pixel. For this purpose, the filter region R is divided into K partially overlapping subregions R_1, R_2, \dots, R_K . At every image position (u, v) , the *mean* μ_k and the *variance* σ_k^2 of each subregion R_k are calculated from the corresponding pixel values in I as

$$\mu_k(I, u, v) = \frac{1}{|R_k|} \cdot \sum_{(i,j) \in R_k} I(u+i, v+j) = \frac{1}{n_k} \cdot S_{1,k}(I, u, v), \quad (17.1)$$

$$\sigma_k^2(I, u, v) = \frac{1}{|R_k|} \cdot \sum_{(i,j) \in R_k} (I(u+i, v+j) - \mu_k(I, u, v))^2 \quad (17.2)$$

$$= \frac{1}{|R_k|} \cdot \left(S_{2,k}(I, u, v) - \frac{S_{1,k}^2(I, u, v)}{|R_k|} \right), \quad (17.3)$$

for $k = 1, \dots, K$, with²

$$S_{1,k}(I, u, v) = \sum_{(i,j) \in R_k} I(u+i, v+j), \quad (17.4)$$

$$S_{2,k}(I, u, v) = \sum_{(i,j) \in R_k} I^2(u+i, v+j). \quad (17.5)$$

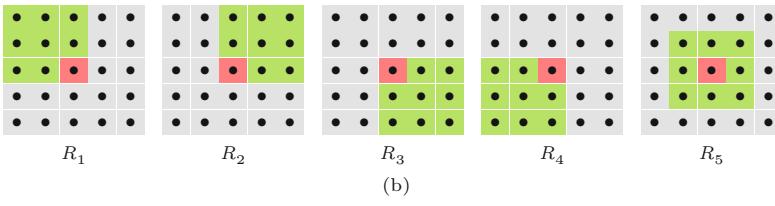
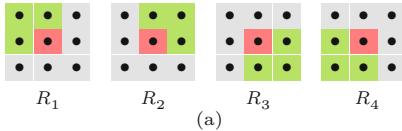
The mean (μ) of the subregion with the smallest variance (σ^2) is selected as the update value, that is,

$$I'(u, v) \leftarrow \mu_{k'}(u, v), \quad \text{with } k' = \operatorname{argmin}_{k=1, \dots, K} \sigma_k^2(I, u, v). \quad (17.6)$$

The subregion structure originally proposed by Kuwahara et al. [144] is shown in Fig. 17.1(a) for a 3×3 filter ($r = 1$). It uses four square subregions of size $(r+1) \times (r+1)$ that overlap at the center. In general, the size of the whole filter is $(2r+1) \times (2r+1)$. This particular filter process is summarized in Alg. 17.1.

Note that this filter does not have a centered subregion, which means that the center pixel is always replaced by the mean of one of the neighboring regions, even if it had perfectly fit the surrounding values. Thus the filter always performs a spatial shift, which introduces jitter and banding artifacts in regions of smooth intensity change. This effect is reduced with the filter proposed by Tomita and Tsuji [230], which is similar but includes a fifth subregion at its center (Fig. 17.1(b)). Filters of arbitrary size can be built by simply scaling the corresponding structure. In case of the *Tomita-Tsuji* filter, the side length of the subregions should be odd.

² $|R_k|$ denotes the size (number of pixels) of the subregion R_k .

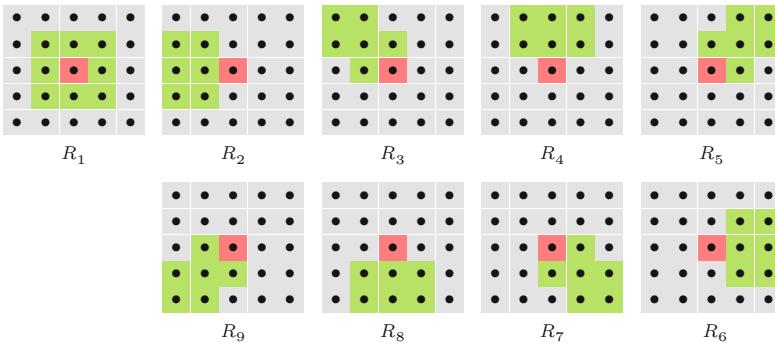


17.1 KUWAHARA-TYPE FILTERS

Fig. 17.1

Subregion structures for Kuwahara-type filters. The original *Kuwahara-Hachimura* filter (a) considers four square, overlapping subregions [144]. *Tomita-Tsuji* filter (b) with five subregions ($r = 2$). The current center pixel (red) is contained in all subregions. Das aktuelle Zentralpixel (rot) ist in allen Subregionen enthalten.

Note that replacing a pixel value by the mean of a square neighborhood is equivalent to linear filtering with a simple box kernel, which is not an optimal smoothing operator. To reduce the artifacts caused by the square subregions, alternative filter structures have been proposed, such as the 5×5 *Nagao-Matsuyama* filter [170] shown in Fig. 17.2.



If all subregions are of identical size $|R_k| = n$, the quantities

$$\sigma_k^2(I, u, v) \cdot n = S_{2,k}(I, u, v) - S_{1,k}^2(I, u, v)/n \quad \text{or} \quad (17.7)$$

$$\sigma_k^2(I, u, v) \cdot n^2 = S_{2,k}(I, u, v) \cdot n - S_{1,k}^2(I, u, v) \quad (17.8)$$

can be used to measure the amount of variation within the corresponding subregion. Both expressions require calculating one multiplication less for each pixel than the “real” variance σ_k^2 in Eqn. (17.3). Moreover, if all subregions have the same *shape* (such as the filters in Fig. 17.1), additional optimizations are possible that substantially improve the performance. In this case, the local mean and variance need to be calculated only once over a fixed neighborhood for each image position. This type of filter can be efficiently implemented by using a set of pre-calculated maps for the local variance and mean values, as described in Alg. 17.2. As before, the parameter r specifies the radius of the composite filter, with subregions of size $(r+1) \times (r+1)$ and overall size $(2r+1) \times (2r+1)$. The individual subregions are of size $(r+1) \times (r+1)$; for example, $r = 2$ for the 5×5 filter shown in Fig. 17.1(b).

All these filters tend to generate banding artifacts in smooth image regions due to erratic spatial displacements, which become worse

Fig. 17.2

Subregions for the 5×5 ($r = 2$) *Nagao-Matsuyama* filter [170]. Note that the centered subregion (R_1) has a different size than the remaining subregions (R_2, \dots, R_9).

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.1
Simple Kuwahara-Hachimura filter.

```

1: KuwaharaFilter( $I$ )
   Input:  $I$ , a grayscale image of size  $M \times N$ .
   Returns a new (filtered) image of size  $M \times N$ .
2:  $R_1 \leftarrow \{(-1, -1), (0, -1), (-1, 0), (0, 0)\}$ 
3:  $R_2 \leftarrow \{(0, -1), (1, -1), (0, 0), (1, 0)\}$ 
4:  $R_3 \leftarrow \{(0, 0), (1, 0), (1, 0), (1, 1)\}$ 
5:  $R_4 \leftarrow \{(-1, 0), (0, 0), (-1, 1), (1, 0)\}$ 
6:  $I' \leftarrow \text{Duplicate}(I)$ 
7:  $(M, N) \leftarrow \text{Size}(I)$ 
8: for all image coordinates  $(u, v) \in M \times N$  do
9:    $\sigma_{\min}^2 \leftarrow \infty$ 
10:  for  $R \leftarrow R_1, \dots, R_4$  do
11:     $(\sigma^2, \mu) \leftarrow \text{EvalSubregion}(I, R, u, v)$ 
12:    if  $\sigma^2 < \sigma_{\min}^2$  then
13:       $\sigma_{\min}^2 \leftarrow \sigma^2$ 
14:       $\mu_{\min} \leftarrow \mu$ 
15:     $I'(u, v) \leftarrow \mu_{\min}$ 
16:  return  $I'$ 

17: EvalSubregion( $I, R, u, v$ )
   Returns the variance and mean of the grayscale image  $I$  for the
   subregion  $R$  positioned at  $(u, v)$ .
18:  $n \leftarrow \text{Size}(R)$ 
19:  $S_1 \leftarrow 0, S_2 \leftarrow 0$ 
20: for all  $(i, j) \in R$  do
21:    $a \leftarrow I(u + i, v + j)$ 
22:    $S_1 \leftarrow S_1 + a$                                  $\triangleright$  Eq. 17.4
23:    $S_2 \leftarrow S_2 + a^2$                              $\triangleright$  Eq. 17.5
24:  $\sigma^2 \leftarrow (S_2 - S_1^2/n)/n$            $\triangleright$  variance of subregion  $R$ , see Eq. 17.1
25:  $\mu \leftarrow S_1/n$                                  $\triangleright$  mean of subregion  $R$ , see Eq. 17.3
26: return  $(\sigma^2, \mu)$ 
```

with increasing filter size. If a centered subregion is used (such as R_5 in Fig. 17.1 or R_1 in Fig. 17.2), one could reduce this effect by applying a threshold (t_σ) to select any off-center subregion R_k *only* if its variance is significantly smaller than the variance of the center region R_1 (see Alg. 17.2, line 13).

17.1.1 Application to Color Images

While all of the aforementioned filters were originally designed for grayscale images, they are easily modified to work with color images. We only need to specify how to calculate the variance and mean for any subregion; the decision and replacement mechanisms then remain the same.

Given an RGB color image $\mathbf{I} = (I_R, I_G, I_B)$ with a subregion R_k , we can calculate the local mean and variance for each color channel as

$$\boldsymbol{\mu}_k(\mathbf{I}, u, v) = \begin{pmatrix} \mu_k(I_R, u, v) \\ \mu_k(I_G, u, v) \\ \mu_k(I_B, u, v) \end{pmatrix}, \quad \boldsymbol{\sigma}_k^2(\mathbf{I}, u, v) = \begin{pmatrix} \sigma_k^2(I_R, u, v) \\ \sigma_k^2(I_G, u, v) \\ \sigma_k^2(I_B, u, v) \end{pmatrix}, \quad (17.9)$$

```

1: FastKuwaharaFilter( $I, r, t_\sigma$ )
   Input:  $I$ , a grayscale image of size  $M \times N$ ;  $r$ , filter radius ( $r \geq 1$ );
           $t_\sigma$ , variance threshold.
   Returns a new (filtered) image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3: Create maps:
    $S : M \times N \rightarrow \mathbb{R}$   $\triangleright$  local variance  $S(u, v) \equiv n \cdot \sigma^2(I, u, v)$ 
    $A : M \times N \rightarrow \mathbb{R}$   $\triangleright$  local mean  $A(u, v) \equiv \mu(I, u, v)$ 
4:  $d_{\min} \leftarrow (r \div 2) - r$   $\triangleright$  subregions' left/top position
5:  $d_{\max} \leftarrow d_{\min} + r$   $\triangleright$  subregions' right/bottom position
6: for all image coordinates  $(u, v) \in M \times N$  do
7:    $(s, \mu) \leftarrow \text{EvalSquareSubregion}(I, u, v, d_{\min}, d_{\max})$ 
8:    $S(u, v) \leftarrow s$ 
9:    $A(u, v) \leftarrow \mu$ 
10:   $n \leftarrow (r + 1)^2$   $\triangleright$  fixed subregion size
11:   $I' \leftarrow \text{Duplicate}(I)$ 
12:  for all image coordinates  $(u, v) \in M \times N$  do
13:     $s_{\min} \leftarrow S(u, v) - t_\sigma \cdot n$   $\triangleright$  variance of center region
14:     $\mu_{\min} \leftarrow A(u, v)$   $\triangleright$  mean of center region
15:    for  $p \leftarrow d_{\min}, \dots, d_{\max}$  do
16:      for  $q \leftarrow d_{\min}, \dots, d_{\max}$  do
17:        if  $S(u + p, v + q) < s_{\min}$  then
18:           $s_{\min} \leftarrow S(u + p, v + q)$ 
19:           $\mu_{\min} \leftarrow A(u + p, v + q)$ 
20:         $I'(u, v) \leftarrow \mu_{\min}$ 
21:    return  $I'$ 


---


22: EvalSquareSubregion( $I, u, v, d_{\min}, d_{\max}$ )
   Returns the variance and mean of the grayscale image  $I$  for a
   square subregion positioned at  $(u, v)$ .
23:  $S_1 \leftarrow 0, S_2 \leftarrow 0$ 
24: for  $i \leftarrow d_{\min}, \dots, d_{\max}$  do
25:   for  $j \leftarrow d_{\min}, \dots, d_{\max}$  do
26:      $a \leftarrow I(u + i, v + j)$ 
27:      $S_1 \leftarrow S_1 + a$   $\triangleright$  Eq. 17.4
28:      $S_2 \leftarrow S_2 + a^2$   $\triangleright$  Eq. 17.5
29:    $s \leftarrow S_2 - S_1^2/n$   $\triangleright$  subregion variance ( $s \equiv n \cdot \sigma^2$ )
30:    $\mu \leftarrow S_1/n$   $\triangleright$  subregion mean ( $\mu$ )
31: return  $(s, \mu)$ 

```

with $\mu_k()$, $\sigma_k^2()$ as defined in Eqns. (17.1) and (17.3), respectively. Analogous to the grayscale case, each pixel is then replaced by the average color in the subregion with the smallest variance, that is,

$$I'(u, v) \leftarrow \mu_{k'}(I, u, v), \quad \text{with } k' = \underset{k=1, \dots, K}{\operatorname{argmin}} \sigma_{k, \text{RGB}}^2(I, u, v). \quad (17.10)$$

The overall variance $\sigma_{k, \text{RGB}}^2$, used to determine k' in Eqn. (17.10), can be defined in different ways, for example, as the sum of the variances in the individual color channels, that is,

$$\sigma_{k, \text{RGB}}^2(I, u, v) = \sigma_k^2(I_R, u, v) + \sigma_k^2(I_G, u, v) + \sigma_k^2(I_B, u, v). \quad (17.11)$$

17.1 KUWAHARA-TYPE FILTERS

Alg. 17.2

Fast Kuwahara-type (Tomita-Tsuji) filter with variable size and fixed subregion structure. The filter uses five square subregions of size $(r+1) \times (r+1)$, with a composite filter of $(2r+1) \times (2r+1)$, as shown in Fig. 17.1(b). The purpose of the variance threshold t_σ is to reduce banding effects in smooth image regions (typically $t_\sigma = 5, \dots, 50$ for 8-bit images).

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.3

Color version of the *Kuwahara-type* filter (adapted from Alg. 17.1). The algorithm uses the definition in Eqn. (17.11) for the total variance σ^2 in the subregion R (see line 25). The vector μ (calculated in line 26) is the average color of the subregion.

<pre> 1: KuwaharaFilterColor(I) 2: Input: I, an RGB image of size $M \times N$. 3: Returns a new (filtered) color image of size $M \times N$. 4: $R_1 \leftarrow \{(-1, -1), (0, -1), (-1, 0), (0, 0)\}$ 5: $R_2 \leftarrow \{(0, -1), (1, -1), (0, 0), (1, 0)\}$ 6: $R_3 \leftarrow \{(0, 0), (1, 0), (1, 0), (1, 1)\}$ 7: $R_4 \leftarrow \{(-1, 0), (0, 0), (-1, 1), (1, 0)\}$ 8: for all image coordinates $(u, v) \in M \times N$ do 9: $\sigma_{\min}^2 \leftarrow \infty$ 10: for $R \leftarrow R_1, \dots, R_4$ do 11: $(\sigma^2, \mu) \leftarrow \text{EvalSubregion}(I, R_k, u, v)$ 12: if $\sigma^2 < \sigma_{\min}^2$ then 13: $\sigma_{\min}^2 \leftarrow \sigma^2$ 14: $\mu_{\min} \leftarrow \mu$ 15: $I'(u, v) \leftarrow \mu_{\min}$ 16: return I' </pre>	<p>17: EvalSubregion(I, R, u, v) Returns the total variance and the mean vector of the color image I for the subregion R positioned at (u, v).</p> <pre> 18: $n \leftarrow \text{Size}(R)$ 19: $S_1 \leftarrow \mathbf{0}, S_2 \leftarrow \mathbf{0}$ $\triangleright S_1, S_2 \in \mathbb{R}^3$ 20: for all $(i, j) \in R$ do 21: $a \leftarrow I(u+i, v+j)$ $\triangleright a \in \mathbb{R}^3$ 22: $S_1 \leftarrow S_1 + a$ 23: $S_2 \leftarrow S_2 + a^2$ $\triangleright a^2 = a \cdot a$ (dot product) 24: $S \leftarrow (S_2 - S_1^2 \cdot \frac{1}{n}) \cdot \frac{1}{n}$ $\triangleright S = (\sigma_R^2, \sigma_G^2, \sigma_B^2)$ 25: $\sigma_{\text{RGB}}^2 \leftarrow \Sigma S$ $\triangleright \sigma_{\text{RGB}}^2 = \sigma_R^2 + \sigma_G^2 + \sigma_B^2$, total variance in R 26: $\mu \leftarrow \frac{1}{n} \cdot S_1$ $\triangleright \mu \in \mathbb{R}^3$, avg. color vector for subregion R 27: return $(\sigma_{\text{RGB}}^2, \mu)$ </pre>
--	---

This is sometimes called the “total variance”. The resulting filter process is summarized in Alg. 17.3 and color examples produced with this algorithm are shown in Figs. 17.3 and 17.4.

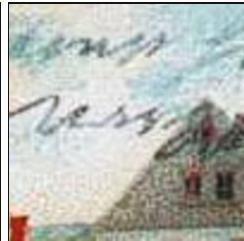
Alternatively [109], one could define the combined color variance as the norm of the *color covariance matrix*³ for the subregion R_k ,

$$\Sigma_k(I, u, v) = \begin{pmatrix} \sigma_{k,RR} & \sigma_{k,RG} & \sigma_{k,RB} \\ \sigma_{k,GR} & \sigma_{k,GG} & \sigma_{k,GB} \\ \sigma_{k,BR} & \sigma_{k,BG} & \sigma_{k,BB} \end{pmatrix}, \quad (17.12)$$

$$\text{with } \sigma_{k,pq} = \frac{1}{|R_k|} \cdot \sum_{(i,j) \in R_k} [I_p(u+i, v+j) - \mu_k(I_p, u, v)] \cdot [I_q(u+i, v+j) - \mu_k(I_q, u, v)], \quad (17.13)$$

for all possible color pairs $(p, q) \in \{\text{R, G, B}\}^2$. Note that $\sigma_{k,pp} = \sigma_{k,p}^2$ and $\sigma_{k,pq} = \sigma_{k,qp}$, and thus the matrix Σ_k is symmetric and only 6 of its 9 entries need to be calculated. The (Frobenius) *norm* of the 3×3 color covariance matrix is defined as

³ See Sec. D.2 in the Appendix for details.



(a) RGB test image with selected details



(b) $r = 1$ (3×3 filter)



(c) $r = 2$ (5×5 filter)



(d) $r = 3$ (7×7 filter)



(e) $r = 4$ (9×9 filter)

17.1 KUWAHARA-TYPE FILTERS

Fig. 17.3
Kuwahara-type (*Tomita-Tsuji*) filter—color example using the variance definition in Eqn. (17.11). The filter radius is varied from $r = 1$ (b) to $r = 4$ (e).

$$\sigma_{k,\text{RGB}}^2 = \|\Sigma_k(\mathbf{I}, u, v)\|_2^2 = \sum_{\substack{p,q \in \\ \{\text{R,G,B}\}}} (\sigma_{k,pq})^2 \quad (17.14)$$

Note that the total variance in Eqn. (17.11)—which is simpler to calculate than this norm—is equivalent to the *trace* of the covariance matrix Σ_k .

Fig. 17.4

Color versions of the *Tomita-Tsuji* (Fig. 17.1(b)) and *Nagao-Matsuyama* filter (Fig. 17.2). Both filters are of size 5×5 and use the variance definition in Eqn. (17.11). Results are visually similar, but in general the *Nagao-Matsuyama* filter is slightly less destructive on diagonal structures. Original image in Fig. 17.3(a).


 (a) 5×5 Tomita-Tsuji filter ($r = 2$)

 (b) 5×5 Nagao-Matsuyama filter

Since each pixel of the filtered image is calculated as the *mean* (i.e., a linear combination) of a set of original color pixels, the results depend on the color space used, as discussed in Chapter 15, Sec. 15.1.2.

17.2 Bilateral Filter

Traditional linear smoothing filters operate by convolving the image with a kernel, whose coefficients act as weights for the corresponding image pixels and only depend on the spatial distance from the center coordinate. Pixels close to the filter center are typically given larger weights while pixels at a greater distance carry smaller weights. Thus the convolution kernel effectively encodes the closeness of the underlying pixels in space. In the following, a filter whose weights depend only on the distance in the spatial domain is called a *domain filter*.

To make smoothing filters less destructive on edges, a typical strategy is to exclude individual pixels from the filter operation or to reduce the weight of their contribution if they are very dissimilar *in value* to the pixel found at the center position. This operation too can be formulated as a filter, but this time the kernel coefficients depend only upon the differences in pixel *values* or *range*. Therefore this is called a *range filter*, as explained in more detail Sec. 17.2.2. The idea of the *bilateral filter*, proposed by Tomasi and Manduchi in [229], is to *combine* both domain and range filtering into a common, edge-preserving smoothing filter.

17.2.1 Domain Filter

In an ordinary 2D linear filter (or “convolution”) operation,⁴

⁴ See also Chapter 5, Eqn. (5.5) on page 92.

$$I'(u, v) \leftarrow \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I(u+m, v+n) \cdot H(m, n) \quad (17.15)$$

$$= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot H(i-u, j-v), \quad (17.16)$$

every new pixel value $I'(u, v)$ is the weighted average of the original image pixels I in a certain neighborhood, with the weights specified by the elements of the filter kernel H .⁵ The weight assigned to each pixel only depends on its spatial position relative to the current center coordinate (u, v) . In particular, $H(0, 0)$ specifies the weight of the center pixel $I(u, v)$, and $H(m, n)$ is the weight assigned to a pixel displaced by (m, n) from the center. Since only the spatial image coordinates are relevant, such a filter is called a *domain filter*. Obviously, ordinary filters as we know them are *all* domain filters.

17.2.2 Range Filter

Although the idea may appear strange at first, one could also apply a linear filter to the pixel *values* or *range* of an image in the form

$$I'_r(u, v) \leftarrow \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot H_r(I(i, j) - I(u, v)). \quad (17.17)$$

The contribution of each pixel is specified by the function H_r and depends on the difference between its own *value* $I(i, j)$ and the value at the current center pixel $I(u, v)$. The operation in Eqn. (17.17) is called a *range filter*, where the spatial position of a contributing pixel is irrelevant and only the difference in values is considered. For a given position (u, v) , all surrounding image pixels $I(i, j)$ with the same value contribute equally to the result $I'_r(u, v)$. Consequently, the application of a *range filter* has no *spatial* effect upon the image—in contrast to a *domain filter*, no blurring or sharpening will occur. Instead, a range filter effectively performs a global *point operation* by remapping the intensity or color values. However, a global *range filter* by itself is of little use, since it combines pixels from the entire image and only changes the intensity or color map of the image, equivalent to a nonlinear, image-dependent point operation.

17.2.3 Bilateral Filter—General Idea

The key idea behind the bilateral filter is to *combine* domain filtering (Eqn. (17.16)) *and* range filtering (Eqn. (17.17)) in the form

$$I'(u, v) = \frac{1}{W_{u,v}} \cdot \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot \underbrace{H_d(i-u, j-v) \cdot H_r(I(i, j) - I(u, v))}_{w_{i,j}}, \quad (17.18)$$

⁵ In Eqn. (17.16), functions I and H are assumed to be zero outside their domains of definition.

where H_d , H_r are the *domain* and *range* kernels, respectively, $w_{i,j}$ are the composite weights, and

$$W_{u,v} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} w_{i,j} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} H_d(i-u, j-v) \cdot H_r(I(i,j) - I(u,v)) \quad (17.19)$$

is the (position-dependent) sum of the weights $w_{i,j}$ used to normalize the combined filter kernel.

In this form, the scope of range filtering is constrained to the spatial neighborhood defined by the domain kernel H_d . At a given filter position (u, v) , the weight $w_{i,j}$ assigned to each contributing pixel depends upon (1) its spatial position relative to (u, v) , and (2) the similarity of its pixel value to the value at the center position (u, v) . In other words, the resulting pixel is the weighted average of pixels that are nearby *and* similar to the original pixel. In a flat image region, where most surrounding pixels have values similar to the center pixel, the bilateral filter acts as a conventional smoothing filter, controlled only by the domain kernel H_d . However, when placed near a step edge or on an intensity ridge, only those pixels are included in the smoothing process that are similar in value to the center pixel, thus avoiding blurring the edges.

If the domain kernel H_d has a limited radius D , or size $(2D+1) \times (2D+1)$, the bilateral filter defined in Eqn. (17.18) can be written as

$$I'(u,v) = \frac{\sum_{i=u-D}^{u+D} \sum_{j=v-D}^{v+D} I(i,j) \cdot H_d(i-u, j-v) \cdot H_r(I(i,j) - I(u,v))}{\sum_{i=u-D}^{u+D} \sum_{j=v-D}^{v+D} H_d(i-u, j-v) \cdot H_r(I(i,j) - I(u,v))} \quad (17.20)$$

$$= \frac{\sum_{m=-D}^D \sum_{n=-D}^D I(u+m, v+n) \cdot H_d(m,n) \cdot H_r(I(u+m, v+n) - I(u,v))}{\sum_{m=-D}^D \sum_{n=-D}^D H_d(m,n) \cdot H_r(I(u+m, v+n) - I(u,v))} \quad (17.21)$$

(by substituting $(i-u) \rightarrow m$ and $(j-v) \rightarrow n$). The effective, space variant filter kernel for the image I at position (u, v) then is

$$\bar{H}_{I,u,v}(i,j) = \frac{H_d(i,j) \cdot H_r(I(u+i, v+j) - I(u,v))}{\sum_{m=-D}^D \sum_{n=-D}^D H_d(m,n) \cdot H_r(I(u+m, v+n) - I(u,v))}, \quad (17.22)$$

for $-D \leq i, j \leq D$, whereas $\bar{H}_{I,u,v}(i,j) = 0$ otherwise. This quantity specifies the contribution of the original image pixels $I(u+i, v+j)$ to the resulting new pixel value $I'(u, v)$.

17.2.4 Bilateral Filter with Gaussian Kernels

17.2 BILATERAL FILTER

A special (but common) case is the use of Gaussian kernels for both the domain and the range parts of the bilateral filter. The discrete 2D Gaussian *domain kernel* of width σ_d is defined as

$$H_d^{G,\sigma_d}(m, n) = \frac{1}{2\pi\sigma_d^2} \cdot e^{-\frac{\rho^2}{2\sigma_d^2}} = \frac{1}{2\pi\sigma_d^2} \cdot e^{-\frac{m^2+n^2}{2\sigma_d^2}} \quad (17.23)$$

$$= \frac{1}{\sqrt{2\pi}\sigma_d} \cdot \exp\left(-\frac{m^2}{2\sigma_d^2}\right) \cdot \frac{1}{\sqrt{2\pi}\sigma_d} \cdot \exp\left(-\frac{n^2}{2\sigma_d^2}\right), \quad (17.24)$$

for $m, n \in \mathbb{Z}$. It has its maximum at the center ($m = n = 0$) and declines smoothly and isotropically with increasing radius $\rho = \sqrt{m^2 + n^2}$; for $\rho > 3.5\sigma_d$, $H_d^{G,\sigma_d}(m, n)$ is practically zero. The factorization in Eqn. (17.24) indicates that the Gaussian 2D kernel can be separated into 1D Gaussians, allowing for a more efficient implementation.⁶ The constant factors $1/(\sqrt{2\pi}\sigma_d)$ can be omitted in practice, since the bilateral filter requires individual normalization at each image position (Eqn. (17.19)).

Similarly, the corresponding *range filter kernel* is defined as a (continuous) 1D Gaussian of width σ_r ,

$$H_r^{G,\sigma_r}(x) = \frac{1}{\sqrt{2\pi}\sigma_r} \cdot e^{-\frac{x^2}{2\sigma_r^2}} = \frac{1}{\sqrt{2\pi}\sigma_r} \cdot \exp\left(-\frac{x^2}{2\sigma_r^2}\right), \quad (17.25)$$

for $x \in \mathbb{R}$. The constant factor $1/(\sqrt{2\pi}\sigma_r)$ may again be omitted and the resulting composite filter (Eqn. (17.18)) can thus be written as

$$I'(u, v) = \frac{1}{W_{u,v}} \cdot \sum_{\substack{i=1 \\ u-D}}^{u+D} \sum_{\substack{j=1 \\ v-D}}^{v+D} \left[I(i, j) \cdot H_d^{G,\sigma_d}(i - u, j - v) \cdot H_r^{G,\sigma_r}(I(i, j) - I(u, v)) \right] \quad (17.26)$$

$$= \frac{1}{W_{u,v}} \cdot \sum_{\substack{m=1 \\ -D}}^D \sum_{\substack{n=1 \\ -D}}^D \left[I(u + m, v + n) \cdot H_d^{G,\sigma_d}(m, n) \cdot H_r^{G,\sigma_r}(I(u + m, v + n) - I(u, v)) \right] \quad (17.27)$$

$$= \frac{1}{W_{u,v}} \cdot \sum_{\substack{m=1 \\ -D}}^D \sum_{\substack{n=1 \\ -D}}^D \left[I(u + m, v + n) \cdot \exp\left(-\frac{m^2+n^2}{2\sigma_d^2}\right) \cdot \exp\left(-\frac{(I(u+m, v+n) - I(u, v))^2}{2\sigma_r^2}\right) \right], \quad (17.28)$$

with $D = \lceil 3.5 \cdot \sigma_d \rceil$ and

$$W_{u,v} = \sum_{\substack{m=1 \\ -D}}^D \sum_{\substack{n=1 \\ -D}}^D \exp\left(-\frac{m^2+n^2}{2\sigma_d^2}\right) \cdot \exp\left(-\frac{(I(u+m, v+n) - I(u, v))^2}{2\sigma_r^2}\right). \quad (17.29)$$

For 8-bit grayscale images, with pixel values in the range $[0, 255]$, the width of the range kernel is typically set to $\sigma_r = 10, \dots, 50$. The width of the domain kernel (σ_d) depends on the desired amount of spatial smoothing. Algorithm 17.4 gives a summary of the steps involved in bilateral filtering for grayscale images.

⁶ See also Chapter 5, Sec. 5.3.3.

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.4

Bilateral filter with Gaussian kernels (grayscale version).

```

1: BilateralFilterGray( $I, \sigma_d, \sigma_r$ )
   Input:  $I$ , a grayscale image of size  $M \times N$ ;  $\sigma_d$ , width of the 2D
          Gaussian domain kernel;  $\sigma_r$ , width of the 1D Gaussian range
          kernel. Returns a new filtered image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $D \leftarrow \lceil 3.5 \cdot \sigma_d \rceil$                                  $\triangleright$  width of domain filter kernel
4:  $I' \leftarrow \text{Duplicate}(I)$ 
5: for all image coordinates  $(u, v) \in M \times N$  do
6:    $S \leftarrow 0$                                           $\triangleright$  sum of weighted pixel values
7:    $W \leftarrow 0$                                           $\triangleright$  sum of weights
8:    $a \leftarrow I(u, v)$                                       $\triangleright$  center pixel value
9:   for  $m \leftarrow -D, \dots, D$  do
10:    for  $n \leftarrow -D, \dots, D$  do
11:       $b \leftarrow I(u + m, v + n)$                           $\triangleright$  off-center pixel value
12:       $w_d \leftarrow \exp\left(-\frac{m^2 + n^2}{2\sigma_d^2}\right)$   $\triangleright$  domain coefficient
13:       $w_r \leftarrow \exp\left(-\frac{(a-b)^2}{2\sigma_r^2}\right)$   $\triangleright$  range coefficient
14:       $w \leftarrow w_d \cdot w_r$                                 $\triangleright$  composite coefficient
15:       $S \leftarrow S + w \cdot b$ 
16:       $W \leftarrow W + w$ 
17:    $I'(u, v) \leftarrow S/W$ 
18: return  $I'$ 

```

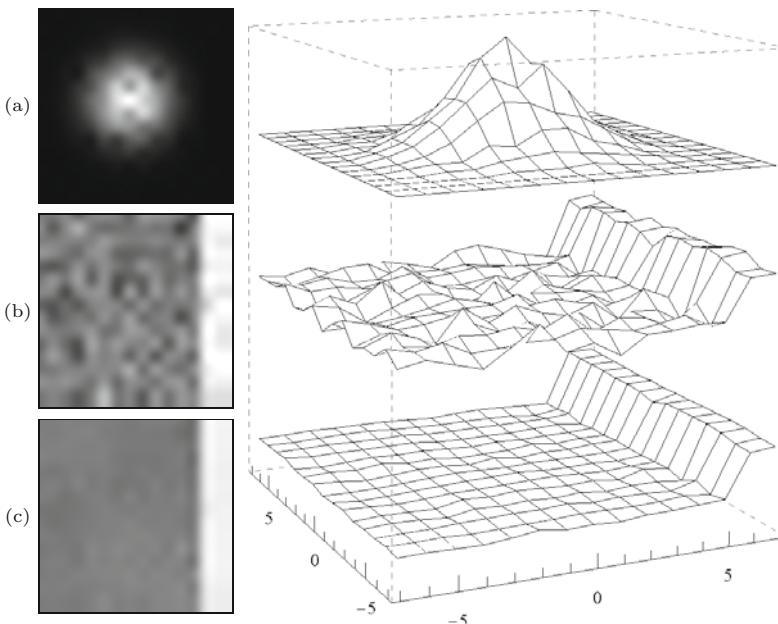
Figures 17.5–17.9 show the effective, space-variant filter kernels (see Eqn. (17.22)) and the results of applying a bilateral filter with Gaussian domain and range kernels in different situations. Uniform noise was applied to the original images to demonstrate the filtering effect. One can see clearly how the range part makes the combined filter kernel adapt to the local image structure. Only those surrounding parts that have brightness values similar to the center pixel are included in the filter operation. The filter parameters were set to $\sigma_d = 2.0$ and $\sigma_r = 50$; the domain kernel is of size 15×15 .

17.2.5 Application to Color Images

Linear smoothing filters are typically used on color images by separately applying the same filter to the individual color channels. As discussed in Chapter 15, Sec. 15.1, this is legitimate if a suitable working color space is used to avoid the introduction of unnatural intensity and chromaticity values. Thus, for the domain-part of the bilateral filter, the same considerations apply as for any linear smoothing filter. However, as will be described, the bilateral filter as a whole cannot be implemented by filtering the color channels separately.

In the *range* part of the filter, the weight assigned to each contributing pixel depends on its difference to the value of the center pixel. Given a suitable distance measure $\text{dist}(\mathbf{a}, \mathbf{b})$ between two color vectors \mathbf{a}, \mathbf{b} , the bilateral filter in Eqn. (17.18) can be easily modified for a color image \mathbf{I} to

$$\mathbf{I}'(u, v) = \frac{1}{W_{u,v}} \cdot \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \mathbf{I}(i, j) \cdot H_d(i-u, j-v) \cdot H_r(\text{dist}(\mathbf{I}(i, j), \mathbf{I}(u, v))), \quad (17.30)$$



17.2 BILATERAL FILTER

Fig. 17.5

Bilateral filter response when positioned in a flat, noisy image region. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

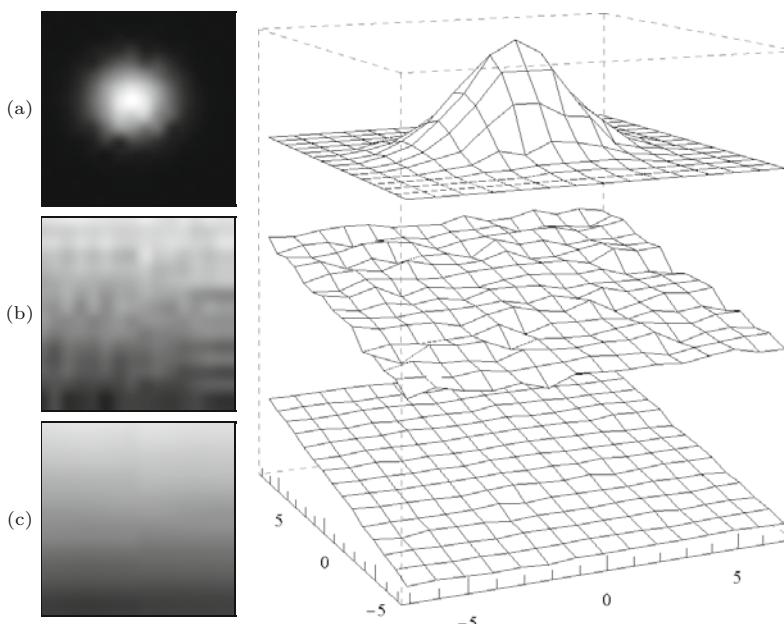


Fig. 17.6

Bilateral filter response when positioned on a linear ramp. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

with

$$W_{u,v} = \sum_{i,j} H_d(i-u, j-v) \cdot H_r(\text{dist}(\mathbf{I}(i,j), \mathbf{I}(u,v))). \quad (17.31)$$

It is common to use one of the popular norms for measuring color distances, such as the L_1 , L_2 (Euclidean), or the L_∞ (maximum) norms, for example,

17 EDGE-PRESERVING SMOOTHING FILTERS

Fig. 17.7

Bilateral filter response when positioned left to a vertical step edge. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

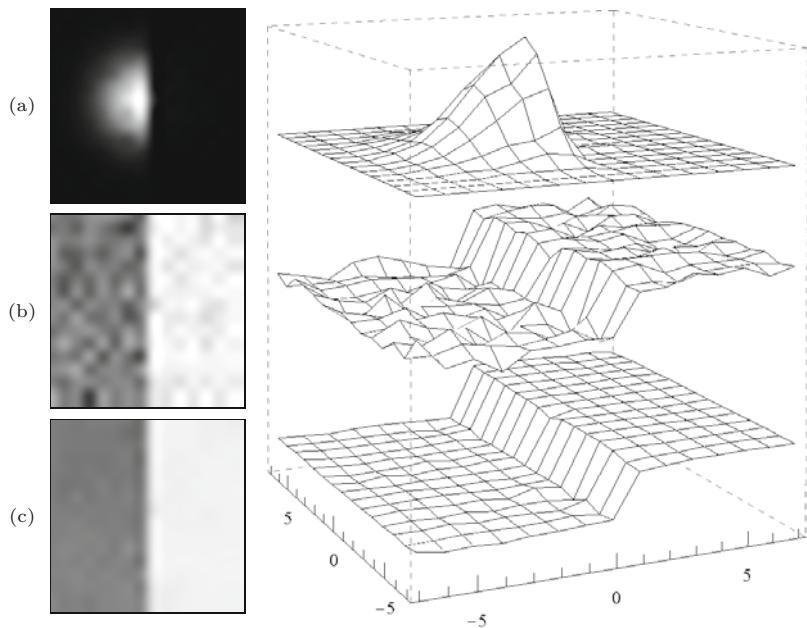
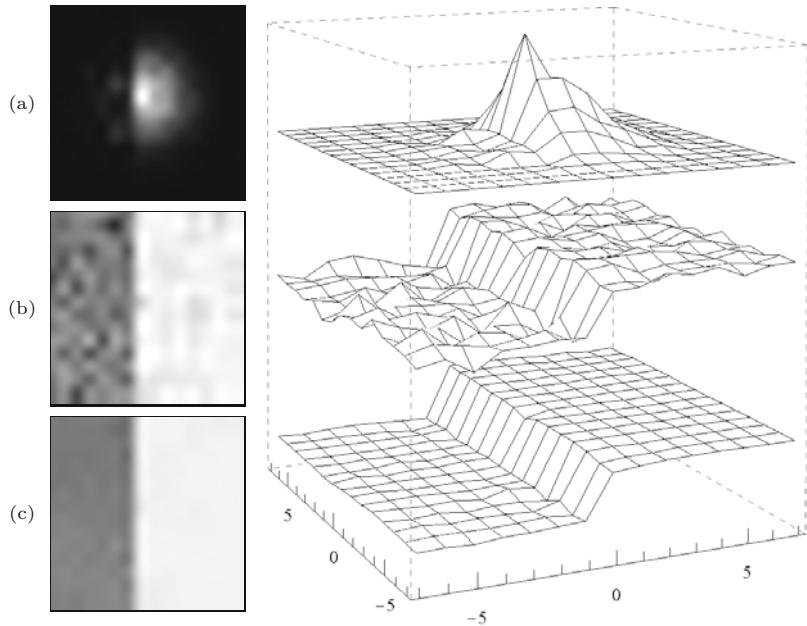


Fig. 17.8

Bilateral filter response when positioned right to a vertical step edge. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

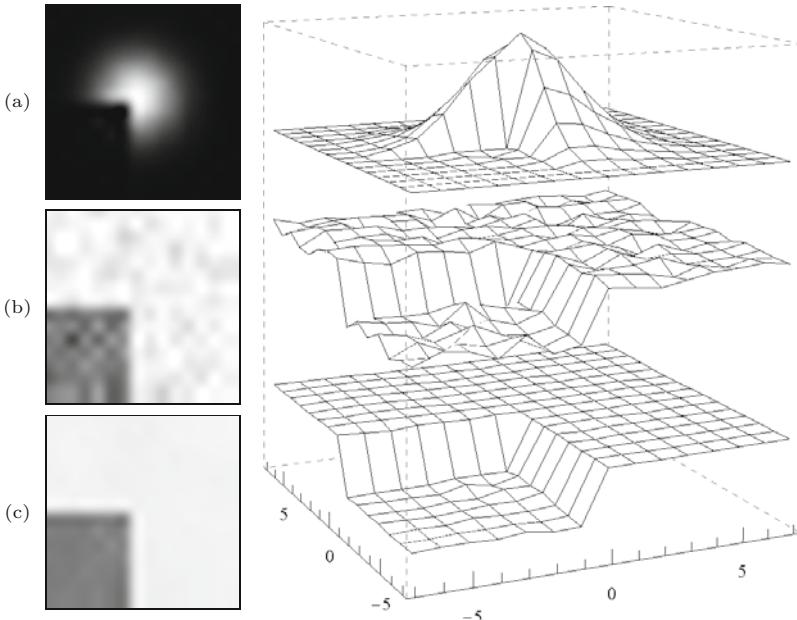


$$\text{dist}_1(\mathbf{a}, \mathbf{b}) := \frac{1}{3} \cdot \|\mathbf{a} - \mathbf{b}\|_1 = \frac{1}{3} \cdot \sum_{k=1}^K |a_k - b_k|, \quad (17.32)$$

$$\text{dist}_2(\mathbf{a}, \mathbf{b}) := \frac{1}{\sqrt{3}} \cdot \|\mathbf{a} - \mathbf{b}\|_2 = \frac{1}{\sqrt{3}} \cdot \left(\sum_{k=1}^K (a_k - b_k)^2 \right)^{1/2}, \quad (17.33)$$

$$\text{dist}_{\infty}(\mathbf{a}, \mathbf{b}) := \|\mathbf{a} - \mathbf{b}\|_{\infty} = \max_k |a_k - b_k|. \quad (17.34)$$

The normalizing factors $1/3$ and $1/\sqrt{3}$ in Eqns. (17.32)–(17.33) are necessary to obtain results comparable in magnitude to those of



17.2 BILATERAL FILTER

Fig. 17.9

Bilateral filter response when positioned at a corner. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

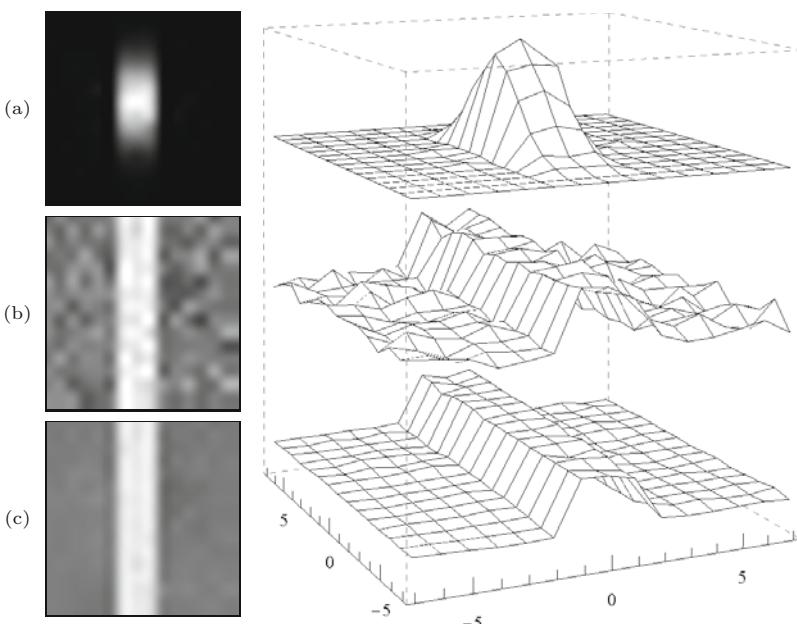


Fig. 17.10

Bilateral filter response when positioned on a vertical ridge. Original image function (b), filtered image (c), combined impulse response (a) of the filter at the given position.

grayscale images when using the same range kernel H_r .⁷ Of course in most color spaces, none of these norms measures perceived color difference.⁸ However, the distance function itself is not really critical since it only affects the relative *weights* assigned to the contributing

⁷ For example, with 8-bit RGB color images, $\text{dist}(a, b)$ is always in the range [0, 255].

⁸ The CIELAB and CIELUV color spaces are designed to use the Euclidean distance (L_2 norm) as a valid metric for color difference.

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.5

Bilateral filter with Gaussian kernels (color version). The function $\text{dist}(\mathbf{a}, \mathbf{b})$ measures the distance between two colors \mathbf{a} and \mathbf{b} , for example, using the L_2 norm (line 5, see Eqns. (17.32)–(17.34) for other options).

```

1: BilateralFilterColor( $I, \sigma_d, \sigma_r$ )
   Input:  $I$ , a color image of size  $M \times N$ ;  $\sigma_d$ , width of the 2D Gaussian domain kernel;  $\sigma_r$ , width of the 1D Gaussian range kernel. Returns a new filtered color image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $D \leftarrow \lceil 3.5 \cdot \sigma_d \rceil$                                  $\triangleright$  width of domain filter kernel
4:  $I' \leftarrow \text{Duplicate}(I)$ 
5:  $\text{dist}(\mathbf{a}, \mathbf{b}) := \frac{1}{\sqrt{3}} \cdot \|\mathbf{a} - \mathbf{b}\|_2$        $\triangleright$  color distance (e.g., Euclidean)
6: for all image coordinates  $(u, v) \in (M \times N)$  do
7:    $S \leftarrow \mathbf{0}$                        $\triangleright S \in \mathbb{R}^3$ , sum of weighted pixel vectors
8:    $W \leftarrow 0$                          $\triangleright$  sum of pixel weights (scalar)
9:    $\mathbf{a} \leftarrow I(u, v)$                  $\triangleright \mathbf{a} \in \mathbb{R}^3$ , center pixel vector
10:  for  $m \leftarrow -D, \dots, D$  do
11:    for  $n \leftarrow -D, \dots, D$  do
12:       $\mathbf{b} \leftarrow I(u + m, v + n)$   $\triangleright \mathbf{b} \in \mathbb{R}^3$ , off-center pixel vector
13:       $w_d \leftarrow \exp\left(-\frac{m^2 + n^2}{2\sigma_d^2}\right)$            $\triangleright$  domain coefficient
14:       $w_r \leftarrow \exp\left(-\frac{(\text{dist}(\mathbf{a}, \mathbf{b}))^2}{2\sigma_r^2}\right)$            $\triangleright$  range coefficient
15:       $w \leftarrow w_d \cdot w_r$                      $\triangleright$  composite coefficient
16:       $S \leftarrow S + w \cdot \mathbf{b}$ 
17:       $W \leftarrow W + w$ 
18:       $I'(u, v) \leftarrow \frac{1}{W} \cdot S$ 
19:  return  $I'$ 
```

color pixels. Regardless of the distance function used, the resulting chromaticities are linear, convex combinations of the original colors in the filter region, and thus the choice of the working color space is more important (see Chapter 15, Sec. 15.1).

The process of bilateral filtering for color images (again using Gaussian kernels for the domain and the range filters) is summarized in Alg. 17.5. The Euclidean distance (L_2 norm) is used to measure the difference between color vectors. The examples in Fig. 17.11 were produced using sRGB as the color working space.

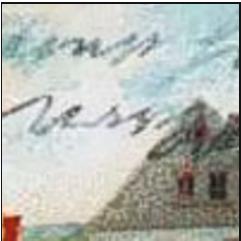
17.2.6 Efficient Implementation by x/y Separation

The bilateral filter, if implemented in the way described in Algs. 17.4–17.5, is computationally expensive, with a time complexity of $\mathcal{O}(K^2)$ for each pixel, where K denotes the radius of the filter. Some mild speedup is possible by tabulating the domain and range kernels, but the performance of the brute-force implementation is usually not acceptable for practical applications. In [185] a separable *approximation* of the bilateral filter is proposed that brings about a significant performance increase. In this implementation, a 1D bilateral filter is first applied in the horizontal direction only, which uses 1D domain and range kernels h_d and h_r , respectively, and produces the intermediate image I^\triangleright , that is,

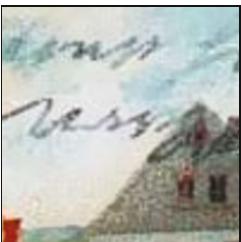
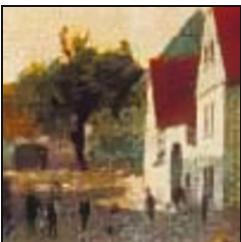
17.2 BILATERAL FILTER

Fig. 17.11

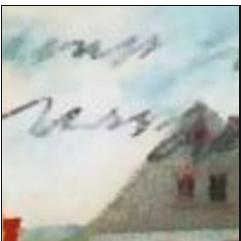
Bilateral filter—color example. A Gaussian kernel with $\sigma_d = 2.0$ (kernel size 15×15) is used for the domain part of the filter; working color space is sRGB. The width of the range filter is varied from $\sigma_r = 10$ to 100. The filter was applied in sRGB color space.



(a) $\sigma_r = 10$



(b) $\sigma_r = 20$



(c) $\sigma_r = 50$



(d) $\sigma_r = 100$

$$I^{\triangleright}(u, v) = \frac{\sum_{m=-D}^D I(u+m, v) \cdot h_d(m) \cdot h_r(I(u+m, v) - I(u, v))}{\sum_{m=-D}^D h_d(m) \cdot h_r(I(u+m, v) - I(u, v))} \quad (17.35)$$

In the second pass, the *same* filter is applied to the intermediate result I^{\triangleright} in the vertical direction to obtain the final result I' as

$$I'(u, v) = \frac{\sum_{n=-D}^D I^{\triangleright}(u, v+n) \cdot h_d(n) \cdot h_r(I^{\triangleright}(u, v+n) - I^{\triangleright}(u, v))}{\sum_{n=-D}^D h_d(n) \cdot h_r(I^{\triangleright}(u, v+n) - I^{\triangleright}(u, v))} \quad (17.36)$$

for all (u, v) , using the same 1D domain and range kernels h_d and h_r , respectively, as in Eqn. (17.35).

For the *horizontal* part of the filter, the effective space-variant kernel at image position (u, v) is

$$\bar{h}_{I,u,v}^{\triangleright}(i) = \frac{h_d(i) \cdot h_r(I(u+i, v) - I(u, v))}{\sum_{m=-D}^D h_d(m) \cdot h_r(I(u+m, v) - I(u, v))}, \quad (17.37)$$

for $-D \leq i \leq D$ (zero otherwise). Analogously, the effective kernel for the *vertical* part of the filter is

$$\bar{h}_{I,u,v}^{\nabla}(j) = \frac{h_d(j) \cdot h_r(I(u, v+j) - I(u, v))}{\sum_{n=-D}^D h_d(n) \cdot h_r(I(u, v+n) - I(u, v))}, \quad (17.38)$$

again for $-D \leq j \leq D$. For the *combined* filter, the effective 2D kernel at position (u, v) then is

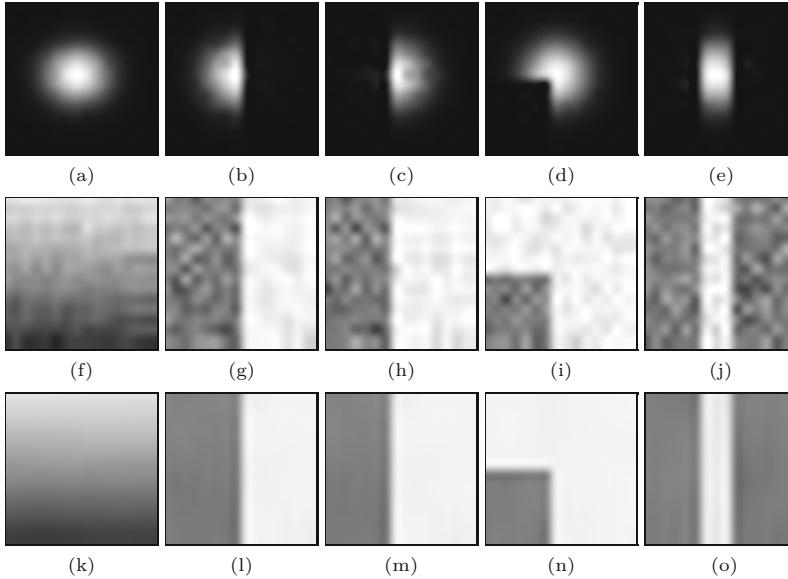
$$\bar{H}_{I,u,v}(i, j) = \begin{cases} \bar{h}_{I,u,v}^{\triangleright}(i) \cdot \bar{h}_{I,u,v}^{\nabla}(j) & \text{for } -D \leq i, j \leq D, \\ 0 & \text{otherwise,} \end{cases} \quad (17.39)$$

where I is the original image and I^{\triangleright} denotes the intermediate image, as defined in Eqn. (17.35).

Alternatively, the vertical filter could be applied first, followed by the horizontal filter. Algorithm 17.6 shows a direct implementation of the separable bilateral filter for grayscale images, using Gaussian kernels for both the domain and the range parts of the filter. Again, the extension to color images is straightforward (see Eqn. (17.31) and Exercise 17.3).

As intended, the advantage of the separable filter is performance. For a given kernel radius D , the original (non-separable) requires $\mathcal{O}(D^2)$ calculations for each pixel, while the separable version takes only $\mathcal{O}(D)$ steps. This means a substantial saving and speed increase, particularly for large filters.

[Figure 17.12](#) shows the response of the 1D separable bilateral filter in various situations. The results produced by the separable filter are very similar to those obtained with the original filter in [Figs. 17.5–17.9](#), partly because the local structures in these images are parallel to the coordinate axes. In general, the results are different, as demonstrated for a diagonal step edge in [Fig. 17.13](#). The effective filter kernels are shown in [Fig. 17.13\(g, h\)](#) for an anchor point positioned on the bright side of the edge. It can be seen that, while the kernel of the full filter [Fig. 17.13\(g\)](#) is orientation-insensitive, the upper part of the separable kernel is clearly truncated in [Fig. 17.13\(h\)](#). But although the separable bilateral filter is sensitive to local structure orientation, it performs well and is usually a sufficient substitute for the non-separable version [185]. The color examples shown in [Fig. 17.14](#) demonstrate the effects of 1D bilateral filtering in the x - and y -directions. Note that the results are not exactly the same if the filter is first applied in the x - or in y -direction, but usually the differences are negligible.



17.2 BILATERAL FILTER

Fig. 17.12
Response of a *separable* bilateral filter in various situations. Effective kernel $\tilde{H}_{I,u,v}$ (Eqn. (17.39)) at the center pixel (a–e), original image data (f–j), filtered image data (k–o). Settings are the same as in Figs. 17.5–17.9.

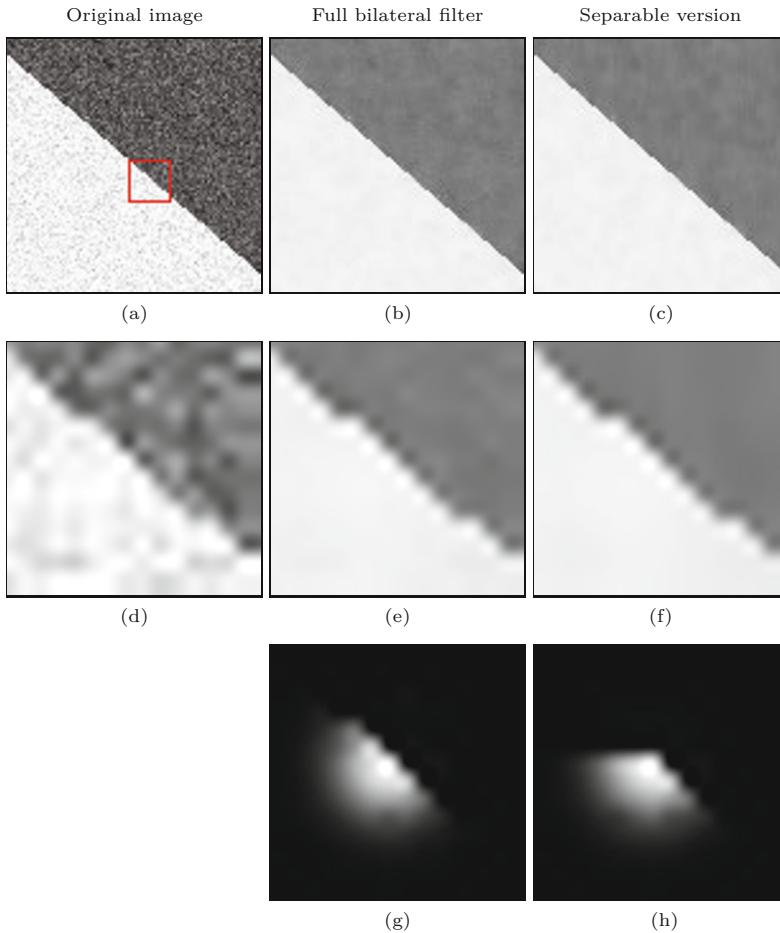


Fig. 17.13
Bilateral filter—full vs. separable version. Original image (a) and enlarged detail (d). Results of the full bilateral filter (b, e) and the separable version (c, f). The corresponding local filter kernels for the center pixel (positioned on the bright side of the step edge) for the full filter (g) and the separable version (h). Note how the upper part of the kernel in (h) is truncated along the horizontal axis, which shows that the separable filter is orientation-sensitive. In both cases, $\sigma_d = 2.0$, $\sigma_r = 25$.

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.6

Separable bilateral filter with Gaussian kernels (adapted from Alg. 17.4). The input image is processed in two passes. In each pass, a 1D kernel is applied in horizontal or vertical direction, respectively (see Eqns. (17.35)–(17.36)). Note that results of the separable filter are similar (but not identical) to the full (2D) bilateral filter in Alg. 17.4.

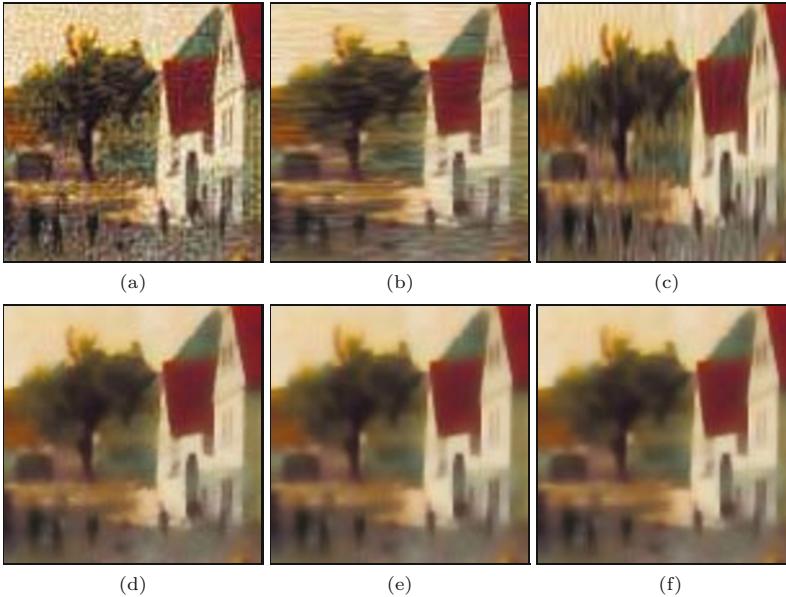
```

1: BilateralFilterGraySeparable( $I, \sigma_d, \sigma_r$ )
   Input:  $I$ , a grayscale image of size  $M \times N$ ;  $\sigma_d$ , width of the 2D Gaussian domain kernel;  $\sigma_r$ , width of the 1D Gaussian range kernel. Returns a new filtered image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $D \leftarrow \lceil 3.5 \cdot \sigma_d \rceil$                                  $\triangleright$  width of domain filter kernel
4:  $I^P \leftarrow \text{Duplicate}(I)$ 
   Pass 1 (horizontal):
5: for all coordinates  $(u, v) \in M \times N$  do
6:    $a \leftarrow I(u, v)$ 
7:    $S \leftarrow 0, W \leftarrow 0$ 
8:   for  $m \leftarrow -D, \dots, D$  do
9:      $b \leftarrow I(u + m, v)$ 
10:     $w_d \leftarrow \exp\left(-\frac{m^2}{2\sigma_d^2}\right)$            $\triangleright$  domain kernel coeff.  $h_d(m)$ 
11:     $w_r \leftarrow \exp\left(-\frac{(a-b)^2}{2\sigma_r^2}\right)$        $\triangleright$  range kernel coeff.  $h_r(b)$ 
12:     $w \leftarrow w_d \cdot w_r$                                 $\triangleright$  composite filter coeff.
13:     $S \leftarrow S + w \cdot b$ 
14:     $W \leftarrow W + w$ 
15:     $I^P(u, v) \leftarrow S/W$                              $\triangleright$  see Eq. 17.35
16:    $I' \leftarrow \text{Duplicate}(I)$ 
   Pass 2 (vertical):
17:   for all coordinates  $(u, v) \in M \times N$  do
18:      $a \leftarrow I^P(u, v)$ 
19:      $S \leftarrow 0, W \leftarrow 0$ 
20:     for  $n \leftarrow -D, \dots, D$  do
21:        $b \leftarrow I^P(u, v + n)$ 
22:        $w_d \leftarrow \exp\left(-\frac{n^2}{2\sigma_d^2}\right)$            $\triangleright$  domain kernel coeff.  $H_d(n)$ 
23:        $w_r \leftarrow \exp\left(-\frac{(a-b)^2}{2\sigma_r^2}\right)$        $\triangleright$  range kernel coeff.  $H_r(b)$ 
24:        $w \leftarrow w_d \cdot w_r$                                 $\triangleright$  composite filter coeff.
25:        $S \leftarrow S + w \cdot b$ 
26:        $W \leftarrow W + w$ 
27:      $I'(u, v) \leftarrow S/W$                              $\triangleright$  see Eq. 17.36
28:   return  $I'$ 

```

17.2.7 Further Reading

A thorough analysis of the bilateral filter as well as its relationship to adaptive smoothing and nonlinear diffusion can be found in [16] and [67]. In addition to the simple separable implementation described, several other fast versions of the bilateral filter have been proposed. For example, the method described in [65] approximates the bilateral filter by filtering sub-sampled copies of the image with discrete intensity kernels and recombining the results using linear interpolation. An improved and theoretically well-grounded version of this method was presented in [179]. The fast technique proposed in [253] eliminates the redundant calculations performed in partly overlapping image regions, albeit being restricted to the use of box-shaped domain kernels. As demonstrated in [187, 259], real-time performance using arbitrary-shaped kernels can be obtained by decomposing the filter into a set of smaller spatial filters.



17.3 ANISOTROPIC DIFFUSION FILTERS

Fig. 17.14

Separable bilateral filter (color example). Original image (a), bilateral filter applied only in the x -direction (b) and only in the y -direction (c). Result of applying the *full* bilateral filter (d) and the *separable* bilateral filter applied in x/y order (e) and y/x order (f). Settings: $\sigma_d = 2.0$, $\sigma_r = 50$, L_2 color distance.

17.3 Anisotropic Diffusion Filters

Diffusion is a concept adopted from physics that models the spatial propagation of particles or state properties within substances. In the real world, certain physical properties (such as temperature) tend to diffuse homogeneously through a physical body, that is, equally in all directions. The idea viewing image smoothing as a diffusion process has a long history in image processing (see, e.g., [11, 139]). To smooth an image and, at the same time, preserve edges or other “interesting” image structures, the diffusion process must somehow be made locally *non-homogeneous*; otherwise the entire image would come out equally blurred. Typically, the dominant smoothing direction is chosen to be *parallel* to nearby image contours, while smoothing is inhibited in the perpendicular direction, that is, *across* the contours.

Since the pioneering work by Perona and Malik [182], anisotropic diffusion has seen continued interest in the image processing community and research in this area is still strong today. The main elements of their approach are outlined in Sec. 17.3.2. While various other formulations have been proposed since, a key contribution by Weickert [250, 251] and Tschumperlé [233, 236] unified them into a common framework and demonstrated their extension to color images. They also proposed to separate the actual smoothing process from the smoothing geometry in order to obtain better control of the local smoothing behavior. In Sec. 17.3.4 we give a brief introduction to the approach proposed by Tschumperlé and Deriche, as initially described in [233]. Beyond these selected examples, a vast literature exists on this topic, including excellent reviews [95, 250], textbook material [125, 205], and journal articles (see [3, 45, 52, 173, 206, 226], for example).

17.3.1 Homogeneous Diffusion and the Heat Equation

Assume that in a homogeneous, 3D volume some physical property (e.g., temperature) is specified by a continuous function $f(\mathbf{x}, t)$ at position $\mathbf{x} = (x, y, z)$ and time t . With the system left to itself, the local differences in the property f will equalize over time until a global equilibrium is reached. This *diffusion process* in 3D space (x, y, z) and time (t) can be expressed using a partial differential equation (PDE),

$$\frac{\partial f}{\partial t} = c \cdot (\nabla^2 f) = c \cdot \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \right). \quad (17.40)$$

This is the so-called *heat equation*, where $\nabla^2 f$ denotes the *Laplace operator*⁹ applied to the scalar-valued function f , and c is a constant which describes the (thermal) *conductivity* or *conductivity coefficient* of the material. Since the conductivity is independent of position and orientation (c is constant), the resulting process is *isotropic*, that is, the heat spreads evenly in all directions.

For simplicity, we assume $c = 1$. Since f is a multi-dimensional function in space and time, we make this fact a bit more transparent by attaching explicit space and time coordinates \mathbf{x} and τ to Eqn. (17.40), that is,

$$\frac{\partial f}{\partial t}(\mathbf{x}, \tau) = \frac{\partial^2 f}{\partial x^2}(\mathbf{x}, \tau) + \frac{\partial^2 f}{\partial y^2}(\mathbf{x}, \tau) + \frac{\partial^2 f}{\partial z^2}(\mathbf{x}, \tau), \quad (17.41)$$

or, written more compactly,

$$f_t(\mathbf{x}, \tau) = f_{xx}(\mathbf{x}, \tau) + f_{yy}(\mathbf{x}, \tau) + f_{zz}(\mathbf{x}, \tau). \quad (17.42)$$

Diffusion in images

A continuous, time-varying image I may be treated analogously to the function $f(\mathbf{x}, \tau)$, with the local intensities taking on the role of the temperature values in Eqn. (17.42). In this 2D case, the isotropic diffusion equation can be written as¹⁰

$$\frac{\partial I}{\partial t} = \nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \quad \text{or} \quad (17.43)$$

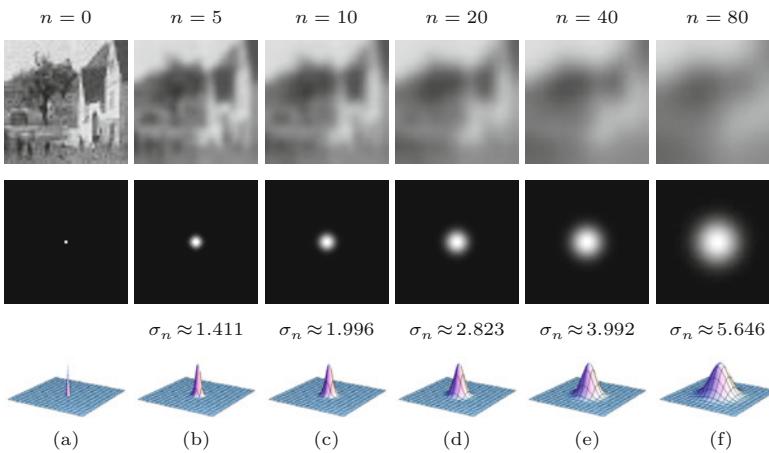
$$I_t(\mathbf{x}, \tau) = I_{xx}(\mathbf{x}, \tau) + I_{yy}(\mathbf{x}, \tau), \quad (17.44)$$

with the derivatives $I_t = \partial I / \partial t$, $I_{xx} = \partial^2 I / \partial x^2$, and $I_{yy} = \partial^2 I / \partial y^2$. An approximate, numerical solution of such a PDE can be obtained by replacing the derivatives with finite differences.

Starting with the initial (typically noisy) image $I^{(0)} = I$, the solution to the differential equation in Eqn. (17.44) can be calculated iteratively in the form

⁹ Remember that ∇f denotes the *gradient* of the function f , which is a vector for any multi-dimensional function. The Laplace operator (or *Laplacian*) $\nabla^2 f$ corresponds to the *divergence* of the *gradient* of f , denoted $\operatorname{div} \nabla f$, which is a scalar value (see Secs. C.2.5 and C.2.4 in the Appendix). Other notations for the Laplacian are $\nabla \cdot (\nabla f)$, $(\nabla \cdot \nabla) f$, $\nabla \cdot \nabla f$, $\nabla^2 f$, or Δf .

¹⁰ Function arguments (ξ, τ) are omitted here for better readability.



17.3 ANISOTROPIC DIFFUSION FILTERS

Fig. 17.15
Discrete isotropic diffusion. Blurred images and impulse response obtained after n iterations, with $\alpha = 0.20$ (see Eqn. (17.45)). The size of the images is 50×50 . The width of the equivalent Gaussian kernel (σ_n) grows with the square root of n (the number of iterations). Impulse response plots are normalized to identical peak values.

$$I^{(n)}(\mathbf{u}) \leftarrow \begin{cases} I(\mathbf{u}) & \text{for } n = 0, \\ I^{(n-1)}(\mathbf{u}) + \alpha \cdot [\nabla^2 I^{(n-1)}(\mathbf{u})] & \text{for } n > 0, \end{cases} \quad (17.45)$$

for each image position $\mathbf{u} = (u, v)$, with n denoting the iteration number. This is called the “direct” solution method (there are other methods but this is the simplest). The constant α in Eqn. (17.45) is the time increment, which controls the speed of the diffusion process. Its value should be in the range $(0, 0.25]$ for the numerical scheme to be stable. At each iteration n , the variations in the image function are reduced and (depending on the boundary conditions) the image function should eventually flatten out to a constant plane as n approaches infinity.

For a discrete image I , the Laplacian $\nabla^2 I$ in Eqn. (17.45) can be approximated by a linear 2D filter,

$$\nabla^2 I \approx I * H^L, \quad \text{with } H^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad (17.46)$$

as described earlier.¹¹ An essential property of isotropic diffusion is that it has the same effect as a Gaussian filter whose width grows with the elapsed time. For a discrete 2D image, in particular, the result obtained after n diffusion steps (Eqn. (17.45)), is the same as applying a linear filter to the original image I ,

$$I^{(n)} \equiv I * H^{G, \sigma_n}, \quad (17.47)$$

with the normalized Gaussian kernel

$$H^{G, \sigma_n}(x, y) = \frac{1}{2\pi\sigma_n^2} \cdot e^{-\frac{x^2+y^2}{2\sigma_n^2}} \quad (17.48)$$

of width $\sigma_n = \sqrt{2t} = \sqrt{2n/\alpha}$. The example in Fig. 17.15 illustrates this Gaussian smoothing behavior obtained with discrete isotropic diffusion.

¹¹ See also Chapter 6, Sec. 6.6.1 and Sec. C.3.1 in the Appendix.

17.3.2 Perona-Malik Filter

Isotropic diffusion, as we have described, is a homogeneous operation that is independent of the underlying image content. Like any Gaussian filter, it effectively suppresses image noise but also tends to blur away sharp boundaries and detailed structures, a property that is often undesirable. The idea proposed in [182] is to make the conductivity coefficient *variable* and dependent on the local image structure. This is done by replacing the conductivity constant c in Eqn. (17.40), which can be written as

$$\frac{\partial I}{\partial t}(\mathbf{x}, \tau) = c \cdot [\nabla^2 I](\mathbf{x}, \tau), \quad (17.49)$$

by a *function* $c(\mathbf{x}, t)$ that *varies* over space \mathbf{x} and time t , that is,

$$\frac{\partial I}{\partial t}(\mathbf{x}, \tau) = c(\mathbf{x}, \tau) \cdot [\nabla^2 I](\mathbf{x}, \tau). \quad (17.50)$$

If the conductivity function $c()$ is constant, then the equation reduces to the isotropic diffusion model in Eqn. (17.44).

Different behaviors can be implemented by selecting a particular function $c()$. To achieve edge-preserving smoothing, the conductivity $c()$ is chosen as a function of the magnitude of the local gradient vector ∇I , that is,

$$c(\mathbf{x}, \tau) := g(d) = g(\|[\nabla I^{(\tau)}](\mathbf{x})\|). \quad (17.51)$$

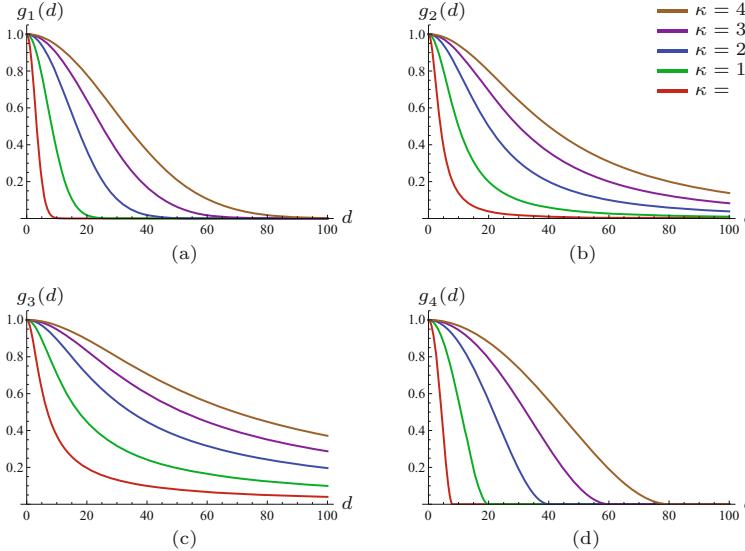
To preserve edges, the function $g(d) : \mathbb{R} \rightarrow [0, 1]$ should return high values in areas of low image gradient, enabling smoothing of homogeneous regions, but return low values (and thus inhibit smoothing) where the local brightness changes rapidly. Commonly used conductivity functions $g(d)$ are, for example [48, 182],

$$\begin{aligned} g_1(d) &= e^{-(d/\kappa)^2}, & g_2(d) &= \frac{1}{1+(d/\kappa)^2}, \\ g_3(d) &= \frac{1}{\sqrt{1+(d/\kappa)^2}}, & g_4(d) &= \begin{cases} (1-(d/2\kappa)^2)^2 & \text{for } d \leq 2\kappa, \\ 0 & \text{otherwise,} \end{cases} \end{aligned} \quad (17.52)$$

where $\kappa > 0$ is a constant that is either set manually (typically in the range [5, 50] for 8-bit images) or adjusted to the amount of image noise. Graphs of the four functions in Eqn. (17.52) are shown in Fig. 17.16 for selected values of κ . The Gaussian conductivity function g_1 tends to promote high-contrast edges, whereas g_2 and even more g_3 prefer wide, flat regions over smaller ones. Function g_4 , which corresponds to Tukey's *biweight* function known from robust statistics [205, p. 230], is strictly zero for any argument $d > 2\kappa$. The exact shape of the function $g()$ does not appear to be critical; other functions with similar properties (e.g., with a linear cutoff) are sometimes used instead.

As an approximate discretization of Eqn. (17.50), Perona and Malik [182] proposed the simple iterative scheme

$$I^{(n)}(\mathbf{u}) \leftarrow I^{(n-1)}(\mathbf{u}) + \alpha \cdot \sum_{i=0}^3 g(|\delta_i(I^{(n-1)}, \mathbf{u})|) \cdot \delta_i(I^{(n-1)}, \mathbf{u}), \quad (17.53)$$



17.3 ANISOTROPIC DIFFUSION FILTERS

Fig. 17.16

Typical conductivity functions $g_1(\cdot), \dots, g_4(\cdot)$ for $\kappa = 4, 10, 20, 30, 40$ (see Eqn. (17.52)). If the magnitude of the local gradient d is small (near zero), smoothing amounts to a maximum (1.0), whereas diffusion is reduced where the gradient is high, for example, at or near edges. Smaller values of κ result in narrower curves, thereby restricting the smoothing operation to image areas with only small variations.

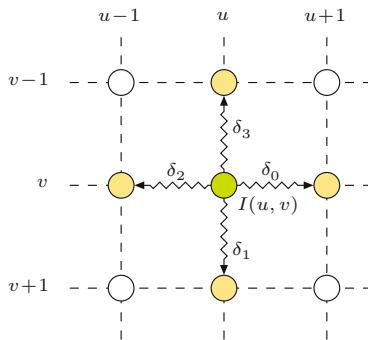


Fig. 17.17

Discrete lattice used for implementing diffusion filters in the Perona-Malik algorithm. The green element represents the current image pixel at position $\mathbf{u} = (u, v)$ and the yellow elements are its direct 4-neighbors.

where $I^{(0)} = I$ is the original image and

$$\delta_i(I, \mathbf{u}) = I(\mathbf{u} + \mathbf{d}_i) - I(\mathbf{u}) \quad (17.54)$$

denotes the difference between the pixel value $I(\mathbf{u})$ and its direct neighbor $i = 0, \dots, 3$ (see Fig. 17.17), with

$$\mathbf{d}_0 = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \right), \quad \mathbf{d}_1 = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix} \right), \quad \mathbf{d}_2 = -\left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \right), \quad \mathbf{d}_3 = -\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix} \right). \quad (17.55)$$

The procedure for computing the Perona-Malik filter for scalar-valued images is summarized in Alg. 17.7. The examples in Fig. 17.18 demonstrate how this filter performs along a step edge in a noisy grayscale image compared to isotropic (i.e., Gaussian) filtering.

In summary, the principle operation of this filter is to inhibit smoothing in the direction of strong local gradient vectors. Wherever the local contrast (and thus the gradient) is small, diffusion occurs uniformly in all directions, effectively implementing a Gaussian smoothing filter. However, in locations of high gradients, smoothing is inhibited along the gradient direction and allowed only in the direction perpendicular to it. If viewed as a heat diffusion process, a high-gradient brightness edge in an image acts like an insulating layer between areas of different temperatures. While temperatures

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.7

Perona-Malik anisotropic diffusion filter for scalar (grayscale) images. The input image I is assumed to be real-valued (floating-point). Temporary real-valued maps D_x, D_y are used to hold the directional gradient values, which are then re-calculated in every iteration.

The conductivity function $g(d)$ can be one of the functions defined in Eqn. (17.52), or any similar function.

```

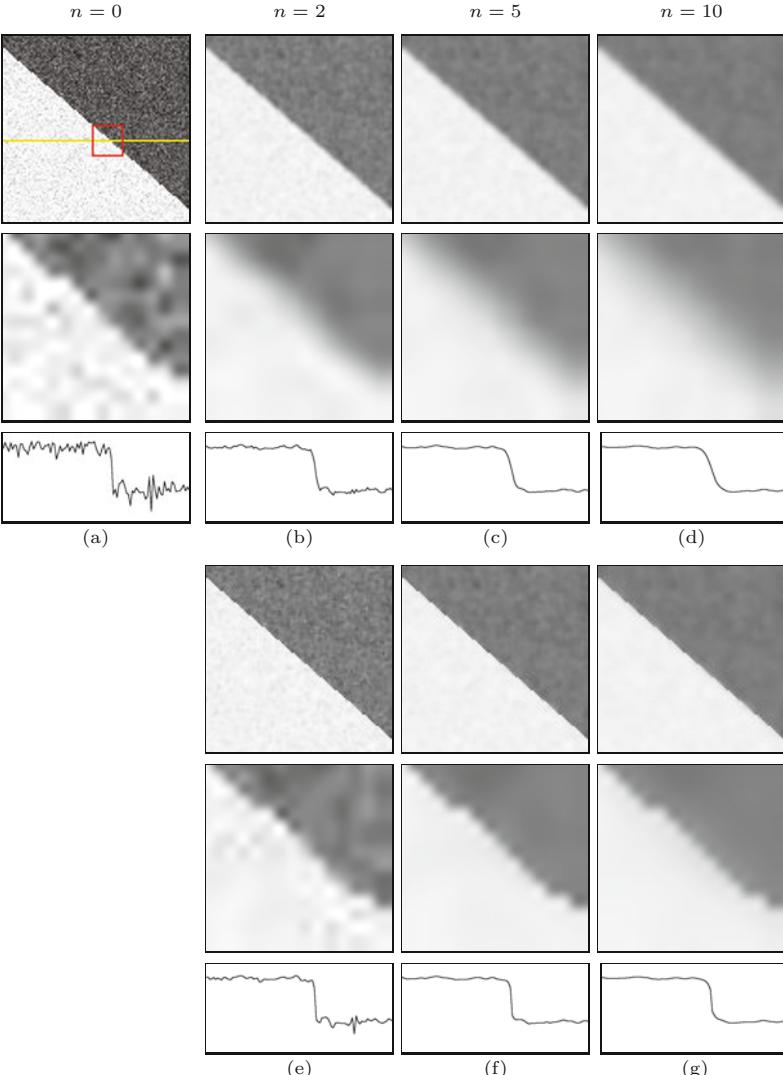
1: PeronaMalikGray( $I, \alpha, \kappa, T$ )
   Input:  $I$ , a grayscale image of size  $M \times N$ ;  $\alpha$ , update rate;  $\kappa$ , smoothness parameter;  $T$ , number of iterations. Returns the modified image  $I$ .
   Specify the conductivity function:
2:  $g(d) := e^{-(d/\kappa)^2}$        $\triangleright$  for example, see alternatives in Eq. 17.52
3:  $(M, N) \leftarrow \text{Size}(I)$ 
4: Create maps  $D_x, D_y: M \times N \rightarrow \mathbb{R}$ 
5: for  $n \leftarrow 1, \dots, T$  do           $\triangleright$  perform  $T$  iterations
6:   for all coordinates  $(u, v) \in M \times N$  do     $\triangleright$  re-calculate gradients
7:      $D_x(u, v) \leftarrow \begin{cases} I(u+1, v) - I(u, v) & \text{if } u < M-1 \\ 0 & \text{otherwise} \end{cases}$ 
8:      $D_y(u, v) \leftarrow \begin{cases} I(u, v+1) - I(u, v) & \text{if } v < N-1 \\ 0 & \text{otherwise} \end{cases}$ 
9:   for all coordinates  $(u, v) \in M \times N$  do     $\triangleright$  update the image
10:     $\delta_0 \leftarrow D_x(u, v)$ 
11:     $\delta_1 \leftarrow D_y(u, v)$ 
12:     $\delta_2 \leftarrow \begin{cases} -D_x(u-1, v) & \text{if } u > 0 \\ 0 & \text{otherwise} \end{cases}$ 
13:     $\delta_3 \leftarrow \begin{cases} -D_y(u, v-1) & \text{if } v > 0 \\ 0 & \text{otherwise} \end{cases}$ 
14:     $I(u, v) \leftarrow I(u, v) + \alpha \cdot \sum_{k=0}^3 g(|\delta_k|) \cdot \delta_k$ 
15: return  $I$ 
```

continuously level out in the homogeneous regions on either side of an edge, thermal energy does not diffuse across the edge itself.

Note that the Perona-Malik filter (as defined in Eqn. (17.50)) is formally considered a *nonlinear* filter but not an *anisotropic* diffusion filter because the conductivity function $g()$ is only a scalar and not a (directed) vector-valued function [250]. However, the (inexact) discretization used in Eqn. (17.53), where each lattice direction is attenuated individually, makes the filter appear to perform in an anisotropic fashion.

17.3.3 Perona-Malik Filter for Color Images

The original Perona-Malik filter is not explicitly designed for color images or vector-valued images in general. The simplest way to apply this filter to a color image is (as usual) to treat the color channels as a set of independent scalar images and filter them separately. Edges should be preserved, since they occur only where at least one of the color channels exhibits a strong variation. However, different filters are applied to the color channels and thus new chromaticities may be produced that were not contained in the original image. Nevertheless, the results obtained (see the examples in Fig. 17.19(b-d)) are often satisfactory and the approach is frequently used because of its simplicity.



17.3 ANISOTROPIC DIFFUSION FILTERS

Fig. 17.18
Isotropic vs. anisotropic diffusion applied to a noisy step edge. Original image, enlarged detail, and horizontal profile (a), results of isotropic diffusion (b–d), results of anisotropic diffusion (e–g) after $n = 2, 5, 10$ iterations, respectively ($\alpha = 0.20$, $\kappa = 40$).

Color diffusion based on the brightness gradient

As an alternative to filtering each color channel separately, it has been proposed to use only the brightness (intensity) component to control the diffusion process of all color channels. Given an RGB color image $\mathbf{I} = (I_R, I_G, I_B)$ and a brightness function $\beta(\mathbf{I})$, the iterative scheme in Eqn. (17.53) could be modified to

$$\mathbf{I}^{(n)}(\mathbf{u}) \leftarrow \mathbf{I}^{(n-1)}(\mathbf{u}) + \alpha \cdot \sum_{i=0}^3 g(|\beta_i(\mathbf{I}^{(n-1)}, \mathbf{u})|) \cdot \delta_i(\mathbf{I}^{(n-1)}, \mathbf{u}), \quad (17.56)$$

$$\text{where } \beta_i(\mathbf{I}, \mathbf{u}) = \beta(\mathbf{I}(\mathbf{u} + \mathbf{d}_i)) - \beta(\mathbf{I}(\mathbf{u})), \quad (17.57)$$

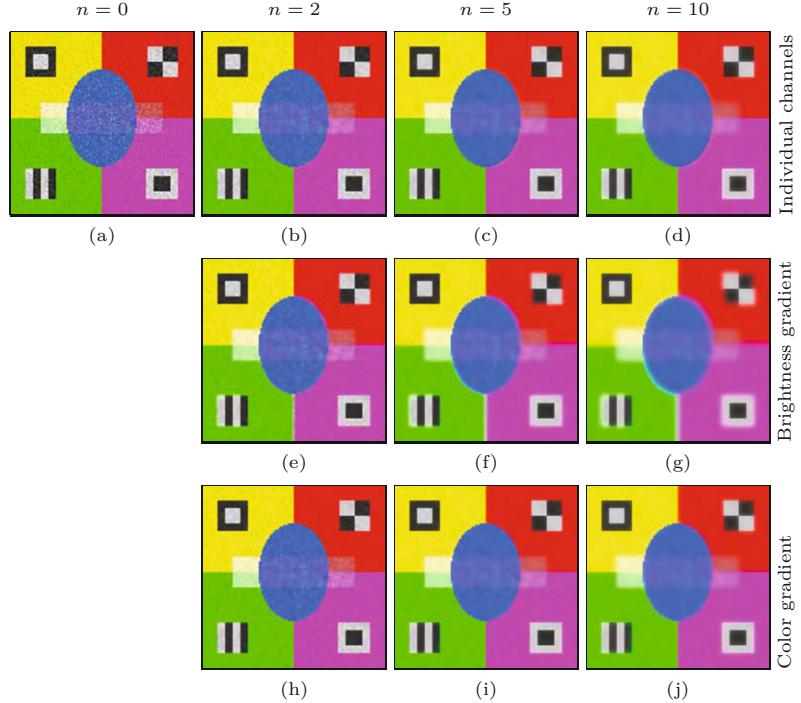
\mathbf{d}_i is the local brightness difference (as defined in Eqn. (17.55)) and

$$\delta_i(\mathbf{I}, \mathbf{u}) = \begin{pmatrix} I_R(\mathbf{u} + \mathbf{d}_i) - I_R(\mathbf{u}) \\ I_G(\mathbf{u} + \mathbf{d}_i) - I_G(\mathbf{u}) \\ I_B(\mathbf{u} + \mathbf{d}_i) - I_B(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \delta_i(I_R, \mathbf{u}) \\ \delta_i(I_G, \mathbf{u}) \\ \delta_i(I_B, \mathbf{u}) \end{pmatrix} \quad (17.58)$$

Fig. 17.19

Anisotropic diffusion filter (color). Noisy test image (a). Anisotropic diffusion filter applied separately to *individual color channels* (b–d), diffusion controlled by *brightness gradient* (e–g), diffusion controlled by *color gradient* (h–j), after 2, 5, and 10 iterations, respectively ($\alpha = 0.20$, $\kappa = 40$).

With diffusion controlled by the brightness gradient, strong blurring occurs between regions of different color but similar brightness (e–g). The most consistent results are obtained by diffusion controlled by the *color gradient* (h–j). Filtering was performed in linear RGB color space.



is the local color difference vector for the neighboring pixels in directions $i = 0, \dots, 3$ (see Fig. 17.17). Typical choices for the brightness function $\beta()$ are the *luminance* Y (calculated as a weighted sum of the linear R, G, B components), *luma* Y' (from nonlinear R', G', B' components), or the lightness component L of the CIELAB and CIELUV color spaces (see Chapter 15, Sec. 15.1 for a detailed discussion).

Algorithm 17.7 can be easily adapted to implement this type of color filter. An obvious disadvantage of this method is that it naturally blurs across color edges if the neighboring colors are of similar brightness, as the examples in Fig. 17.19(e–g)) demonstrate. This limits its usefulness for practical applications.

Using the color gradient

A better option for controlling the diffusion process in all three color channels is to use the color gradient (see Ch. 16, Sec. 16.2.1). As defined in Eqn. (16.17), the color gradient

$$(\text{grad}_\theta \mathbf{I})(\mathbf{u}) = \mathbf{I}_x(\mathbf{u}) \cdot \cos(\theta) + \mathbf{I}_y(\mathbf{u}) \cdot \sin(\theta) \quad (17.59)$$

is a 3D vector, representing the combined variations of the color image \mathbf{I} at position \mathbf{u} in a given direction θ . The squared norm of this vector, $S_\theta(\mathbf{I}, \mathbf{u}) = \|(\text{grad}_\theta \mathbf{I})(\mathbf{u})\|^2$, called the *squared local contrast*, is a scalar quantity useful for color edge detection. Along the horizontal and vertical directions of the discrete diffusion lattice (see Fig. 17.17), the angle θ is a multiple of $\pi/2$, and thus one of the cosine/sine terms in Eqn. (17.59) vanishes, that is,

$$\begin{aligned}\|(\text{grad}_\theta \mathbf{I})(\mathbf{u})\| &= \|(\text{grad}_{i\pi/2} \mathbf{I})(\mathbf{u})\| \\ &= \begin{cases} \|\mathbf{I}_x(\mathbf{u})\| & \text{for } i = 0, 2, \\ \|\mathbf{I}_y(\mathbf{u})\| & \text{for } i = 1, 3. \end{cases} \quad (17.60)\end{aligned}$$

Taking δ_i (Eqn. (17.58)) as an estimate for the horizontal and vertical derivatives $\mathbf{I}_x, \mathbf{I}_y$, the diffusion iteration (adapted from Eqn. (17.53)) thus becomes

$$\mathbf{I}^{(n)}(\mathbf{u}) \leftarrow \mathbf{I}^{(n-1)}(\mathbf{u}) + \alpha \cdot \sum_{i=0}^3 g(\|\delta_i(\mathbf{I}^{(n-1)}, \mathbf{u})\|) \cdot \delta_i(\mathbf{I}^{(n-1)}, \mathbf{u}), \quad (17.61)$$

with $g()$ chosen from one of the conductivity functions in Eqn. (17.52). Note that this is almost identical to the formulation in Eqn. (17.53), except for the use of vector-valued images and the absolute values $|\cdot|$ being replaced by the vector norm $\|\cdot\|$. The diffusion process is coupled between color channels, because the local diffusion strength depends on the combined color difference vectors. Thus, unlike in the brightness-governed diffusion scheme in Eqn. (17.56), opposing variations in different color do not cancel out and edges between colors of similar brightness are preserved (see the examples in Fig. 17.19(h–j)).

The resulting process is summarized in Alg. 17.8. The algorithm assumes that the components of the color image \mathbf{I} are real-valued. In practice, integer-valued images must be converted to floating point before this procedure can be applied and integer results should be recovered by appropriate rounding.

Examples

Figure 17.20 shows the results of applying the Perona-Malik filter to a color image, using different modalities to control the diffusion process. In Fig. 17.20(a) the *scalar* (grayscale) diffusion filter (described in Alg. 17.7) is applied *separately* to each color channel. In Fig. 17.20(b) the diffusion process is coupled over all three color channels and controlled by the *brightness gradient*, as specified in Eqn. (17.56). Finally, in Fig. 17.20(c) the *color gradient* is used to control the common diffusion process, as defined in Eqn. (17.61) and Alg. 17.8. In each case, $T = 10$ diffusion iterations were applied, with update rate $\alpha = 0.20$, smoothness $\kappa = 25$, and conductivity function $g_1(d)$. The example demonstrates that, under otherwise equal conditions, edges and line structures are best preserved by the filter if the diffusion process is controlled by the color gradient.

17.3.4 Geometry Preserving Anisotropic Diffusion

Historically, the seminal publication by Perona and Malik [182] was followed by increased interest in the use of diffusion filters based on partial differential equations. Numerous different schemes were proposed, mainly with the aim to better adapt the diffusion process to the underlying image geometry.

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.8

Anisotropic diffusion filter for color images based on the color gradient (see Ch. 16, Sec. 16.2.1). The conductivity function $g(d)$ may be chosen from the functions defined in Eqn. (17.52), or any similar function. Note that (unlike in Alg. 17.7) the maps D_x, D_y are vector-valued.

```

1: PeronaMalikColor( $I, \alpha, \kappa, T$ )
   Input:  $I$ , an RGB color image of size  $M \times N$ ;  $\alpha$ , update rate;
           $\kappa$ , smoothness parameter;  $T$ , number of iterations. Returns the
          modified image  $I$ .
   Specify the conductivity function:
2:  $g(d) := e^{-(d/\kappa)^2}$             $\triangleright$  for example, see alternatives in Eq. 17.52
3:  $(M, N) \leftarrow \text{Size}(I)$ 
4: Create maps  $D_x, D_y: M \times N \rightarrow \mathbb{R}^3$ ;  $S_x, S_y: M \times N \rightarrow \mathbb{R}$ 
5: for  $n \leftarrow 1, \dots, T$  do            $\triangleright$  perform  $T$  iterations
   for all  $(u, v) \in M \times N$  do        $\triangleright$  re-calculate gradients
7:    $D_x(u, v) \leftarrow \begin{cases} I(u+1, v) - I(u, v) & \text{if } u < M-1 \\ \mathbf{0} & \text{otherwise} \end{cases}$ 
8:    $D_y(u, v) \leftarrow \begin{cases} I(u, v+1) - I(u, v) & \text{if } v < N-1 \\ \mathbf{0} & \text{otherwise} \end{cases}$ 
9:    $S_x(u, v) \leftarrow (D_x(u, v))^2$             $\triangleright = I_{R,x}^2 + I_{G,x}^2 + I_{B,x}^2$ 
10:   $S_y(u, v) \leftarrow (D_y(u, v))^2$             $\triangleright = I_{R,y}^2 + I_{G,y}^2 + I_{B,y}^2$ 
11:  for all  $(u, v) \in M \times N$  do        $\triangleright$  update the image
12:     $s_0 \leftarrow S_x(u, v), \Delta_0 \leftarrow D_x(u, v)$ 
13:     $s_1 \leftarrow S_y(u, v), \Delta_1 \leftarrow D_y(u, v)$ 
14:     $s_2 \leftarrow 0, \Delta_2 \leftarrow \mathbf{0}$ 
15:     $s_3 \leftarrow 0, \Delta_3 \leftarrow \mathbf{0}$ 
16:    if  $u > 0$  then
17:       $s_2 \leftarrow S_x(u-1, v)$ 
18:       $\Delta_2 \leftarrow -D_x(u-1, v)$ 
19:    if  $v > 0$  then
20:       $s_3 \leftarrow S_y(u, v-1)$ 
21:       $\Delta_3 \leftarrow -D_y(u, v-1)$ 
22:     $I(u, v) \leftarrow I(u, v) + \alpha \cdot \sum_{k=0}^3 g(|s_k|) \cdot \Delta_k$ 
23:  return  $I$ 

```

Generalized divergence-based formulation

Weickert [249, 250] generalized the divergence-based formulation of the Perona-Malik approach (see Eqn. (17.49)), that is,

$$\frac{\partial I}{\partial t} = \text{div}(c \cdot \nabla I),$$

by replacing the time-varying, scalar diffusivity field $c(\mathbf{x}, \tau) \in \mathbb{R}$ by a *diffusion tensor* field $\mathbf{D}(\mathbf{x}, \tau) \in \mathbb{R}^{2 \times 2}$ in the form

$$\frac{\partial I}{\partial t} = \text{div}(\mathbf{D} \cdot \nabla I). \quad (17.62)$$

The time-varying tensor field $\mathbf{D}(\mathbf{x}, \tau)$ specifies a symmetric, positive-definite 2×2 matrix for each 2D image position \mathbf{x} and time τ (i.e., $\mathbf{D} : \mathbb{R}^3 \rightarrow \mathbb{R}^{2 \times 2}$ in the continuous case). Geometrically, \mathbf{D} specifies an oriented, stretched ellipse which controls the local diffusion process. \mathbf{D} may be independent of the image I but is typically derived from it. For example, the original Perona-Malik diffusion equation could be (trivially) written in the form¹²

¹² \mathbf{I}_2 denotes the 2×2 identity matrix.



(a) Color channels filtered separately



(b) Diffusion controlled by the local brightness gradient



(c) Diffusion controlled by the local color gradient

17.3 ANISOTROPIC DIFFUSION FILTERS

Fig. 17.20
Perona-Malik color example. Scalar diffusion filter applied separately to each color channel (a); diffusion controlled by the brightness gradient (b); diffusion controlled by color gradient (c). Common settings are $T = 10$, $\alpha = 0.20$, $g(d) = g_1(d)$, $\kappa = 25$; original image in Fig. 17.3(a).

$$\frac{\partial I}{\partial t} = \operatorname{div} \left[\underbrace{(c \cdot \mathbf{I}_2) \cdot \nabla I}_{\mathbf{D}} \right] = \operatorname{div} \left[\begin{pmatrix} c & 0 \\ 0 & c \end{pmatrix} \cdot \nabla I \right], \quad (17.63)$$

where $c = g(\|\nabla I(\mathbf{x}, t)\|)$ (see Eqn. (17.51)), and thus \mathbf{D} is coupled to the image content. In Weickert's approach, \mathbf{D} is constructed from the eigenvalues of the local "image structure tensor" [251], which we have encountered under different names in several places. This approach was also adapted to work with color images [252].

Trace-based formulation

Similar to the work of Weickert, the approach proposed by Tschumperlé and Deriche [233, 235] also pursues a geometry-oriented generalization of anisotropic diffusion. The approach is directly aimed at vector-valued (color) images, but can also be applied to single-channel (scalar-valued) images. For a vector-valued image $\mathbf{I} = (I_1, \dots, I_n)$, the smoothing process is specified as

$$\frac{\partial I_k}{\partial t} = \operatorname{trace} (\mathbf{A} \cdot \mathbf{H}_k), \quad (17.64)$$

for each channel k , where \mathbf{H}_k denotes the *Hessian* matrix of the scalar-valued image function of channel I_k , and \mathbf{A} is a square (2×2 for 2D images) matrix that depends on the complete image \mathbf{I} and

adapts the smoothing process to the local image geometry. Note that \mathbf{A} is the same for all image channels. Since the trace of the Hessian matrix¹³ is the Laplacian of the corresponding function (i.e., $\text{trace}(\mathbf{H}_I) = \nabla^2 I$) the diffusion equation for the Perona-Malik filter (Eqn. (17.49)) can be written as

$$\begin{aligned}\frac{\partial I}{\partial t} &= c \cdot (\nabla^2 I) = \text{div}(c \cdot \nabla I) \\ &= \text{trace}((c \cdot \mathbf{I}_2) \cdot \mathbf{H}_I) = \text{trace}(c \cdot \mathbf{H}_I).\end{aligned}\quad (17.65)$$

In this case, $\mathbf{A} = c \cdot \mathbf{I}_2$, which merely applies the constant scalar factor c to the Hessian matrix \mathbf{H}_I (and thus to the resulting Laplacian) that is derived from the local image (since $c = g(\|\nabla I(\mathbf{x}, t)\|)$) and does not represent any geometric information.

17.3.5 Tschumperlé-Deriche Algorithm

This is different in the trace-based approach proposed by Tschumperlé and Deriche [233, 235], where the matrix \mathbf{A} in Eqn. (17.64) is composed by the expression

$$\mathbf{A} = f_1(\lambda_1, \lambda_2) \cdot (\hat{\mathbf{q}}_2 \cdot \hat{\mathbf{q}}_2^\top) + f_2(\lambda_1, \lambda_2) \cdot (\hat{\mathbf{q}}_1 \cdot \hat{\mathbf{q}}_1^\top), \quad (17.66)$$

where λ_1, λ_2 and $\hat{\mathbf{q}}_1, \hat{\mathbf{q}}_2$ are the eigenvalues and normalized eigenvectors, respectively, of the (smoothed) 2×2 structure matrix

$$\mathbf{G} = \sum_{k=1}^K (\nabla I_k) \cdot (\nabla I_k)^\top, \quad (17.67)$$

with ∇I_k denoting the local gradient vector in image channel I_k . The functions $f_1()$, $f_2()$, which are defined in Eqn. (17.79), use the two eigenvalues to control the diffusion strength along the dominant direction of the contours (f_1) and perpendicular to it (f_2). Since the resulting algorithm is more involved than most previous ones, we describe it in more detail than usual.

Given a vector-valued image $\mathbf{I}: M \times N \rightarrow \mathbb{R}^n$, the following steps are performed in each iteration of the algorithm:

Step 1:

Calculate the gradient at each image position $\mathbf{u} = (u, v)$,

$$\nabla I_k(\mathbf{u}) = \begin{pmatrix} \frac{\partial I_k}{\partial x}(\mathbf{u}) \\ \frac{\partial I_k}{\partial y}(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} I_{k,x}(\mathbf{u}) \\ I_{k,y}(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} (I_k * H_x^\nabla)(\mathbf{u}) \\ (I_k * H_y^\nabla)(\mathbf{u}) \end{pmatrix}, \quad (17.68)$$

for each color channel $k = 1, \dots, K$.¹⁴ The first derivatives of the gradient vector ∇I_k are estimated by convolving the image with the kernels

¹³ See Sec. C.2.6 in the Appendix for details.

¹⁴ Note that $\nabla I_k(\mathbf{u})$ in Eqn. (17.68) is a 2D, vector-valued function, that is, a dedicated vector is calculated for every image position $\mathbf{u} = (u, v)$. For better readability, we omit the spatial coordinate (\mathbf{u}) in the following and simply write ∇I_k instead of $\nabla I_k(\mathbf{u})$. Analogously, all related vectors and matrices defined below (including the vectors $\mathbf{e}_1, \mathbf{e}_2$ and the matrices $\mathbf{G}, \bar{\mathbf{G}}, \mathbf{A}$, and \mathbf{H}_k) are also calculated for each image point \mathbf{u} , without the spatial coordinate being explicitly given.

$$H_x^\nabla = \begin{bmatrix} -a & 0 & a \\ -b & 0 & b \\ -a & 0 & a \end{bmatrix} \quad \text{and} \quad H_y^\nabla = \begin{bmatrix} -a & -b & -a \\ 0 & 0 & 0 \\ a & b & a \end{bmatrix}, \quad (17.69)$$

with $a = (2 - \sqrt{2})/4$ and $b = (\sqrt{2} - 1)/2$ (such that $2a + b = 1/2$).¹⁵

Step 2:

Smooth the channel gradients $I_{k,x}, I_{k,y}$ with an isotropic 2D Gaussian filter kernel H^{G,σ_d} of radius σ_d ,

$$\overline{\nabla I}_k = \begin{pmatrix} \bar{I}_{k,x} \\ \bar{I}_{k,y} \end{pmatrix} = \begin{pmatrix} I_{k,x} * H^{G,\sigma_d} \\ I_{k,y} * H^{G,\sigma_d} \end{pmatrix}, \quad (17.70)$$

for each image channel $k = 1, \dots, K$. In practice, this step is usually skipped by setting $\sigma_d = 0$.

Step 3:

Calculate the *Hessian matrix* (see Sec. C.2.6 in the Appendix) for each image channel I_k , $k = 1, \dots, K$, that is,

$$\mathbf{H}_k = \begin{pmatrix} \frac{\partial^2 I_k}{\partial x^2} & \frac{\partial^2 I_k}{\partial x \partial y} \\ \frac{\partial^2 I_k}{\partial y \partial x} & \frac{\partial^2 I_k}{\partial y^2} \end{pmatrix} = \begin{pmatrix} I_{k,xx} & I_{k,xy} \\ I_{k,xy} & I_{k,yy} \end{pmatrix} = \begin{pmatrix} I_k * H_{xx}^\nabla & I_k * H_{xy}^\nabla \\ I_k * H_{xy}^\nabla & I_k * H_{yy}^\nabla \end{pmatrix}, \quad (17.71)$$

using the filter kernels

$$H_{xx}^\nabla = [1 \ -2 \ 1], \quad H_{yy}^\nabla = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}, \quad H_{xy}^\nabla = \frac{1}{4} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}. \quad (17.72)$$

Step 4:

Calculate the local variation (structure) matrix as

$$\begin{aligned} \mathbf{G} &= \begin{pmatrix} G_0 & G_1 \\ G_1 & G_2 \end{pmatrix} = \sum_{k=1}^K (\overline{\nabla I}_k) \cdot (\overline{\nabla I}_k)^\top \quad (17.73) \\ &= \sum_{k=1}^K \begin{pmatrix} \bar{I}_{k,x}^2 & \bar{I}_{k,x} \cdot \bar{I}_{k,y} \\ \bar{I}_{k,x} \cdot \bar{I}_{k,y} & \bar{I}_{k,y}^2 \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^K \bar{I}_{k,x}^2 & \sum_{k=1}^K \bar{I}_{k,x} \cdot \bar{I}_{k,y} \\ \sum_{k=1}^K \bar{I}_{k,x} \cdot \bar{I}_{k,y} & \sum_{k=1}^K \bar{I}_{k,y}^2 \end{pmatrix}, \end{aligned}$$

for each image position \mathbf{u} . Note that the matrix \mathbf{G} is symmetric (and positive semidefinite). In particular, for a RGB color image this is (coordinates \mathbf{u} again omitted)

$$\begin{aligned} \mathbf{G} &= \begin{pmatrix} \bar{I}_{R,x}^2 & \bar{I}_{R,x} \bar{I}_{R,y} \\ \bar{I}_{R,x} \bar{I}_{R,y} & \bar{I}_{R,y}^2 \end{pmatrix} + \begin{pmatrix} \bar{I}_{G,x}^2 & \bar{I}_{G,x} \bar{I}_{G,y} \\ \bar{I}_{G,x} \bar{I}_{G,y} & \bar{I}_{G,y}^2 \end{pmatrix} + \begin{pmatrix} \bar{I}_{B,x}^2 & \bar{I}_{B,x} \bar{I}_{B,y} \\ \bar{I}_{B,x} \bar{I}_{B,y} & \bar{I}_{B,y}^2 \end{pmatrix} \\ &= \begin{pmatrix} \bar{I}_{R,x}^2 + \bar{I}_{G,x}^2 + \bar{I}_{B,x}^2 & \bar{I}_{R,x} \bar{I}_{R,y} + \bar{I}_{G,x} \bar{I}_{G,y} + \bar{I}_{B,x} \bar{I}_{B,y} \\ \bar{I}_{R,x} \bar{I}_{R,y} + \bar{I}_{G,x} \bar{I}_{G,y} + \bar{I}_{B,x} \bar{I}_{B,y} & \bar{I}_{R,y}^2 + \bar{I}_{G,y}^2 + \bar{I}_{B,y}^2 \end{pmatrix}. \quad (17.74) \end{aligned}$$

¹⁵ Any other common set of x/y gradient kernels (e.g., Sobel masks) could be used instead, but these filters have better rotation invariance than their traditional counterparts. Similar kernels (with $a = 3/32$, $b = 10/32$) were proposed by Jähne in [126, p. 353].

Step 5:

Smooth the elements of the structure matrix \mathbf{G} using an isotropic Gaussian filter kernel H^{G,σ_g} of radius σ_g , that is,

$$\bar{\mathbf{G}} = \begin{pmatrix} \bar{G}_0 & \bar{G}_1 \\ \bar{G}_1 & \bar{G}_2 \end{pmatrix} = \begin{pmatrix} G_0 * H^{G,\sigma_g} & G_1 * H^{G,\sigma_g} \\ G_1 * H^{G,\sigma_g} & G_2 * H^{G,\sigma_g} \end{pmatrix}. \quad (17.75)$$

Step 6:

For each image position \mathbf{u} , calculate the eigenvalues λ_1, λ_2 for the smoothed 2×2 matrix $\bar{\mathbf{G}}$, such that $\lambda_1 \geq \lambda_2$, and the corresponding normalized eigenvectors¹⁶

$$\hat{\mathbf{q}}_1 = \begin{pmatrix} \hat{x}_1 \\ \hat{y}_1 \end{pmatrix}, \quad \hat{\mathbf{q}}_2 = \begin{pmatrix} \hat{x}_2 \\ \hat{y}_2 \end{pmatrix},$$

such that $\|\hat{\mathbf{q}}_1\| = \|\hat{\mathbf{q}}_2\| = 1$. Note that $\hat{\mathbf{q}}_1$ points in the direction of maximum change and $\hat{\mathbf{q}}_2$ points in the perpendicular direction, that is, along the edge tangent. Thus, smoothing should occur predominantly along $\hat{\mathbf{q}}_2$. Since $\hat{\mathbf{q}}_1$ and $\hat{\mathbf{q}}_2$ are normal to each other, we can express $\hat{\mathbf{q}}_2$ in terms of $\hat{\mathbf{q}}_1$, for example,

$$\hat{\mathbf{q}}_2 \equiv \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot \hat{\mathbf{q}}_1 = \begin{pmatrix} -\hat{y}_1 \\ \hat{x}_1 \end{pmatrix}. \quad (17.76)$$

Step 7:

From the eigenvalues (λ_1, λ_2) and the normalized eigenvectors $(\hat{\mathbf{q}}_1, \hat{\mathbf{q}}_2)$ of $\bar{\mathbf{G}}$, compose the symmetric matrix \mathbf{A} in the form

$$\begin{aligned} \mathbf{A} = \begin{pmatrix} A_0 & A_1 \\ A_1 & A_2 \end{pmatrix} &= \underbrace{f_1(\lambda_1, \lambda_2)}_{c_1} \cdot (\hat{\mathbf{q}}_2 \cdot \hat{\mathbf{q}}_2^\top) + \underbrace{f_2(\lambda_1, \lambda_2)}_{c_2} \cdot (\hat{\mathbf{q}}_1 \cdot \hat{\mathbf{q}}_1^\top) \\ &= c_1 \cdot \begin{pmatrix} \hat{y}_1^2 & -\hat{x}_1 \cdot \hat{y}_1 \\ -\hat{x}_1 \cdot \hat{y}_1 & \hat{x}_1^2 \end{pmatrix} + c_2 \cdot \begin{pmatrix} \hat{x}_1^2 & \hat{x}_1 \cdot \hat{y}_1 \\ \hat{x}_1 \cdot \hat{y}_1 & \hat{y}_1^2 \end{pmatrix} \end{aligned} \quad (17.77)$$

$$= \begin{pmatrix} c_1 \cdot \hat{y}_1^2 + c_2 \cdot \hat{x}_1^2 & (c_2 - c_1) \cdot \hat{x}_1 \cdot \hat{y}_1 \\ (c_2 - c_1) \cdot \hat{x}_1 \cdot \hat{y}_1 & c_1 \cdot \hat{x}_1^2 + c_2 \cdot \hat{y}_1^2 \end{pmatrix}, \quad (17.78)$$

using the conductivity coefficients

$$\begin{aligned} c_1 &= f_1(\lambda_1, \lambda_2) = \frac{1}{(1 + \lambda_1 + \lambda_2)^{a_1}}, \\ c_2 &= f_2(\lambda_1, \lambda_2) = \frac{1}{(1 + \lambda_1 + \lambda_2)^{a_2}}, \end{aligned} \quad (17.79)$$

with fixed parameters $a_1, a_2 > 0$ to control the non-isotropy of the filter: a_1 specifies the amount of smoothing along contours, a_2 in perpendicular direction (along the gradient). Small values of a_1, a_2 facilitate diffusion in the corresponding direction, while larger values inhibit smoothing. With a_1 close to zero, diffusion is practically unconstrained along the tangent direction. Typical default values are $a_1 = 0.5$ and $a_2 = 0.9$; results from other settings are shown in the examples.

¹⁶ See Sec. B.4.1 in the Appendix for details on calculating the eigensystem of a 2×2 matrix.

Step 8:

Finally, each image channel I_k is updated using the recurrence relation

$$I_k \leftarrow I_k + \alpha \cdot \text{trace}(\mathbf{A} \cdot \mathbf{H}_k) = I_k + \alpha \cdot \beta_k \quad (17.80)$$

$$= I_k + \alpha \cdot (A_0 \cdot I_{k,xx} + A_1 \cdot I_{k,xy} + A_1 \cdot I_{k,yx} + A_2 \cdot I_{k,yy}) \quad (17.81)$$

$$= I_k + \alpha \cdot \underbrace{(A_0 \cdot I_{k,xx} + 2 \cdot A_1 \cdot I_{k,xy} + A_2 \cdot I_{k,yy})}_{\beta_k} \quad (17.82)$$

(since $I_{k,xy} = I_{k,yx}$). The term $\beta_k = \text{trace}(\mathbf{A} \cdot \mathbf{H}_k)$ represents the local image *velocity* in channel k . Note that, although a separate Hessian matrix \mathbf{H}_k is calculated for each channel, the structure matrix \mathbf{A} is the same for all image channels. The image is thus smoothed along a common image geometry which considers the correlation between color channels, since \mathbf{A} is derived from the joint structure matrix \mathbf{G} (Eqn. (17.74)) and therefore combines all K color channels.

In each iteration, the factor α in Eqn. (17.82) is adjusted dynamically to the maximum current velocity β_k in all channels in the form

$$\alpha = \frac{d_t}{\max \beta_k} = \frac{d_t}{\max_{k,u} |\text{trace}(\mathbf{A} \cdot \mathbf{H}_k)|}, \quad (17.83)$$

where d_t is the (constant) “time increment” parameter. Thus the time step α is kept small as long as the image gradients (vector field velocities) are large. As smoothing proceeds, image gradients are reduced and thus α typically increases over time. In the actual implementation, the values of I_k (in Eqn. (17.82)) are hard-limited to the initial minimum and maximum.

The steps (1–8) we have just outlined are repeated for the specified number of iterations. The complete procedure is summarized in Alg. 17.9 and a corresponding Java implementation can be found on the book’s website (see Sec. 17.4).

Beyond this baseline algorithm, several variations and extensions of this filter exist, including the use of spatially-adaptive, oriented smoothing filters.¹⁷ This type of filter has also been used with good results for *image inpainting* [234], where diffusion is applied to fill out only selected (masked) parts of the image where the content is unknown or should be removed.

Examples

The example in Fig. 17.21 demonstrates the influence of image geometry and how the non-isotropy of the Tschumperl  -Deriche filter can be controlled by varying the diffusion parameters a_1, a_2 (see Eqn. (17.79)). Parameter a_1 , which specifies the diffusion in the direction of contours, is changed while a_2 (controlling the diffusion in the gradient direction) is held constant. In Fig. 17.21(a), smoothing along contours is modest and very small across edges with the default settings $a_1 = 0.5$ and $a_2 = 0.9$. With lower values of a_1 , increased

¹⁷ A recent version was released by the original authors as part of the “GREYC’s Magic Image Converter” open-source framework, which is also available as a GIMP plugin (<http://gmic.sourceforge.net>).

17 EDGE-PRESERVING SMOOTHING FILTERS

Alg. 17.9

Tschumperlé-Deriche anisotropic diffusion filter for vector-valued (color) images. Typical settings are $T = 5, \dots, 20$, $d_t = 20$, $\sigma_g = 0$, $\sigma_s = 0.5$, $a_1 = 0.5$, $a_2 = 0.9$. See Sec. B.4.1 for a description of the procedure `RealEigenValues2x2` (used in line 12).

```

1: TschumperleDericheFilter( $I, T, d_t, \sigma_g, \sigma_s, a_1, a_2$ )
   Input:  $I = (I_1, \dots, I_K)$ , color image of size  $M \times N$  with  $K$ 
          channels;  $T$ , number of iterations;  $d_t$ , time increment;  $\sigma_g$ , width
          of the Gaussian kernel for smoothing the gradient;  $\sigma_s$ , width of
          the Gaussian kernel for smoothing the structure matrix;  $a_1, a_2$ ,
          diffusion parameters for directions of min./max. variation,
          respectively. Returns the modified image  $I$ .
2: Create maps:
    $D : K \times M \times N \rightarrow \mathbb{R}^2$   $\triangleright D(k, u, v) \equiv \nabla I_k(u, v)$ , grad. vector
    $H : K \times M \times N \rightarrow \mathbb{R}^{2 \times 2}$   $\triangleright H(k, u, v) \equiv H_k(u, v)$ , Hess. matrix
    $G : M \times N \rightarrow \mathbb{R}^{2 \times 2}$   $\triangleright G(u, v) \equiv G(u, v)$ , structure matrix
    $A : M \times N \rightarrow \mathbb{R}^{2 \times 2}$   $\triangleright A(u, v) \equiv A(u, v)$ , geometry matrix
    $B : K \times M \times N \rightarrow \mathbb{R}$   $\triangleright B(k, u, v) \equiv \beta_k(u, v)$ , velocity
3: for  $t \leftarrow 1, \dots, T$  do  $\triangleright$  perform  $T$  iterations
4:   for  $k \leftarrow 1, \dots, K$  and all coordinates  $(u, v) \in M \times N$  do
5:      $D(k, u, v) \leftarrow \begin{pmatrix} (I_k * H_x^\nabla)(u, v) \\ (I_k * H_y^\nabla)(u, v) \end{pmatrix}$   $\triangleright$  Eq. 17.68–17.69
6:      $H(k, u, v) \leftarrow \begin{pmatrix} (I_k * H_{xx}^\nabla)(u, v) & (I_k * H_{xy}^\nabla)(u, v) \\ (I_k * H_{xy}^\nabla)(u, v) & (I_k * H_{yy}^\nabla)(u, v) \end{pmatrix}$   $\triangleright$  Eq. 17.71–17.72
7:      $D \leftarrow D * H_G^{\sigma_a}$   $\triangleright$  smooth elements of  $D$  over  $(u, v)$ 
8:     for all coordinates  $(u, v) \in M \times N$  do
9:        $G(u, v) \leftarrow \sum_{k=1}^K \begin{pmatrix} (D_x(k, u, v))^2 & D_x(k, u, v) \cdot D_y(k, u, v) \\ D_x(k, u, v) \cdot D_y(k, u, v) & (D_y(k, u, v))^2 \end{pmatrix}$ 
10:       $G \leftarrow G * H_G^{\sigma_g}$   $\triangleright$  smooth elements of  $G$  over  $(u, v)$ 
11:      for all coordinates  $(u, v) \in M \times N$  do
12:         $(\lambda_1, \lambda_2, q_1, q_2) \leftarrow \text{RealEigenValues2x2}(G(u, v))$   $\triangleright$  p. 724
13:         $\hat{q}_1 \leftarrow \begin{pmatrix} \hat{x}_1 \\ \hat{y}_1 \end{pmatrix} = \frac{q_1}{\|q_1\|}$   $\triangleright$  normalize 1st eigenvector ( $\lambda_1 \geq \lambda_2$ )
14:         $c_1 \leftarrow \frac{1}{(1+\lambda_1+\lambda_2)^{a_1}}, c_2 \leftarrow \frac{1}{(1+\lambda_1+\lambda_2)^{a_2}}$   $\triangleright$  Eq. 17.79
15:         $A(u, v) \leftarrow \begin{pmatrix} c_1 \cdot \hat{y}_1^2 + c_2 \cdot \hat{x}_1^2 & (c_2 - c_1) \cdot \hat{x}_1 \cdot \hat{y}_1 \\ (c_2 - c_1) \cdot \hat{x}_1 \cdot \hat{y}_1 & c_1 \cdot \hat{x}_1^2 + c_2 \cdot \hat{y}_1^2 \end{pmatrix}$   $\triangleright$  Eq. 17.78
16:         $\beta_{\max} \leftarrow -\infty$ 
17:        for  $k \leftarrow 1, \dots, K$  and all  $(u, v) \in M \times N$  do
18:           $B(k, u, v) \leftarrow \text{trace}(A(u, v) \cdot H(k, u, v))$   $\triangleright \beta_k$ , Eq. 17.82
19:           $\beta_{\max} \leftarrow \max(\beta_{\max}, |B(k, u, v)|)$ 
20:         $\alpha \leftarrow d_t / \beta_{\max}$   $\triangleright$  Eq. 17.83
21:        for  $k \leftarrow 1, \dots, K$  and all  $(u, v) \in M \times N$  do
22:           $I_k(u, v) \leftarrow I_k(u, v) + \alpha \cdot B(k, u, v)$   $\triangleright$  update the image
23: return  $I$ 

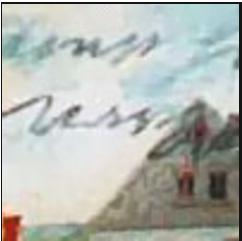
```

blurring occurs in the direction of the contours, as shown in Figs. 17.21(b, c).

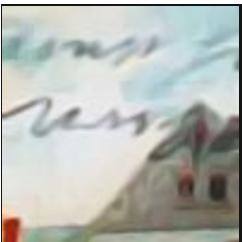
17.4 Java Implementation

Implementations of the filters described in this chapter are available as part of the `imagingbook`¹⁸ library at the book's website. The associated classes `KuwaharaFilter`, `NagaMatsuyamaFilter`, `PeronaMalikFilter` and `TschumperleDericheFilter` are based on

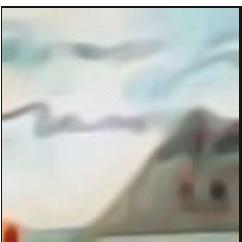
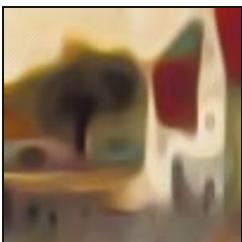
¹⁸ Package `imagingbook.pub.edgepreservingfilters`.



(a) $a_1 = 0.50$



(b) $a_1 = 0.25$



(c) $a_1 = 0.00$

17.4 JAVA IMPLEMENTATION

Fig. 17.21

Tschumperlé-Deriche filter example. The non-isotropy of the filter can be adjusted by changing parameter a_1 , which controls the diffusion along contours (see Eqn. (17.79)): $a_1 = 0.50, 0.25, 0.00$ (a–c). Parameter $a_2 = 0.90$ (constant) controls the diffusion in the direction of the gradient (perpendicular to contours). Remaining settings are $T = 20$, $d_t = 20$, $\sigma_g = 0.5$, $\sigma_s = 0.5$ (see the description of Alg. 17.9); original image in Fig. 17.3(a).

the common super-class `GenericFilter`¹⁹ and define the following constructors:

KuwaharaFilter (Parameters p)

Creates a Kuwahara-type filter for grayscale and color images, as described in Sec. 17.1 (Alg. 17.2), with radius r (default 2) and variance threshold `tsigma` (denoted t_σ in Alg. 17.2, default 0.0). The size of the resulting filter is $(2r + 1) \times (2r + 1)$.

BilateralFilter (Parameters p)

Creates a bilateral filter for grayscale and color images using Gaussian kernels, as described in Sec. 17.2 (see Algs. 17.4 and 17.5). Parameters `sigmaD` (σ_d , default 2.0) and `sigmaR` (σ_r , default 50.0) specify the widths of the domain and the range kernels, respectively. The type of norm for measuring color distances is specified by `colorNormType` (default is `NormType.L2`).

BilateralFilterSeparable (Parameters p)

Creates a x/y -separable bilateral filter for grayscale and color images, (see Alg. 17.6). Constructor parameters are the same as for the class `BilateralFilter` above.

¹⁹ Package `imagingbook.lib.filters`. Filters of this type can be applied to images using the method `applyTo(ImageProcessor ip)`, as described in Chapter 15, Sec. 15.3.

PeronaMalikFilter (Parameters p)

Creates an anisotropic diffusion filter for grayscale and color images (see Algs. 17.7 and 17.8). The key parameters and their default values are `iterations` ($T = 10$), `alpha` ($\alpha = 0.2$), `kappa` ($\kappa = 25$), `smoothRegions` (`true`), `colorMode` (`SeparateChannels`). With `smoothRegions = true`, function $g_\kappa^{(2)}$ is used to control conductivity, otherwise $g_\kappa^{(1)}$ (see Eqn. (17.52)). For filtering color images, three different color modes can be specified for diffusion control: `SeparateChannels`, `BrightnessGradient`, or `ColorGradient`. See Prog. 17.1 for an example of using this class in a simple ImageJ plugin.

TschumperleDericheFilter (Parameters p)

Creates an anisotropic diffusion filter for color images, as described in Sec. 17.3.4 (Alg. 17.9). Parameters and default values are `iterations` ($T = 20$), `dt` ($d_t = 20$), `sigmaG` ($\sigma_g = 0.0$), `sigmaS` ($\sigma_s = 0.5$), `a1` ($a_1 = 0.25$), `a2` ($a_2 = 0.90$). Otherwise the usage of this class is analogous to the example in Prog. 17.1.

All default values pertain to the parameterless constructors that are also available. Note that these filters are generic and can be applied to grayscale and color images without any modification.

17.5 Exercises

Exercise 17.1. Implement a pure *range filter* (Eqn. (17.17)) for grayscale images, using a 1D Gaussian kernel

$$H_r(x) = \frac{1}{\sqrt{2\pi \cdot \sigma}} \cdot \exp\left(-\frac{x^2}{2\sigma^2}\right).$$

Investigate the effects of this filter upon the image and its histogram for $\sigma = 10, 20$, and 25 .

Exercise 17.2. Modify the Kuwahara-type filter for color images in Alg. 17.3 to use the *norm of the color covariance matrix* (as defined in Eqn. (17.12)) for quantifying the amount of variation in each subregion. Estimate the number of additional calculations required for processing each image pixel. Implement the modified algorithm, compare the results and execution times.

Exercise 17.3. Modify the separable bilateral filter algorithm (given in Alg. 17.6) to handle color images, using Alg. 17.5 as a starting point. Implement and test your algorithm, compare the results (see also Fig. 17.14) and execution times.

Exercise 17.4. Verify (experimentally) that n iterations of the diffusion process defined in Eqn. (17.45) have the same effect as a Gaussian filter of width σ_n , as stated in Eqn. (17.48). To determine the impulse response of the resulting diffusion filter, use an “impulse” test image, that is, a black (zero-valued) image with a single bright pixel at the center.

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import imagingbook...PeronaMalikFilter;
5 import imagingbook...PeronaMalikFilter.ColorMode;
6 import imagingbook...PeronaMalikFilter.Parameters;
7
8 public class Perona_Malik_Demo implements PlugInFilter {
9
10    public int setup(String arg0, ImagePlus imp) {
11        return DOES_ALL + DOES_STACKS;
12    }
13
14    public void run(ImageProcessor ip) {
15        // create a parameter object:
16        Parameters params = new Parameters();
17
18        // modify filter settings if needed:
19        params.iterations = 20;
20        params.alpha = 0.15f;
21        params.kappa = 20.0f;
22        params.smoothRegions = true;
23        params.colorMode = ColorMode.ColorGradient;
24
25        // instantiate the filter object:
26        PeronaMalikFilter filter =
27            new PeronaMalikFilter(params);
28
29        // apply the filter:
30        filter.applyTo(ip);
31    }
32
33 }

```

17.5 EXERCISES

Prog. 17.1

Perona-Malik filter (complete ImageJ plugin). Inside the `run()` method, a parameter object (instance of class `PeronaMalikFilter.Parameters`) is created in line 16. Individual parameters may then be modified, as shown in lines 19–23. This would typically be done by querying the user (e.g., with ImageJ’s `GenericDialog` class). In line 27, a new instance of `PeronaMalikFilter` is created, the parameter object (`params`) being passed to the constructor as the only argument. Finally, in line 30, the filter is (destructively) applied to the input image, that is, `ip` is modified. `ColorMode` (in line 23) is implemented as an enumeration type within class `PeronaMalikFilter`, providing the options `SeparateChannels` (default), `BrightnessGradient` and `ColorGradient`. Note that, as specified in the `setup()` method, this plugin works for any type of image and image stacks.

Exercise 17.5. Use the signal-to-noise ratio (SNR) to measure the effectiveness of noise suppression by edge-preserving smoothing filters on grayscale images. Add synthetic Gaussian noise (see Sec. D.4.3 in the Appendix) to the original image I to create a corrupted image \tilde{I} . Then apply the filter to \tilde{I} to obtain $\tilde{\tilde{I}}$. Finally, calculate $\text{SNR}(I, \tilde{\tilde{I}})$ as defined in Eqn. (13.2). Compare the SNR values obtained with various types of filters and different parameter settings, for example, for the *Kuwahara filter* (Alg. 17.2), the *bilateral filter* (Alg. 17.4), and the *Perona-Malik* anisotropic diffusion filter (Alg. 17.7). Analyze if and how the SNR values relate to the perceived image quality.

Introduction to Spectral Techniques

The following three chapters deal with the representation and analysis of images in the frequency domain, based on the decomposition of image signals into sine and cosine functions using the well-known *Fourier transform*. Students often consider this a difficult topic, mainly because of its mathematical flavor and that its practical applications are not immediately obvious. Indeed, most common operations and methods in digital image processing can be sufficiently described in the original signal or image space without even mentioning spectral techniques. This is the reason why we pick up this topic relatively late in this text.

While spectral techniques were often used to improve the efficiency of image-processing operations, this has become increasingly less important due to the high power of modern computers. There exist, however, some important effects, concepts, and techniques in digital image processing that are considerably easier to describe in the frequency domain or cannot otherwise be understood at all. The topic should therefore not be avoided all together. Fourier analysis not only owns a very elegant (perhaps not always sufficiently appreciated) mathematical theory but interestingly enough also complements some important concepts we have seen earlier, in particular linear filters and linear *convolution* (see Chapter 5, Sec. 5.2). Equally important are applications of spectral techniques in many popular methods for image and video compression, and they provide valuable insight into the mechanisms of sampling (discretization) of continuous signals as well as the reconstruction and interpolation of discrete signals.

In the following, we first give a basic introduction to the concepts of frequency and spectral decomposition that tries to be minimally formal and thus should be easily “digestible” even for readers without previous exposure to this topic. We start with the representation of 1D signals and will then extend the discussion to 2D signals (images) in the next chapter. Subsequently, Chapter 20 briefly explains the *discrete cosine transform*, a popular variant of the discrete Fourier transform that is frequently used in image compression.

18.1 The Fourier Transform

The concept of frequency and the decomposition of waveforms into elementary “harmonic” functions first arose in the context of music and sound. The idea of describing acoustic events in terms of “pure” sinusoidal functions does not seem unreasonable, considering that sine waves appear naturally in every form of oscillation (e.g., on a free-swinging pendulum).

18.1.1 Sine and Cosine Functions

The well-known *cosine* function,

$$f(x) = \cos(x), \quad (18.1)$$

has the value 1 at the origin ($\cos(0) = 1$) and performs exactly *one* full cycle between the origin and the point $x = 2\pi$ (Fig. 18.1(a)). We say that the function is periodic with a cycle length (period) $T = 2\pi$; that is,

$$\cos(x) = \cos(x + 2\pi) = \cos(x + 4\pi) = \dots = \cos(x + k2\pi), \quad (18.2)$$

for any $k \in \mathbb{Z}$. The same is true for the corresponding *sine* function, except that its value is zero at the origin (since $\sin(0) = 0$).

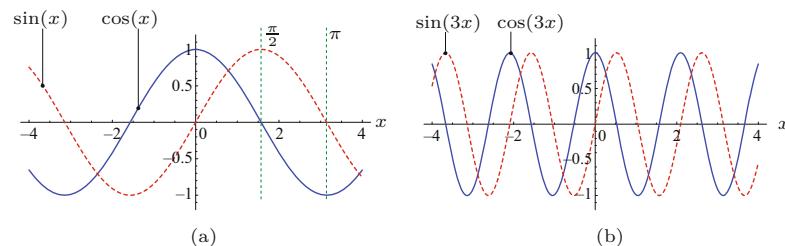


Fig. 18.1

Cosine and sine functions of different frequency. The expression $\cos(\omega x)$ describes a cosine function with angular frequency ω at position x . The angular frequency ω of this periodic function corresponds to a cycle length (period) $T = 2\pi/\omega$. For $\omega = 1$, the period is $T_1 = 2\pi$ (a), and for $\omega = 3$ it is $T_3 = 2\pi/3 \approx 2.0944$ (b). The same holds for the sine function $\sin(\omega x)$.

Frequency and amplitude

The number of oscillations of $\cos(x)$ over the distance $T = 2\pi$ is *one* and thus the value of the *angular frequency*

$$\omega = \frac{2\pi}{T} = 1. \quad (18.3)$$

If we modify the cosine function in Eqn. (18.1) to

$$f(x) = \cos(3x), \quad (18.4)$$

we obtain a compressed cosine wave that oscillates three times faster than the original function $\cos(x)$ (see Fig. 18.1(b)). The function $\cos(3x)$ performs three full cycles over a distance of 2π and thus has the angular frequency $\omega = 3$ and a period $T = \frac{2\pi}{3}$. In general, the period T relates to the angular frequency ω as

$$T = \frac{2\pi}{\omega}, \quad (18.5)$$

for $\omega > 0$. A sine or cosine function oscillates between peak values $+1$ and -1 , and its *amplitude* is 1. Multiplying by a constant $a \in \mathbb{R}$

changes the peak values of the function to $\pm a$ and its *amplitude* to a . In general, the expressions

18.1 THE FOURIER TRANSFORM

$$a \cdot \cos(\omega x) \quad \text{and} \quad a \cdot \sin(\omega x)$$

denote a cosine or sine function, respectively, with amplitude a and angular frequency ω , evaluated at position (or point in time) x . The relation between the angular frequency ω and the “common” frequency f is given by

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{or} \quad \omega = 2\pi f, \quad (18.6)$$

respectively, where f is measured in cycles per length or time unit.¹ In the following, we use either ω or f as appropriate, and the meaning should always be clear from the symbol used.

Phase

Shifting a cosine function along the x axis by a distance φ ,

$$\cos(x) \rightarrow \cos(x - \varphi),$$

changes the *phase* of the cosine wave, and φ denotes the *phase angle* of the resulting function. Thus a sine function is really just a cosine function shifted to the right² by a quarter period ($\varphi = \frac{2\pi}{4} = \frac{\pi}{2}$), so

$$\sin(\omega x) = \cos\left(\omega x - \frac{\pi}{2}\right). \quad (18.7)$$

If we take the cosine function as the reference with phase $\varphi_{\cos} = 0$, then the phase angle of the corresponding sine function is $\varphi_{\sin} = \frac{\pi}{2} = 90^\circ$.

Cosine and sine functions are “orthogonal” in a sense and we can use this fact to create new “sinusoidal” functions with arbitrary frequency, phase, and amplitude. In particular, adding a cosine and a sine function with the identical frequencies ω and arbitrary amplitudes A and B , respectively, creates another sinusoid:

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi). \quad (18.8)$$

The resulting amplitude C and the phase angle φ are defined only by the two original amplitudes A and B as

$$C = \sqrt{A^2 + B^2} \quad \text{and} \quad \varphi = \tan^{-1}\left(\frac{B}{A}\right). \quad (18.9)$$

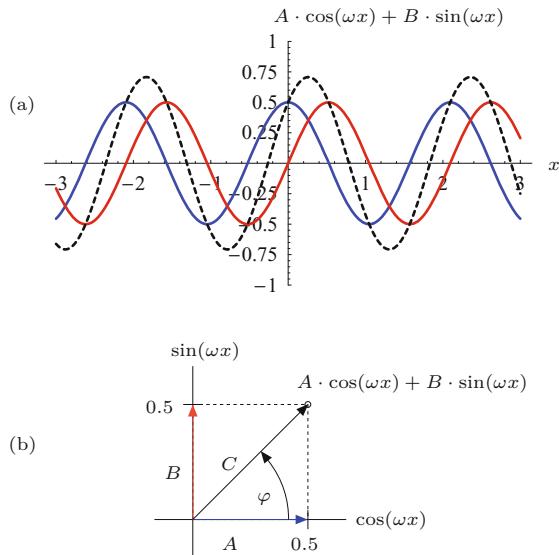
Figure 18.2(a) shows an example with amplitudes $A = B = 0.5$ and a resulting phase angle $\varphi = 45^\circ$.

¹ For example, a temporal oscillation with frequency $f = 1000$ cycles/s (Hertz) has the period $T = 1/1000$ s and therefore the angular frequency $\omega = 2000\pi$. The latter is a unitless quantity.

² In general, the function $f(x-d)$ is the original function $f(x)$ shifted to the right by a distance d .

Fig. 18.2

Adding cosine and sine functions with identical frequencies, $A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$, with $\omega = 3$ and $A = B = 0.5$. The result is a phase-shifted cosine function (dotted curve) with amplitude $C = \sqrt{0.5^2 + 0.5^2} \approx 0.707$ and phase angle $\varphi = 45^\circ$ (a). If the cosine and sine components are treated as orthogonal vectors (A, B) in 2-space, the amplitude and phase of the resulting sinusoid (C) can be easily determined by vector summation (b).



Complex-valued sine functions—Euler's notation

Figure 18.2(b) depicts the contributing cosine and sine components of the new function as a pair of orthogonal vectors in 2-space whose *lengths* correspond to the amplitudes A and B . Not coincidentally, this reminds us of the representation of real and imaginary components of complex numbers,

$$z = a + i b,$$

in the 2D plane \mathbb{C} , where i is the imaginary unit ($i^2 = -1$). This association becomes even stronger if we look at Euler's famous notation of complex numbers along the unit circle,

$$z = e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta), \quad (18.10)$$

where $e \approx 2.71828$ is the Euler number. If we take the expression $e^{i\theta}$ as a function of the angle θ rotating around the unit circle, we obtain a “complex-valued sinusoid” whose real and imaginary parts correspond to a cosine and a sine function, respectively,

$$\begin{aligned} \operatorname{Re}(e^{i\theta}) &= \cos(\theta), \\ \operatorname{Im}(e^{i\theta}) &= \sin(\theta). \end{aligned} \quad (18.11)$$

Since $z = e^{i\theta}$ is placed on the unit circle, the *amplitude* of the complex-valued sinusoid is $|z| = r = 1$. We can easily modify the amplitude of this function by multiplying it by some real value $a \geq 0$, that is,

$$|a \cdot e^{i\theta}| = a \cdot |e^{i\theta}| = a. \quad (18.12)$$

Similarly, we can alter the *phase* of a complex-valued sinusoid by adding a phase angle φ in the function's exponent or, equivalently, by multiplying it by a complex-valued constant $c = e^{i\varphi}$,

$$e^{i(\theta+\varphi)} = e^{i\theta} \cdot e^{i\varphi}. \quad (18.13)$$

In summary, multiplying by some real value affects only the *amplitude* of a sinusoid, while multiplying by some complex value c (with unit amplitude $|c| = 1$) modifies only the function's *phase* (without changing its amplitude). In general, of course, multiplying by some arbitrary complex value changes both the amplitude *and* the phase of the function (also see Sec. A.3 in the Appendix).

The complex notation makes it easy to combine orthogonal pairs of sine functions $\cos(\omega x)$ and $\sin(\omega x)$ with identical frequencies ω into a single expression,

$$e^{i\theta} = e^{i\omega x} = \cos(\omega x) + i \cdot \sin(\omega x). \quad (18.14)$$

We will make more use of this notation later (in Sec. 18.1.4) to explain the Fourier transform.

18.1.2 Fourier Series Representation of Periodic Functions

As we demonstrated in Eqn. (18.8), sinusoidal functions of arbitrary frequency, amplitude, and phase can be described as the sum of suitably weighted cosine and sine functions. One may wonder if non-sinusoidal functions can also be decomposed into a sum of cosine and sine functions. The answer is yes, of course. It was Fourier³ who first extended this idea to arbitrary functions and showed that (almost) any periodic function $g(x)$ with a fundamental frequency ω_0 can be described as a—possibly infinite—sum of “harmonic” sinusoids, that is,

$$g(x) = \sum_{k=0}^{\infty} A_k \cdot \cos(k\omega_0 x) + B_k \cdot \sin(k\omega_0 x). \quad (18.15)$$

This is called a *Fourier series*, and the constant factors A_k , B_k are the *Fourier coefficients* of the function $g(x)$. Notice that in Eqn. (18.15) the frequencies of the sine and cosine functions contributing to the Fourier series are integral multiples (“harmonics”) of the fundamental frequency ω_0 , including the zero frequency for $k = 0$. The corresponding coefficients A_k and B_k , which are initially unknown, can be uniquely derived from the original function $g(x)$. This process is commonly referred to as *Fourier analysis*.

18.1.3 Fourier Integral

Fourier did not want to limit this concept to periodic functions and postulated that nonperiodic functions, too, could be described as sums of sine and cosine functions. While this proved to be true in principle, it generally requires—beyond multiples of the fundamental frequency ($k\omega_0$)—infinitely many, densely spaced frequencies! The resulting decomposition,

$$g(x) = \int_0^{\infty} A_{\omega} \cdot \cos(\omega x) + B_{\omega} \cdot \sin(\omega x) \, d\omega, \quad (18.16)$$

is called a *Fourier integral* and the coefficients A_{ω} , B_{ω} are again the weights for the corresponding cosine and sine functions with the

³ Jean-Baptiste Joseph de Fourier (1768–1830).

(continuous) frequency ω . The Fourier integral is the basis of the Fourier spectrum and the Fourier transform, as will be described (for details, see, e.g., [35, Ch. 15, Sec. 15.3]).

In Eqn. (18.16), every coefficient A_ω and B_ω specifies the *amplitude* of the corresponding cosine or sine function, respectively. The coefficients thus define “how much of each frequency” contributes to a given function or signal $g(x)$. But what are the proper values of these coefficients for a given function $g(x)$, and can they be determined uniquely? The answer is yes again, and the “recipe” for computing the coefficients is amazingly simple:

$$\begin{aligned} A_\omega &= A(\omega) = \frac{1}{\pi} \cdot \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) \, dx, \\ B_\omega &= B(\omega) = \frac{1}{\pi} \cdot \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) \, dx. \end{aligned} \quad (18.17)$$

Since this representation of the function $g(x)$ involves infinitely many densely spaced frequency values ω , the corresponding coefficients $A(\omega)$ and $B(\omega)$ are indeed continuous functions as well. They hold the continuous distribution of frequency components contained in the original signal, which is called a “spectrum”.

Thus the Fourier integral in Eqn. (18.16) describes the original function $g(x)$ as a sum of infinitely many cosine and sine functions, with the corresponding Fourier coefficients contained in the functions $A(\omega)$ and $B(\omega)$. In addition, a signal $g(x)$ is uniquely and fully represented by the corresponding coefficient functions $A(\omega)$ and $B(\omega)$. We know from Eqn. (18.17) how to compute the spectrum for a given function $g(x)$, and Eqn. (18.16) explains how to reconstruct the original function from its spectrum if it is ever needed.

18.1.4 Fourier Spectrum and Transformation

There is now only a small remaining step from the decomposition of a function $g(x)$, as shown in Eqn. (18.17), to the “real” Fourier transform. In contrast to the Fourier *integral*, the Fourier *transform* treats both the original signal and the corresponding spectrum as *complex-valued* functions, which considerably simplifies the resulting notation.

Based on the functions $A(\omega)$ and $B(\omega)$ defined in the Fourier integral (Eqn. (18.17)), the *Fourier spectrum* $G(\omega)$ of a function $g(x)$ is given as

$$\begin{aligned} G(\omega) &= \sqrt{\frac{\pi}{2}} \cdot [A(\omega) - i \cdot B(\omega)] \\ &= \sqrt{\frac{\pi}{2}} \cdot \left[\frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) \, dx - i \cdot \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) \, dx \right] \\ &= \frac{1}{\sqrt{2\pi}} \cdot \int_{-\infty}^{\infty} g(x) \cdot [\cos(\omega x) - i \cdot \sin(\omega x)] \, dx, \end{aligned} \quad (18.18)$$

with $g(x), G(\omega) \in \mathbb{C}$. Using Euler’s notation of complex values (see Eqn. (18.14)) yields the continuous Fourier spectrum in Eqn. (18.18) in its common form:

$$\begin{aligned} G(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot [\cos(\omega x) - i \cdot \sin(\omega x)] dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot e^{-i\omega x} dx. \end{aligned} \tag{18.19}$$

18.1 THE FOURIER TRANSFORM

The transition from the function $g(x)$ to its Fourier spectrum $G(\omega)$ is called the *Fourier transform*⁴ (\mathcal{F}). Conversely, the original function $g(x)$ can be reconstructed completely from its Fourier spectrum $G(\omega)$ using the *inverse Fourier transform*⁵ (\mathcal{F}^{-1}), defined as

$$\begin{aligned} g(x) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot [\cos(\omega x) + i \cdot \sin(\omega x)] d\omega \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot e^{i\omega x} d\omega. \end{aligned} \tag{18.20}$$

In general, even if one of the involved functions ($g(x)$ or $G(\omega)$) is real-valued (which is usually the case for physical signals $g(x)$), the other function is complex-valued. One may also note that the forward transformation \mathcal{F} (Eqn. (18.19)) and the inverse transformation \mathcal{F}^{-1} (Eqn. (18.20)) are almost completely symmetrical, the sign of the exponent being the only difference.⁶ The spectrum produced by the Fourier transform is a new representation of the signal in a space of frequencies. Apparently, this “frequency space” and the original “signal space” are *dual* and interchangeable mathematical representations.

18.1.5 Fourier Transform Pairs

The relationship between a function $g(x)$ and its Fourier spectrum $G(\omega)$ is unique in both directions: the Fourier spectrum is uniquely defined for a given function, and for any Fourier spectrum there is only one matching signal—the two functions $g(x)$ and

$$g(x) \circledcirc G(\omega).$$

Table 18.1 lists the transform pairs for some selected analytical functions, which are also shown graphically in **Figs. 18.3** and **18.4**.

The Fourier spectrum of a *cosine function* $\cos(\omega_0 x)$, for example, consists of two separate thin pulses arranged symmetrically at a distance ω_0 from the origin (**Fig. 18.3(a,c)**). Intuitively, this corresponds to our physical understanding of a spectrum (e.g., if we think of a pure monophonic sound in acoustics or the thin line produced by some extremely pure color in the optical spectrum). Increasing the frequency ω_0 would move the corresponding pulses in the spectrum

⁴ Also called the “direct” or “forward” transformation.

⁵ Also called “backward” transformation.

⁶ Various definitions of the Fourier transform are in common use. They are contrasted mainly by the constant factors outside the integral and the signs of the exponents in the forward and inverse transforms, but all versions are equivalent in principle. The symmetric variant shown here uses the same factor $(1/\sqrt{2\pi})$ in the forward and inverse transforms.

Table 18.1

Fourier transforms of selected analytical functions; $\delta()$ denotes the “impulse” or *Dirac* function (see Sec. 18.2.1).

Function	Transform pair $g(x) \circ\bullet G(\omega)$	Figure
Cosine function with frequency ω_0	$g(x) = \cos(\omega_0 x)$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot (\delta(\omega + \omega_0) + \delta(\omega - \omega_0))$	18.3(a,c)
Sine function with frequency ω_0	$g(x) = \sin(\omega_0 x)$ $G(\omega) = i\sqrt{\frac{\pi}{2}} \cdot (\delta(\omega + \omega_0) - \delta(\omega - \omega_0))$	18.3(b,d)
Gaussian function of width σ	$g(x) = \frac{1}{\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$ $G(\omega) = e^{-\frac{\sigma^2 \omega^2}{2}}$	18.4(a,b)
Rectangular pulse of width $2b$	$g(x) = \Pi_b(x) = \begin{cases} 1 & x \leq b \\ 0 & \text{sonst} \end{cases}$ $G(\omega) = \frac{2b \sin(b\omega)}{\sqrt{2\pi}\omega}$	18.4(c,d)

away from the origin. Notice that the spectrum of the cosine function is real-valued, the imaginary part being zero. Of course, the same relation holds for the sine function (Fig. 18.3(b,d)), with the only difference being that the pulses have different polarities and appear in the imaginary part of the spectrum. In this case, the real part of the spectrum $G(\omega)$ is zero.

The *Gaussian function* is particularly interesting because its Fourier spectrum is also a Gaussian function (Fig. 18.4(a,b))! It is one of the few examples where the function type in frequency space is the same as in signal space. With the Gaussian function, it is also clear to see that *stretching* a function in signal space corresponds to *shortening* its spectrum and vice versa.

The Fourier transform of a *rectangular pulse* (Fig. 18.4(c,d)) is the “Sinc” function of type $\sin(x)/x$. With increasing frequencies, this function drops off quite slowly, which shows that the components contained in the original rectangular signal are spread out over a large frequency range. Thus a rectangular pulse function exhibits a very wide spectrum in general.

18.1.6 Important Properties of the Fourier Transform

Symmetry

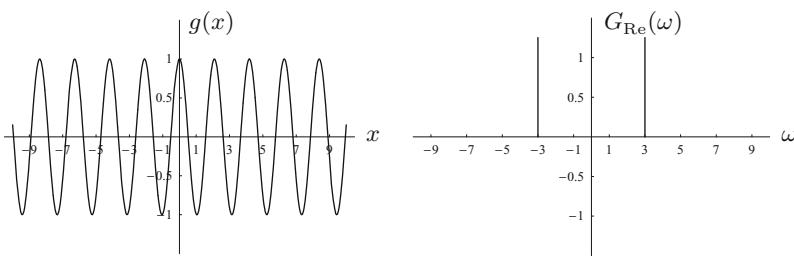
The Fourier spectrum extends over positive and negative frequencies and could, in principle, be an arbitrary complex-valued function. However, in many situations, the spectrum is symmetric about its origin (see, e.g., [43, p. 178]). In particular, the Fourier transform of a real-valued signal $g(x) \in \mathbb{R}$ is a so-called *Hermite* function with the property

$$G(\omega) = G^*(-\omega), \quad (18.21)$$

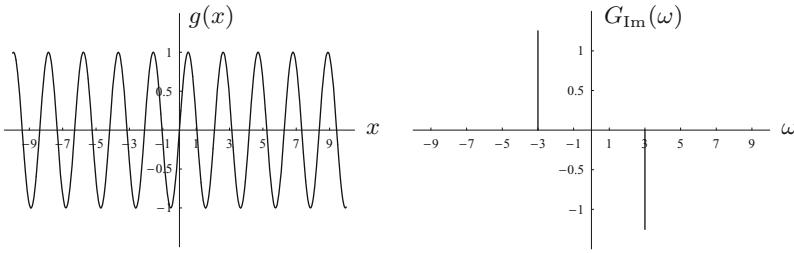
where G^* denotes the complex conjugate of G (see also Sec. A.3 in the Appendix).

18.1 THE FOURIER TRANSFORM

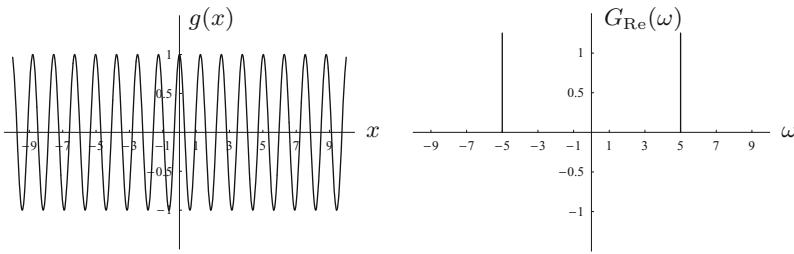
Fig. 18.3
Fourier transform pairs—cosine and sine functions.



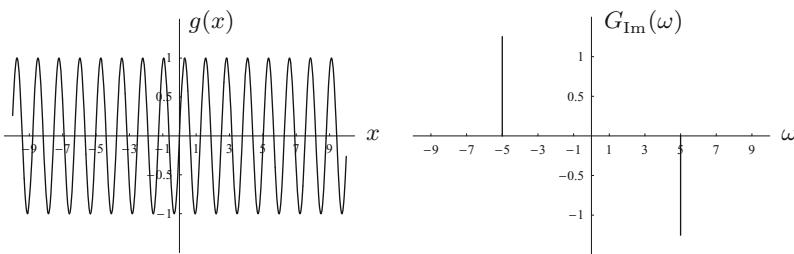
(a) Cosine ($\omega_0=3$): $g(x) = \cos(3x) \Leftrightarrow G(\omega) = \sqrt{\frac{\pi}{2}} \cdot (\delta(\omega+3) + \delta(\omega-3))$



(b) Sine ($\omega_0=3$): $g(x) = \sin(3x) \Leftrightarrow G(\omega) = i\sqrt{\frac{\pi}{2}} \cdot (\delta(\omega+3) - \delta(\omega-3))$



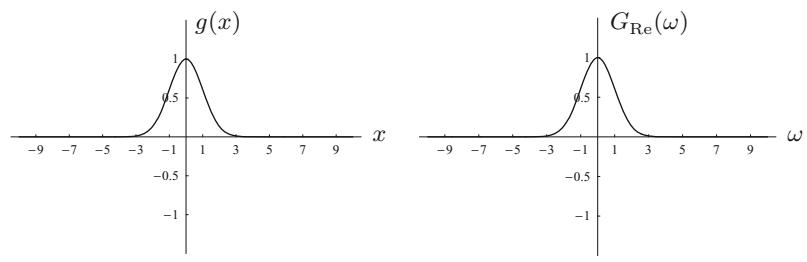
(c) Cosine ($\omega_0=5$): $g(x) = \cos(5x) \Leftrightarrow G(\omega) = \sqrt{\frac{\pi}{2}} \cdot (\delta(\omega+5) + \delta(\omega-5))$



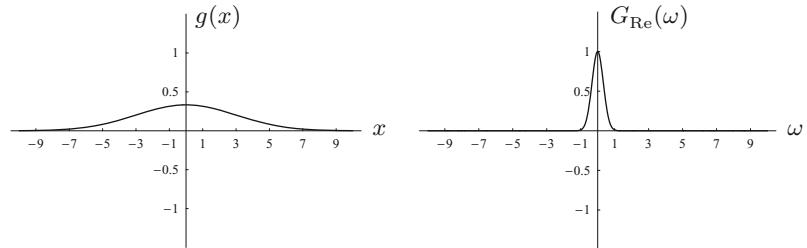
(d) Sine ($\omega_0=5$): $g(x) = \sin(5x) \Leftrightarrow G(\omega) = i\sqrt{\frac{\pi}{2}} \cdot (\delta(\omega+5) - \delta(\omega-5))$

18 INTRODUCTION TO SPECTRAL TECHNIQUES

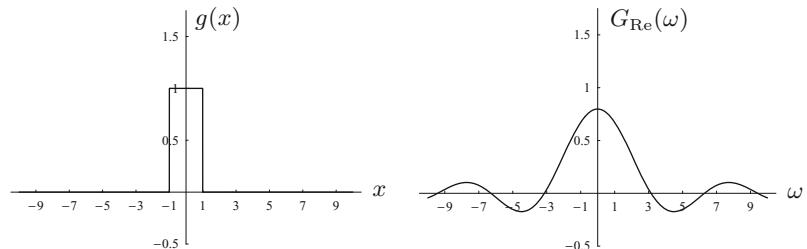
Fig. 18.4
Fourier transform pairs—Gaussian functions and square pulses.



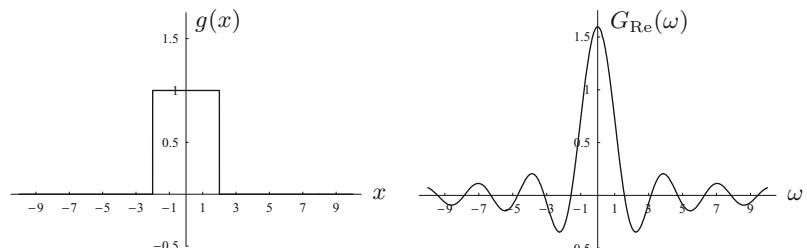
$$(a) \text{ Gauss. } (\sigma=1): g(x) = e^{-\frac{x^2}{2}} \quad \circ \bullet \quad G(\omega) = e^{-\frac{\omega^2}{2}}$$



$$(b) \text{ Gauss. } (\sigma=3): g(x) = \frac{1}{3} \cdot e^{-\frac{x^2}{2 \cdot 9}} \quad \circ \bullet \quad G(\omega) = e^{-\frac{9\omega^2}{2}}$$



$$(c) \text{ Pulse } (b=1): g(x) = \Pi_1(x) \quad \circ \bullet \quad G(\omega) = \frac{2 \sin(\omega)}{\sqrt{2\pi}\omega}$$



$$(d) \text{ Pulse } (b=2): g(x) = \Pi_2(x) \quad \circ \bullet \quad G(\omega) = \frac{4 \sin(2\omega)}{\sqrt{2\pi}\omega}$$

Linearity

The Fourier transform is also a *linear* operation such that multiplying the signal by a constant value $c \in \mathbb{C}$ scales the corresponding spectrum by the same amount,

$$a \cdot g(x) \circledast a \cdot G(\omega). \quad (18.22)$$

Linearity also means that the transform of the sum of two signals $g(x) = g_1(x) + g_2(x)$ is identical to the sum of their individual transforms $G_1(\omega)$ and $G_2(\omega)$ and thus

$$g_1(x) + g_2(x) \circledast G_1(\omega) + G_2(\omega). \quad (18.23)$$

Similarity

If the original function $g(x)$ is scaled in space or time, the opposite effect appears in the corresponding Fourier spectrum. In particular, as observed on the Gaussian function in Fig. 18.4, *stretching* a signal by a factor s (i.e., $g(x) \rightarrow g(sx)$) leads to a *shortening* of the Fourier spectrum:

$$g(sx) \circledast \frac{1}{|s|} \cdot G\left(\frac{\omega}{s}\right). \quad (18.24)$$

Similarly, the signal is shortened if the corresponding spectrum is stretched.

Shift property

If the original function $g(x)$ is shifted by a distance d along its coordinate axis (i.e., $g(x) \rightarrow g(x-d)$), then the Fourier spectrum multiplies by the complex value $e^{-i\omega d}$ dependent on ω :

$$g(x-d) \circledast e^{-i\omega d} \cdot G(\omega). \quad (18.25)$$

Since $e^{-i\omega d}$ lies on the unit circle, the multiplication causes a phase shift on the spectral values (i.e., a redistribution between the real and imaginary components) without altering the magnitude $|G(\omega)|$. Obviously, the amount (angle) of phase shift (ωd) is proportional to the angular frequency ω .

Convolution property

From the image-processing point of view, the most interesting property of the Fourier transform is its relation to linear convolution (see Ch. 5, Sec. 5.3.1). Let us assume that we have two functions $g(x)$ and $h(x)$ and their corresponding Fourier spectra $G(\omega)$ and $H(\omega)$, respectively. If the original functions are subject to linear convolution (i.e., $g(x) * h(x)$), then the Fourier transform of the result equals the (pointwise) product of the individual Fourier transforms $G(\omega)$ and $H(\omega)$:

$$g(x) * h(x) \circledast G(\omega) \cdot H(\omega). \quad (18.26)$$

Due to the duality of signal space and frequency space, the same also holds in the opposite direction; i.e., a pointwise multiplication of two signals is equivalent to convolving the corresponding spectra:

$$g(x) \cdot h(x) \circledast G(\omega) * H(\omega). \quad (18.27)$$

A multiplication of the functions in *one* space (signal or frequency space) thus corresponds to a linear convolution of the Fourier spectra in the *opposite* space.

18.2 Working with Discrete Signals

The definition of the continuous Fourier transform in Sec. 18.1 is of little use for numerical computation on a computer. Neither can arbitrary continuous (and possibly infinite) functions be represented in practice. Nor can the required integrals be computed. In reality, we must always deal with *discrete* signals, and we therefore need a new version of the Fourier transform that treats signals and spectra as finite data vectors—the “discrete” Fourier transform. Before continuing with this issue we want to use our existing wisdom to take a closer look at the process of discretizing signals in general.

18.2.1 Sampling

We first consider the question of how a continuous function can be converted to a discrete signal in the first place. This process is usually called “sampling” (i.e., taking samples of the continuous function at certain points in time (or in space), usually spaced at regular distances). To describe this step in a simple but formal way, we require an inconspicuous but nevertheless important piece from the mathematician’s toolbox.

The impulse function $\delta(x)$

We casually encountered the impulse function (also called the *delta* or *Dirac* function) earlier when we looked at the impulse response of linear filters (see Ch. 5, Sec. 5.3.4) and in the Fourier transforms of the cosine and sine functions (Fig. 18.3). This function, which models a continuous “ideal” impulse, is unusual in several respects: its value is zero everywhere except at the origin, where it is nonzero (though undefined), but its integral is one, that is,

$$\delta(x) = 0 \text{ for } x \neq 0 \quad \text{and} \quad \int_{-\infty}^{\infty} \delta(x) dx = 1. \quad (18.28)$$

One could imagine $\delta(x)$ as a single pulse at position $x = 0$ that is infinitesimally narrow but still contains finite energy (1). Also remarkable is the impulse function’s behavior under scaling along the time (or space) axis (i.e., $\delta(x) \rightarrow \delta(sx)$), with

$$\delta(sx) = \frac{1}{|s|} \cdot \delta(x), \quad (18.29)$$

for $s \neq 0$. Despite the fact that $\delta(x)$ does not exist in physical reality and cannot be plotted (the corresponding plots in Fig. 18.3 are for illustration only), this function is a useful mathematical tool for describing the sampling process, as will be shown.

Sampling with the impulse function

Using the concept of the ideal impulse, the sampling process can be described in a straightforward and intuitive way.⁷ If a continuous

⁷ The following description is intentionally a bit superficial (in a mathematical sense). See, for example, [43, 128] for more precise coverage of these topics.

function $g(x)$ is multiplied with the impulse function $\delta(x)$, we obtain a new function

$$\bar{g}(x) = g(x) \cdot \delta(x) = \begin{cases} g(0) & \text{for } x = 0, \\ 0 & \text{otherwise.} \end{cases} \quad (18.30)$$

The resulting function $\bar{g}(x)$ consists of a single pulse at position 0 whose height corresponds to the original function value $g(0)$ (at position 0). Thus, by multiplying the function $g(x)$ by the impulse function, we obtain a single discrete sample value of $g(x)$ at position $x = 0$. If the impulse function $\delta(x)$ is shifted by a distance x_0 , we can sample $g(x)$ at an arbitrary position $x = x_0$,

$$\bar{g}(x) = g(x) \cdot \delta(x-x_0) = \begin{cases} g(x_0) & \text{for } x = x_0, \\ 0 & \text{otherwise.} \end{cases} \quad (18.31)$$

Here $\delta(x-x_0)$ is the impulse function shifted by x_0 , and the resulting function $\bar{g}(x)$ is zero except at position x_0 , where it contains the original function value $g(x_0)$. This relationship is illustrated in Fig. 18.5 for the sampling position $x_0 = 3$.

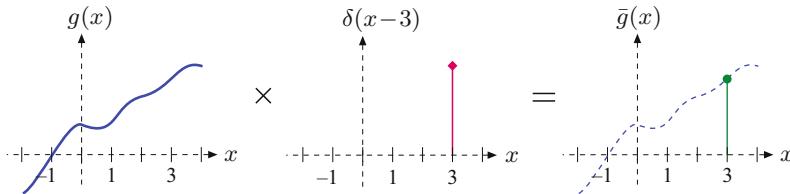


Fig. 18.5
Sampling with the impulse function. The continuous signal $g(x)$ is sampled at position $x_0 = 3$ by multiplying $g(x)$ by a shifted impulse function $\delta(x-3)$.

To sample the function $g(x)$ at more than one position simultaneously (e.g., at positions x_1 and x_2), we use two separately shifted versions of the impulse function, multiply $g(x)$ by both of them, and simply add the resulting function values. In this particular case, we get

$$\bar{g}(x) = g(x) \cdot \delta(x-x_1) + g(x) \cdot \delta(x-x_2) \quad (18.32)$$

$$= g(x) \cdot [\delta(x-x_1) + \delta(x-x_2)] \quad (18.33)$$

$$= \begin{cases} g(x_1) & \text{for } x = x_1, \\ g(x_2) & \text{for } x = x_2, \\ 0 & \text{otherwise.} \end{cases} \quad (18.34)$$

From Eqn. (18.33), sampling a continuous function $g(x)$ at N positions $x_i = 1, 2, \dots, N$ can thus be described as the sum of the N individual samples, that is,

$$\begin{aligned} \bar{g}(x) &= g(x) \cdot [\delta(x-1) + \delta(x-2) + \dots + \delta(x-N)] \\ &= g(x) \cdot \sum_{i=1}^N \delta(x-i). \end{aligned} \quad (18.35)$$

The comb function

The sum of shifted impulses $\sum_{i=1}^N \delta(x-i)$ in Eqn. (18.35) is called a *pulse sequence* or *pulse train*. Extending this sequence to infinity in both directions, we obtain the “comb” or “Shah” function

$$\text{III}(x) = \sum_{i=-\infty}^{\infty} \delta(x - i). \quad (18.36)$$

The process of discretizing a continuous function by taking samples at regular integral intervals can thus be written simply as

$$\bar{g}(x) = g(x) \cdot \text{III}(x), \quad (18.37)$$

that is, as a pointwise multiplication of the original signal $g(x)$ with the comb function $\text{III}(x)$. As Fig. 18.6 illustrates, the function values of $g(x)$ at integral positions $x_i \in \mathbb{Z}$ are transferred to the discrete function $\bar{g}(x_i)$ and ignored at all non-integer positions.

Of course, the sampling interval (i.e., the distance between adjacent samples) is not restricted to 1. To take samples at regular but *arbitrary* intervals τ , the sampling function $\text{III}(x)$ is simply scaled along the time or space axis; that is,

$$\bar{g}(x) = g(x) \cdot \text{III}\left(\frac{x}{\tau}\right), \quad \text{for } \tau > 0. \quad (18.38)$$

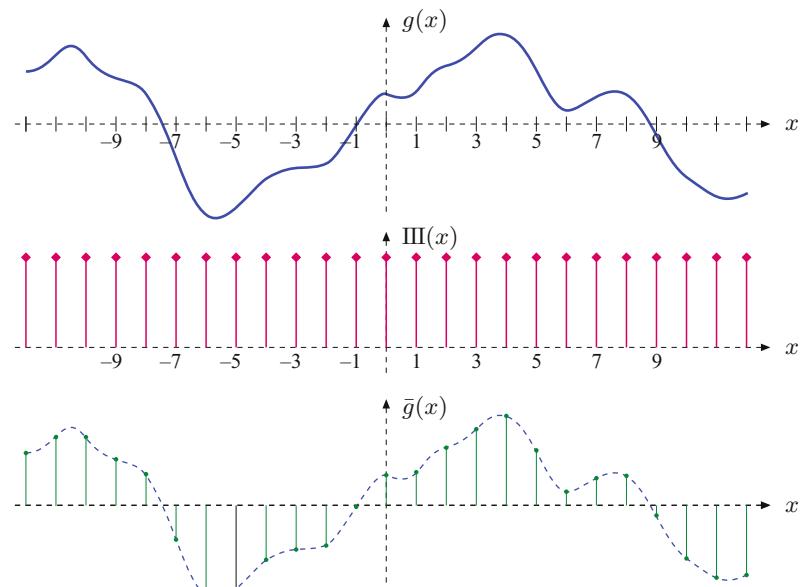
Effects of sampling in frequency space

Despite the elegant formulation made possible by the use of the comb function, one may still wonder why all this math is necessary to describe a process that appears intuitively to be so simple anyway. The Fourier spectrum gives one answer to this question. Sampling a continuous function has massive—though predictable—effects upon the frequency spectrum of the resulting (discrete) signal. Using the comb function as a formal model for the sampling process makes it relatively easy to estimate and interpret those spectral effects. Similar to the Gaussian (see Sec. 18.1.5), the comb function features the special property that its Fourier transform

$$\text{III}(x) \circledast \text{III}\left(\frac{1}{2\pi}\omega\right) \quad (18.39)$$

Fig. 18.6

Sampling with the comb function. The original continuous signal $g(x)$ is multiplied by the comb function $\text{III}(x)$. The function value $g(x)$ is transferred to the resulting function $\bar{g}(x)$ only at integral positions $x = x_i \in \mathbb{Z}$ and ignored at all non-integer positions.



is again a comb function (i.e., the same type of function). In general, the Fourier transform of a comb function scaled to an arbitrary sampling interval τ is

$$\text{III}\left(\frac{x}{\tau}\right) \circledast \tau \text{III}\left(\frac{\tau}{2\pi}\omega\right), \quad (18.40)$$

due to the similarity property of the Fourier transform (Eqn. (18.24)). Figure 18.7 shows two examples of the comb function $\text{III}_\tau(x)$ with sampling intervals $\tau = 1$ and $\tau = 3$ and the corresponding Fourier transforms.

Now, what happens to the Fourier spectrum during discretization, that is, when we multiply a function in signal space by the comb function $\text{III}\left(\frac{x}{\tau}\right)$? We get the answer by recalling the convolution property of the Fourier transform (Eqn. (18.26)): the product of two functions in one space (signal or frequency space) corresponds to the linear convolution of the transformed functions in the opposite space, and thus

$$g(x) \cdot \text{III}\left(\frac{x}{\tau}\right) \circledast G(\omega) * \tau \cdot \text{III}\left(\frac{\tau}{2\pi}\omega\right). \quad (18.41)$$

We already know that the Fourier spectrum of the sampling function is a comb function again and therefore consists of a sequence of regularly spaced pulses (Fig. 18.7). In addition, we know that convolving an arbitrary function with the impulse $\delta(x)$ returns the original function; that is, $f(x) * \delta(x) = f(x)$ (see Ch. 5, Sec. 5.3.4). Convolving with a *shifted* pulse $\delta(x-d)$ also reproduces the original function $f(x)$, though shifted by the same distance d :

$$f(x) * \delta(x-d) = f(x-d). \quad (18.42)$$

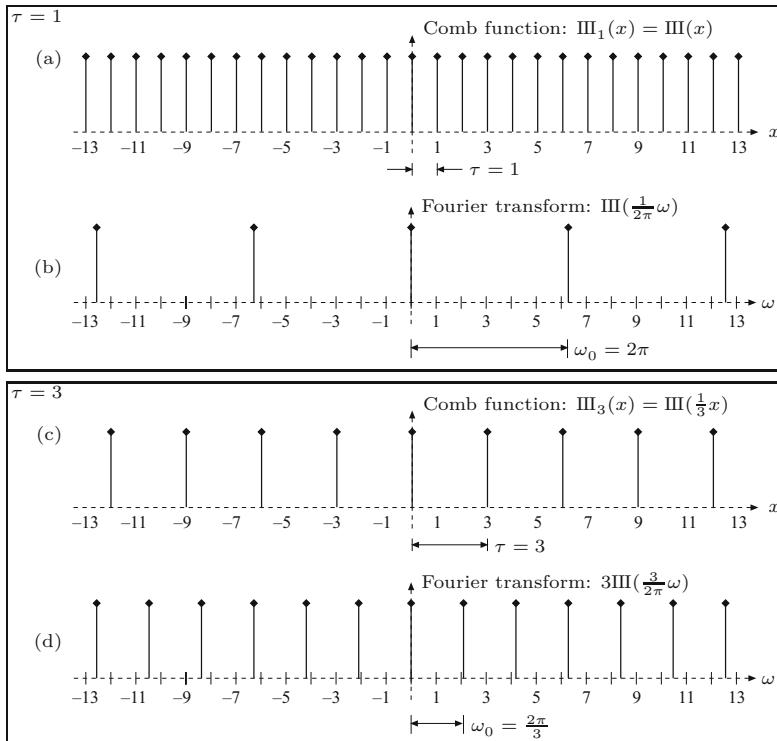
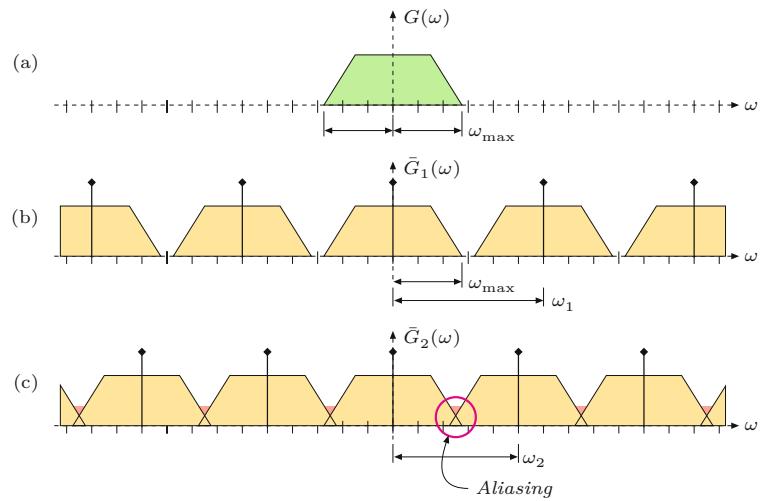


Fig. 18.7

Comb function and its Fourier transform. Comb function $\text{III}_\tau(x)$ for the sampling interval $\tau = 1$ (a) and its Fourier transform (b). Comb function for $\tau = 3$ (c) and its Fourier transform (d). Note that the actual height of the δ -pulses is undefined and shown only for illustration.

Fig. 18.8

Spectral effects of sampling. The spectrum $G(\omega)$ of the original continuous signal is assumed to be band-limited within the range $\pm\omega_{\max}$ (a). Sampling the signal at a rate (sampling frequency) $\omega_s = \omega_1$ causes the signal's spectrum $G(\omega)$ to be replicated at multiples of ω_1 along the frequency (ω) axis (b). Obviously, the replicas in the spectrum do not overlap as long as $\omega_s > 2\omega_{\max}$. In (c), the sampling frequency $\omega_s = \omega_2$ is less than $2\omega_{\max}$, so there is overlap between the replicas in the spectrum, and frequency components are mirrored at $2\omega_{\max}$ and superimpose the original spectrum. This effect is called “aliasing” because the original spectrum (and thus the original signal) cannot be reproduced from such a corrupted spectrum.



As a consequence, the spectrum $G(\omega)$ of the original continuous signal becomes *replicated* in the Fourier spectrum $\bar{G}(\omega)$ of a sampled signal at every pulse of the sampling function's spectrum; that is, infinitely many times (see Fig. 18.8(a, b))! Thus the resulting Fourier spectrum is repetitive with a period $\frac{2\pi}{\tau}$, which corresponds to the sampling frequency ω_s .

Aliasing and the sampling theorem

As long as the spectral replicas in $\bar{G}(\omega)$ created by the sampling process do not overlap, the original spectrum $G(\omega)$ —and thus the original continuous function—can be reconstructed without loss from any isolated replica of $G(\omega)$ in the periodic spectrum $\bar{G}(\omega)$. As we can see in Fig. 18.8, this requires that the frequencies contained in the original signal $g(x)$ be within some upper limit ω_{\max} ; that is, the signal contains no components with frequencies greater than ω_{\max} . The maximum allowed signal frequency ω_{\max} depends upon the sampling frequency ω_s used to discretize the signal, with the requirement

$$\omega_{\max} \leq \frac{1}{2} \cdot \omega_s \quad \text{or} \quad \omega_s \geq 2 \cdot \omega_{\max}. \quad (18.43)$$

Discretizing a continuous signal $g(x)$ with frequency components in the range $0 \leq \omega \leq \omega_{\max}$ thus requires a sampling frequency ω_s of at least twice the maximum signal frequency ω_{\max} . If this condition is not met, the replicas in the spectrum of the sampled signal overlap (Fig. 18.8(c)) and the spectrum becomes corrupted. Consequently, the original signal cannot be recovered flawlessly from the sampled signal's spectrum. This effect is commonly called “aliasing”.

What we just said in simple terms is nothing but the essence of the famous “sampling theorem” formulated by Shannon and Nyquist (see, e.g., [43, p. 256]). It actually states that the sampling frequency must be at least twice the *bandwidth*⁸ of the continuous signal to avoid aliasing effects. However, if we assume that a signal's frequency range

⁸ This may be surprising at first because it allows a signal with high frequency—but low bandwidth—to be sampled (and correctly recon-

starts at zero, then bandwidth and maximum frequency are the same anyway.

18.3 THE DISCRETE FOURIER TRANSFORM (DFT)

18.2.2 Discrete and Periodic Functions

Assume that we are given a continuous signal $g(x)$ that is periodic with a period of length T . In this case, the corresponding Fourier spectrum $G(\omega)$ is a sequence of thin spectral lines equally spaced at a distance $\omega_0 = 2\pi/T$. As discussed in Sec. 18.1.2, the Fourier spectrum of a periodic function can be represented as a Fourier series and is therefore *discrete*. Conversely, if a continuous signal $g(x)$ is *sampled* at regular intervals τ , then the corresponding Fourier spectrum becomes *periodic* with a period of length $\omega_s = 2\pi/\tau$.

Sampling in signal space thus leads to periodicity in frequency space and vice versa. [Figure 18.9](#) illustrates this relationship and the transition from a continuous nonperiodic signal to a discrete periodic function, which can be represented as a finite vector of numbers and thus easily processed on a computer.

Thus, in general, the Fourier spectrum of a continuous, nonperiodic signal $g(x)$ is also continuous and nonperiodic ([Fig. 18.9\(a,b\)](#)). However, if the signal $g(x)$ is *periodic*, then the corresponding spectrum is *discrete* ([Fig. 18.9\(c,d\)](#)). Conversely, a discrete—but not necessarily periodic—signal leads to a periodic spectrum ([Fig. 18.9\(e,f\)](#)). Finally, if a signal is discrete *and* periodic with M samples per period, then its spectrum is also discrete and periodic with M values ([Fig. 18.9\(g,h\)](#)). Note that the particular signals and spectra in [Fig. 18.9](#) were chosen for illustration only and do not really correspond with each other.

18.3 The Discrete Fourier Transform (DFT)

In the case of a discrete periodic signal, only a finite sequence of M sample values is required to completely represent either the signal $g(u)$ itself or its Fourier spectrum $G(m)$.⁹ This representation as finite vectors makes it straightforward to store and process signals and spectra on a computer. What we still need is a version of the Fourier transform applicable to discrete signals.

18.3.1 Definition of the DFT

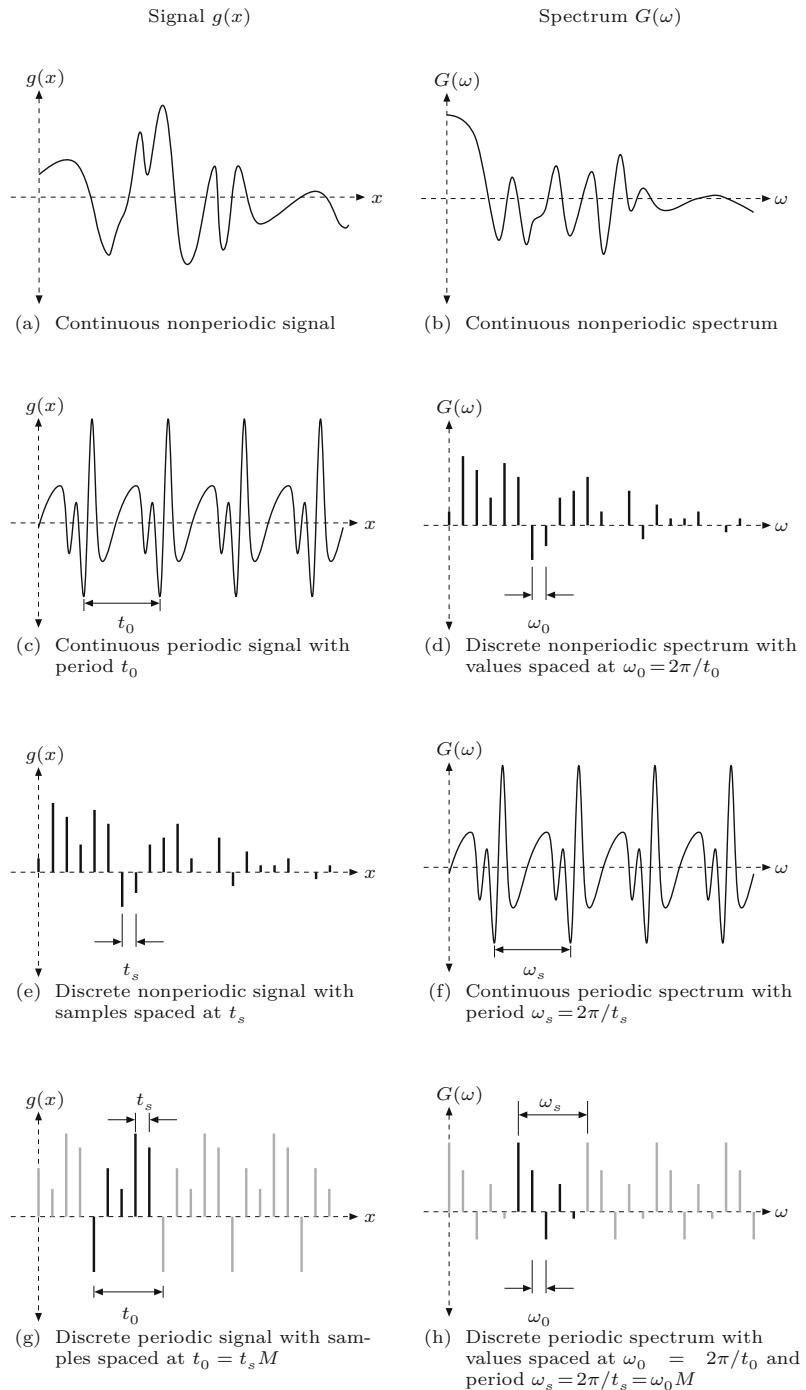
The discrete Fourier transform is, just like its continuous counterpart, identical in both directions. For a discrete signal $g(u)$ of length M ($u = 0 \dots M-1$), the forward transform (**DFT**) is defined as

structed) at a relatively low sampling frequency, even well below the maximum signal frequency. This is possible because one can also use a filter with suitably low bandwidth for reconstructing the original signal.

For example, it may be sufficient to strike (i.e., “sample”) a church bell (a low-bandwidth oscillatory system with small internal damping) to uniquely generate a sound wave of relatively high frequency.

⁹ Notation: We use $g(x)$, $G(\omega)$ for a *continuous* signal or spectrum, respectively, and $g(u)$, $G(m)$ for the *discrete* versions.

Fig. 18.9
Transition from continuous to discrete periodic functions (illustration only).



$$G(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) - i \cdot \sin\left(2\pi \frac{mu}{M}\right) \right] \quad (18.44)$$

$$= \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot e^{-i2\pi \frac{mu}{M}}, \quad (18.45)$$

for $0 \leq m < M$, and the *inverse* transform (DFT^{-1}) is¹⁰

$$g(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) + i \cdot \sin\left(2\pi \frac{mu}{M}\right) \right] \quad (18.46)$$

$$= \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot e^{i2\pi \frac{mu}{M}}, \quad (18.47)$$

for $0 \leq u < M$. Note that both the *signal* $g(u)$ and the discrete *spectrum* $G(m)$ are complex-valued vectors of length M , that is,

$$\begin{aligned} g(u) &= g_{\text{Re}}(u) + i \cdot g_{\text{Im}}(u), \\ G(m) &= G_{\text{Re}}(m) + i \cdot G_{\text{Im}}(m), \end{aligned} \quad (18.48)$$

for $u, m = 0, \dots, M-1$. A numerical example for a DFT with $M = 10$ is shown in Fig. 18.10. Converting Eqn. (18.44) from Euler's exponential notation (Eqn. (18.10)) we obtain the discrete Fourier spectrum in component notation as

$$G(m) = \frac{1}{\sqrt{M}} \cdot \sum_{u=0}^{M-1} \underbrace{\left[g_{\text{Re}}(u) + i \cdot g_{\text{Im}}(u) \right]}_{g(u)} \cdot \underbrace{\left[\cos\left(2\pi \frac{mu}{M}\right) - i \cdot \sin\left(2\pi \frac{mu}{M}\right) \right]}_{C_m^M(u) - i \cdot S_m^M(u)}, \quad (18.49)$$

where we denote as C_m^M and S_m^M the discrete (cosine and sine) basis functions, as described in the next section. Applying the usual complex multiplication,¹¹ we obtain the real and imaginary parts of the discrete Fourier spectrum as

$$G_{\text{Re}}(m) = \frac{1}{\sqrt{M}} \cdot \sum_{u=0}^{M-1} g_{\text{Re}}(u) \cdot C_m^M(u) + g_{\text{Im}}(u) \cdot S_m^M(u), \quad (18.50)$$

$$G_{\text{Im}}(m) = \frac{1}{\sqrt{M}} \cdot \sum_{u=0}^{M-1} g_{\text{Im}}(u) \cdot C_m^M(u) - g_{\text{Re}}(u) \cdot S_m^M(u), \quad (18.51)$$

for $m = 0, \dots, M-1$. Analogously, the *inverse* DFT in Eqn. (18.46) expands to

$$g_{\text{Re}}(u) = \frac{1}{\sqrt{M}} \cdot \sum_{m=0}^{M-1} G_{\text{Re}}(m) \cdot C_u^M(m) - G_{\text{Im}}(m) \cdot S_u^M(m), \quad (18.52)$$

$$g_{\text{Im}}(u) = \frac{1}{\sqrt{M}} \cdot \sum_{m=0}^{M-1} G_{\text{Im}}(m) \cdot C_u^M(m) + G_{\text{Re}}(m) \cdot S_u^M(m), \quad (18.53)$$

for $u = 0, \dots, M-1$.

18.3 THE DISCRETE FOURIER TRANSFORM (DFT)

¹⁰ Compare these definitions with the corresponding expressions for the *continuous* forward and inverse Fourier transforms in Eqns. (18.19) and (18.20), respectively.

¹¹ See also Sec. A.3 in the Appendix.

Fig. 18.10
Complex-valued result of the DFT for a signal of length $M = 10$ (example). In the discrete Fourier transform (DFT), both the original signal $g(u)$ and its spectrum $G(m)$ are complex-valued vectors of length M ; * indicates values with $|G(m)| < 10^{-15}$.

u	$g(u)$		$G(m)$		m
0	1.0000	0.0000	DFT	14.2302	0.0000
1	3.0000	0.0000		-5.6745	-2.9198
2	5.0000	0.0000		*0.0000	*0.0000
3	7.0000	0.0000		-0.0176	-0.6893
4	9.0000	0.0000		*0.0000	*0.0000
5	8.0000	0.0000		0.3162	0.0000
6	6.0000	0.0000		*0.0000	*0.0000
7	4.0000	0.0000		-0.0176	0.6893
8	2.0000	0.0000		*0.0000	*0.0000
9	0.0000	0.0000		-5.6745	2.9198
	Re	Im		Re	Im

18.3.2 Discrete Basis Functions

The inverse DFT (Eqn. (18.46)) performs the decomposition of the discrete function $g(u)$ into a finite sum of M discrete cosine and sine functions (\mathbf{C}_m^M , \mathbf{S}_m^M) whose weights (or “amplitudes”) are determined by the DFT coefficients in $G(m)$. Each of these 1D basis functions (first used in Eqn. (18.49)),

$$\mathbf{C}_m^M(u) = \mathbf{C}_u^M(m) = \cos\left(2\pi \frac{mu}{M}\right), \quad (18.54)$$

$$\mathbf{S}_m^M(u) = \mathbf{S}_u^M(m) = \sin\left(2\pi \frac{mu}{M}\right), \quad (18.55)$$

is periodic with M and has a discrete frequency (wave number) m , which corresponds to the angular frequency

$$\omega_m = 2\pi \cdot \frac{m}{M}. \quad (18.56)$$

For example, Figs. 18.11 and 18.12 show the discrete basis functions (with integer ordinate values $u \in \mathbb{Z}$) for the DFT of length $M = 8$ as well as their continuous counterparts (with ordinate values $x \in \mathbb{R}$).

For wave number $m = 0$, the cosine function $\mathbf{C}_0^M(u)$ (Eqn. (18.54)) has the constant value 1. The corresponding DFT coefficient $G_{\text{Re}}(0)$ —the real part of $G(0)$ —thus specifies the constant part of the signal or the average value of the signal $g(u)$ in Eqn. (18.52). In contrast, the zero-frequency sine function $\mathbf{S}_0^M(u)$ is zero for any value of u and thus cannot contribute anything to the signal. The corresponding DFT coefficients $G_{\text{Im}}(0)$ in Eqn. (18.52) and $G_{\text{Re}}(0)$ in Eqn. (18.53) are therefore of no relevance. For a real-valued signal (i.e., $g_{\text{Im}}(u) = 0$ for all u), the coefficient $G_{\text{Im}}(0)$ in the corresponding Fourier spectrum must also be zero.

As seen in Fig. 18.11, the wave number $m = 1$ relates to a cosine or sine function that performs exactly one full cycle over the signal length $M = 8$. Similarly, the wave numbers $m = 2, \dots, 7$ correspond to $2, \dots, 7$ complete cycles over the signal length M (see Figs. 18.11 and 18.12).

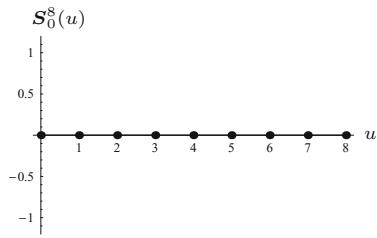
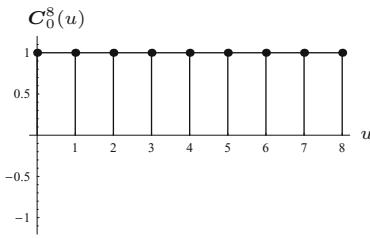
18.3.3 Aliasing Again!

A closer look at Figs. 18.11 and 18.12 reveals an interesting fact: the sampled (discrete) cosine and sine functions for $m = 3$ and $m = 5$ are *identical*, although their continuous counterparts are different! The same is true for the frequency pairs $m = 2, 6$ and $m = 1, 7$. What we

$$C_m^8(u) = \cos\left(\frac{2\pi m}{8}u\right)$$

$$S_m^8(u) = \sin\left(\frac{2\pi m}{8}u\right)$$

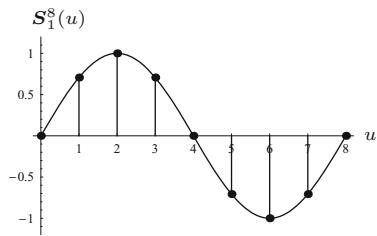
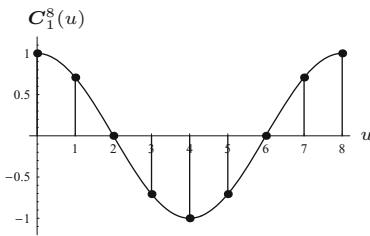
$m = 0$



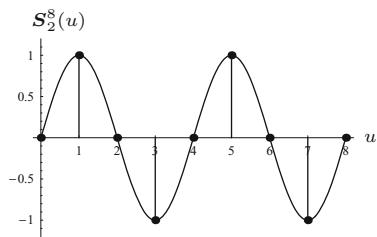
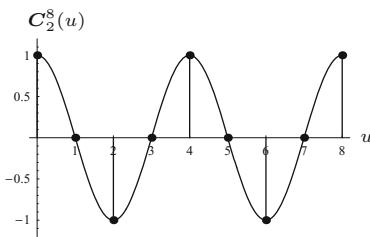
18.3 THE DISCRETE FOURIER TRANSFORM (DFT)

Fig. 18.11

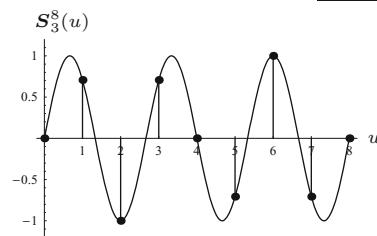
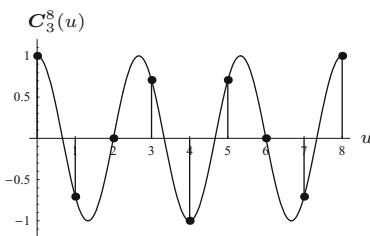
Discrete basis functions $C_m^M(u)$ and $S_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0, \dots, 3$. Each plot shows both the discrete function (round dots) and the corresponding continuous function.



$m = 1$



$m = 2$



$m = 3$

see here is another manifestation of the sampling theorem—which we had originally encountered (Sec. 18.2.1) in frequency space—in *signal space*. Obviously, $m = 4$ is the maximum frequency component that can be represented by a discrete signal of length $M = 8$. Any discrete function with a higher frequency ($m = 5, \dots, 7$ in this case) has an identical counterpart with a lower wave number and thus cannot be reconstructed from the sampled signal (see also Fig. 18.13)!

If a continuous signal is sampled at a regular distance τ , the corresponding Fourier spectrum is repeated at multiples of $\omega_s = 2\pi/\tau$,

$$C_m^8(u) = \cos\left(\frac{2\pi m}{8}u\right)$$

$$S_m^8(u) = \sin\left(\frac{2\pi m}{8}u\right)$$

$m = 4$

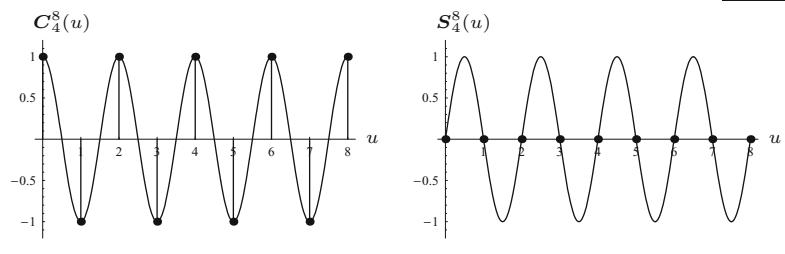
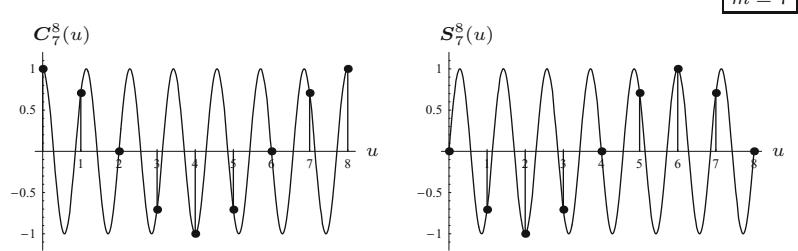
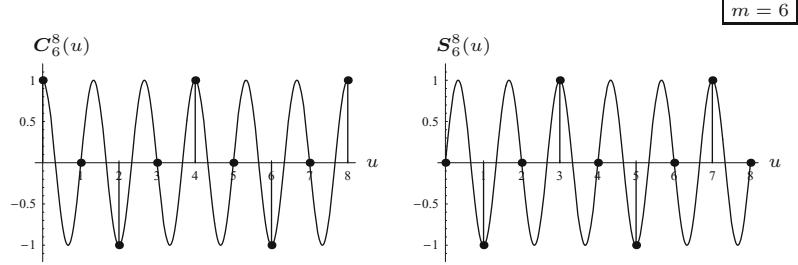
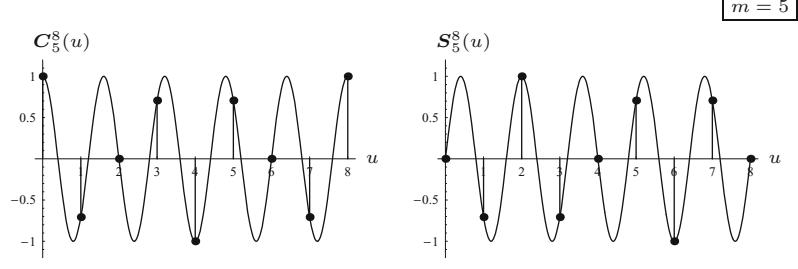


Fig. 18.12

Discrete basis functions (continued). Signal length $M = 8$ and wave numbers $m = 4, \dots, 7$. Notice that, for example, the discrete functions for $m = 5$ and $m = 3$ (Fig. 18.11) are identical because $m = 4$ is the maximum wave number that can be represented in a discrete spectrum of length $M = 8$.

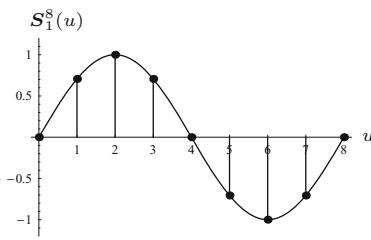
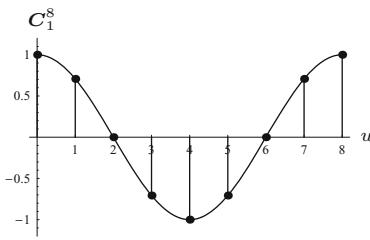


as we have shown earlier (Fig. 18.8). In the discrete case, the spectrum is periodic with length M . Since the Fourier spectrum of a real-valued signal is symmetric about the origin (Eqn. (18.21)), there is for every coefficient with wave number m an equal-sized duplicate with wave number $-m$. Thus the spectral components appear pairwise and mirrored at multiples of M ; that is,

$$C_m^8(u) = \cos\left(\frac{2\pi m}{8}u\right)$$

$$S_m^8(u) = \sin\left(\frac{2\pi m}{8}u\right)$$

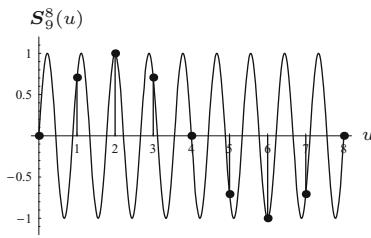
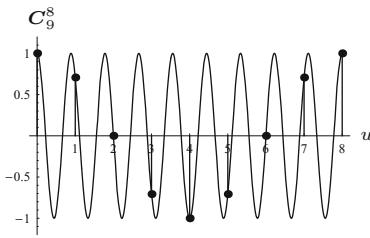
$m = 1$



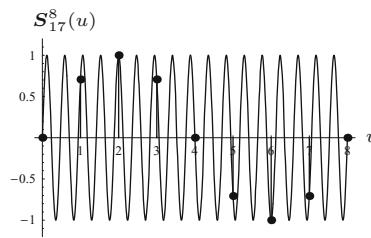
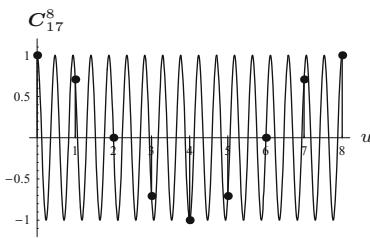
18.3 THE DISCRETE FOURIER TRANSFORM (DFT)

Fig. 18.13

Aliasing in signal space. For the signal length $M = 8$, the discrete cosine and sine basis functions for the wave numbers $m = 1, 9, 17, \dots$ (round dots) are all identical. The sampling frequency itself corresponds to the wave number $m = 8$.



$m = 9$



$m = 17$

$$\begin{aligned} |G(m)| &= |G(M-m)| = |G(M+m)| \\ &= |G(2M-m)| = |G(2M+m)| \\ &\dots \\ &= |G(kM-m)| = |G(kM+m)|, \end{aligned} \quad (18.57)$$

for all $k \in \mathbb{Z}$. If the original continuous signal contains “energy” at the frequencies

$$\omega_m > \omega_{M/2}$$

(i.e., signal components with wave numbers $m > M/2$), then, according to the sampling theorem, the overlapping parts of the spectra are superimposed in the resulting periodic spectrum of the discrete signal.

18.3.4 Units in Signal and Frequency Space

The relation between the units in signal and frequency space and the interpretation of wave numbers m is a common cause of confusion. While the discrete signal and its spectrum are simple numerical vectors and units of measurement are irrelevant for computing the DFT

itself, it is nevertheless important to understand how the coordinates in the spectrum relate to physical dimensions in the real world.

Clearly, every complex-valued spectral coefficient $G(m)$ corresponds to one pair of cosine and sine functions with a particular frequency in signal space. Assume a continuous signal is sampled at M consecutive positions spaced at τ (an interval in time or distance in space). The *wave number* $m = 1$ then corresponds to the *fundamental period* of the discrete signal (which is now assumed to be periodic) with a period of length $M\tau$; that is, to the *frequency*

$$f_1 = \frac{1}{M\tau}. \quad (18.58)$$

In general, the wave number m of a discrete spectrum relates to the physical frequency as

$$f_m = m \frac{1}{M\tau} = m \cdot f_1 \quad (18.59)$$

for $0 \leq m < M$, which is equivalent to the angular frequency

$$\omega_m = 2\pi f_m = m \frac{2\pi}{M\tau} = m \cdot \omega_1. \quad (18.60)$$

Obviously then, the sampling frequency $f_s = 1/\tau = M \cdot f_1$ corresponds to the wave number $m_s = M$. As expected, the maximum nonaliased wave number in the spectrum is

$$m_{\max} = \frac{M}{2} = \frac{m_s}{2}, \quad (18.61)$$

that is, half the sampling frequency index m_s .

Example 1: time-domain signal

We assume for this example that $g(u)$ is a signal in the time domain (e.g., a discrete sound signal) that contains $M = 500$ sample values taken at regular intervals $\tau = 1 \text{ ms} = 10^{-3} \text{ s}$. Thus the sampling frequency is $f_s = 1/\tau = 1000 \text{ Hertz}$ (cycles per second) and the total duration (fundamental period) of the signal is $M\tau = 0.5 \text{ s}$.

The signal is implicitly periodic, and from Eqn. (18.58) we obtain its fundamental frequency as $f_1 = \frac{1}{500 \cdot 10^{-3}} = \frac{1}{0.5} = 2 \text{ Hertz}$. The wave number $m = 2$ in this case corresponds to a real frequency $f_2 = 2f_1 = 4 \text{ Hertz}$, $f_3 = 6 \text{ Hertz}$, etc. The maximum frequency that can be represented by this discrete signal without aliasing is $f_{\max} = \frac{M}{2} f_1 = \frac{1}{2\tau} = 500 \text{ Hertz}$, exactly half the sampling frequency f_s .

Example 2: space-domain signal

Assume we have a 1D print pattern with a resolution (i.e., spatial sampling frequency) of 120 dots per cm, which equals approximately 300 dots per inch (dpi) and a total signal length of $M = 1800$ samples. This corresponds to a spatial sampling interval of $\tau = 1/120 \text{ cm} \approx 83 \mu\text{m}$ and a physical signal length of $(1800/120) \text{ cm} = 15 \text{ cm}$.

The fundamental frequency of this signal (again implicitly assumed to be periodic) is $f_1 = \frac{1}{15}$, expressed in cycles per cm. The sampling frequency is $f_s = 120$ cycles per cm and thus the maximum signal frequency is $f_{\max} = \frac{f_s}{2} = 60$ cycles per cm. The maximum signal frequency specifies the finest structure ($\frac{1}{60} \text{ cm}$) that can be reproduced by this print raster.

18.3.5 Power Spectrum

The *magnitude* of the complex-valued Fourier spectrum,

$$|G(m)| = \sqrt{G_{\text{Re}}^2(m) + G_{\text{Im}}^2(m)}, \quad (18.62)$$

is commonly called the “power spectrum” of a signal. It specifies the energy that individual frequency components in the spectrum contribute to the signal. The power spectrum is real-valued and positive and thus often used for graphically displaying the results of Fourier transforms (see also Ch. 19, Sec. 19.2).

Since all phase information is lost in the power spectrum, the original signal cannot be reconstructed from the power spectrum alone. However, because of the missing phase information, the power spectrum is insensitive to shifts of the original signal and can thus be efficiently used for comparing signals. To be more precise, the power spectrum of a circularly shifted signal is identical to the power spectrum of the original signal. Thus, given a discrete periodic signal $g_1(u)$ of length M and a second signal $g_2(u)$ shifted by some offset d , such that

$$g_2(u) = g_1(u-d) \quad (18.63)$$

the corresponding power spectra are the same, that is,

$$|G_2(m)| = |G_1(m)|, \quad (18.64)$$

although in general the complex-valued spectra $G_1(m)$ and $G_2(m)$ are different. Furthermore, from the symmetry property of the Fourier spectrum, it follows that

$$|G(m)| = |G(-m)|, \quad (18.65)$$

for real-valued signals $g(u) \in \mathbb{R}$.

18.4 Implementing the DFT

18.4.1 Direct Implementation

Based on the definitions in Eqns. (18.50) and (18.51) the DFT can be directly implemented, as shown in Prog. 18.1. The main method `DFT()` transforms a signal vector of arbitrary length M (not necessarily a power of 2). It requires roughly M^2 operations (multiplications and additions); that is, the time complexity of this DFT algorithm is $\mathcal{O}(M^2)$.

One way to improve the efficiency of the DFT algorithm is to use lookup tables for the sin and cos functions (which are relatively “expensive” to compute) since only function values for a set of M different angles φ_m are ever needed. The angles $\varphi_m = 2\pi \frac{m}{M}$ corresponding to $m = 0, \dots, M - 1$ are evenly distributed over the full 360° circle. Any integral multiple $\varphi_m \cdot u$ (for $u \in \mathbb{Z}$) can only fall onto one of these angles again because

18.4 IMPLEMENTING THE DFT

Prog. 18.1

Direct implementation of the DFT based on the definition in Eqns. (18.50) and (18.51). The method `DFT()` returns a complex-valued vector with the same length as the complex-valued input (signal) vector `g`. This method implements both the forward and the inverse transforms, controlled by the Boolean parameter `forward`. The class `Complex` (bottom) defines the structure of the complex-valued vector elements.

```

1  class Complex {
2      double re, im;
3      Complex(double re, double im) { //constructor method
4          this.re = re;
5          this.im = im;
6      }
7 }

8 Complex[] DFT(Complex[] g, boolean forward) {
9     int M = g.length;
10    double s = 1 / Math.sqrt(M); //common scale factor
11    Complex[] G = new Complex[M];
12    for (int m = 0; m < M; m++) {
13        double sumRe = 0;
14        double sumIm = 0;
15        double phim = 2 * Math.PI * m / M;
16        for (int u = 0; u < M; u++) {
17            double gRe = g[u].re;
18            double gIm = g[u].im;
19            double cosw = Math.cos(phim * u);
20            double sinw = Math.sin(phim * u);
21            if (!forward) // inverse transform
22                sinw = -sinw;
23            // complex multiplication: [gRe+i·gIm]·[cos(ω)+i·sin(ω)]
24            sumRe += gRe * cosw + gIm * sinw;
25            sumIm += gIm * cosw - gRe * sinw;
26        }
27        G[m] = new Complex(s * sumRe, s * sumIm);
28    }
29    return G;
30 }
```

$$\varphi_m \cdot u = 2\pi \frac{mu}{M} \equiv \underbrace{\frac{2\pi}{M} \cdot (mu \bmod M)}_{0 \leq k < M} = 2\pi \frac{k}{M} = \varphi_k, \quad (18.66)$$

where `mod` denotes the “modulus” operator.¹² Thus we can set up two constant tables (floating-point arrays) W_C and W_S of size M with the values

$$W_C(k) \leftarrow \cos(\omega_k) = \cos\left(2\pi \frac{k}{M}\right), \quad (18.67)$$

$$W_S(k) \leftarrow \sin(\omega_k) = \sin\left(2\pi \frac{k}{M}\right), \quad (18.68)$$

for $0 \leq k < M$. For computing the DFT, the necessary cosine and sine values (Eqn. (18.49)) can be read from these tables as

$$C_k^M(u) = \cos\left(2\pi \frac{mu}{M}\right) \equiv W_C(mu \bmod M), \quad (18.69)$$

$$S_k^M(u) = \sin\left(2\pi \frac{mu}{M}\right) \equiv W_S(mu \bmod M), \quad (18.70)$$

for arbitrary values of $m, u \in \mathbb{Z}$, without any additional computation. The necessary modification of the `DFT()` method in Prog. 18.1 is left as an exercise (Exercise 18.5).

Despite this significant improvement, the direct implementation of the DFT remains computationally intensive. As a matter of fact,

¹² See also Sec. F.1.2 in the Appendix.

it has been impossible for a long time to compute this form of DFT in sufficiently short time on off-the-shelf computers, and this is still true today for many real applications.

18.5 EXERCISES

18.4.2 Fast Fourier Transform (FFT)

Fortunately, for computing the DFT in practice, fast algorithms exist that lay out the sequence of computations in such a way that intermediate results are only computed once and optimally reused many times. This “fast Fourier transform”, which exists in many variations, generally reduces the time complexity of the computation from $\mathcal{O}(M^2)$ to $\mathcal{O}(M \log_2 M)$. The benefits are substantial, in particular for longer signals. For example, with a signal of length $M = 10^3$, the FFT leads to a speedup by a factor of 100 over the direct DFT implementation and an impressive gain of 10,000 times for a signal of length $M = 10^6$. Since its invention, the FFT has therefore become an indispensable tool in almost any application of spectral signal analysis [34].

Most FFT algorithms, including the one described in the famous publication by Cooley and Tukey in 1965 (see [88, p. 156] for a historic overview), are designed for signals of length $M = 2^k$ (i.e., powers of 2). However, FFT algorithms have also been developed for other lengths, including several small prime numbers [25]. Efficient Java implementations are available, for example, as part of the *JTransform* library¹³ by Piotr Wendykier [255] or the *Apache Commons Math* library¹⁴.

It is important to remember, though, that the DFT and FFT compute exactly the *same* result and the FFT is only a special—though ingenious—method for *implementing* the discrete Fourier transform (Eqn. (18.44)).

18.5 Exercises

Exercise 18.1. Calculate the values of the cosine function $f(x) = \cos(\omega x)$ with angular frequency $\omega = 5$ for the positions $x = -3, -2, \dots, 2, 3$. What is the length of this function’s period?

Exercise 18.2. Determine the phase angle φ of the function $f(x) = A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$ for $A = -1$ and $B = 2$.

Exercise 18.3. Calculate the real part, the imaginary part, and the magnitude of the complex value $z = 1.5 \cdot e^{-i2.5}$.

Exercise 18.4. A 1D optical scanner for sampling film transparencies is supposed to resolve image structures with a precision of 4,000 dpi. What spatial distance (in mm) between samples is required such that no aliasing occurs?

Exercise 18.5. Modify the direct implementation of the 1D DFT given in Prog. 18.1 by using lookup tables for the cos and sin functions as described in Eqns. (18.69)–(18.70).

¹³ <http://sites.google.com/site/piotrwendykier/software/jtransforms>.

¹⁴ <http://commons.apache.org/math/> (class `FastFourierTransformer`).

The Discrete Fourier Transform in 2D

The Fourier transform is defined not only for 1D signals but for functions of arbitrary dimension. Thus, 2D images are nothing special from a mathematical point of view.

19.1 Definition of the 2D DFT

For a 2D, periodic function (e.g., an intensity image) $g(u, v)$ of size $M \times N$, the discrete Fourier transform (2D DFT) is defined as

$$G(m, n) = \frac{1}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u, v) \cdot e^{-i2\pi \frac{mu}{M}} \cdot e^{-i2\pi \frac{nv}{N}} \quad (19.1)$$

$$= \frac{1}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u, v) \cdot e^{-i2\pi(\frac{mu}{M} + \frac{nv}{N})}, \quad (19.2)$$

for the spectral coordinates $m = 0, \dots, M-1$ and $n = 0, \dots, N-1$. As we see, the resulting Fourier transform is again a 2D function of the same size ($M \times N$) as the original signal. Similarly, the *inverse* 2D DFT is defined as

$$g(u, v) = \frac{1}{\sqrt{MN}} \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m, n) \cdot e^{i2\pi \frac{mu}{M}} \cdot e^{i2\pi \frac{nv}{N}} \quad (19.3)$$

$$= \frac{1}{\sqrt{MN}} \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m, n) \cdot e^{i2\pi(\frac{mu}{M} + \frac{nv}{N})}, \quad (19.4)$$

for the image coordinates $u = 0, \dots, M-1$ and $v = 0, \dots, N-1$.

19.1.1 2D Basis Functions

Equation (19.4) shows that a discrete 2D, periodic function $g(u, v)$ can be represented as a linear combination (i.e., as a weighted sum) of 2D sinusoids of the form

$$e^{i \cdot 2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right)} = e^{i \cdot (\omega_m u + \omega_n v)} \quad (19.5)$$

$$= \underbrace{\cos \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right]}_{C_{m,n}^{M,N}(u,v)} + i \cdot \underbrace{\sin \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right]}_{S_{m,n}^{M,N}(u,v)}. \quad (19.6)$$

$C_{m,n}^{M,N}(u,v)$ and $S_{m,n}^{M,N}(u,v)$ are discrete, 2D cosine and sine functions with horizontal and vertical wave numbers n and m , respectively, and the corresponding angular frequencies ω_m , ω_n , that is,

$$C_{m,n}^{M,N}(u,v) = \cos \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right] = \cos(\omega_m u + \omega_n v), \quad (19.7)$$

$$S_{m,n}^{M,N}(u,v) = \sin \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right] = \sin(\omega_m u + \omega_n v). \quad (19.8)$$

Each of these basis functions is periodic with M units in the horizontal direction and N units in the vertical direction.

Examples

Figures 19.1 and 19.2 show a set of 2D cosine functions $C_{m,n}^{M,N}$ of size $M \times N = 16 \times 16$ for various combinations of wave numbers $m, n = 0, \dots, 3$. As we can clearly see, these functions correspond to a directed, cosine-shaped waveform whose orientation is determined by the wave numbers m and n . For example, the wave numbers $m = n = 2$ specify a cosine function $C_{2,2}^{M,N}(u,v)$ that performs two full cycles in both the horizontal and vertical directions, thus creating a diagonally oriented, 2D wave. Of course, the same holds for the corresponding sine functions.

19.1.2 Implementing the 2D DFT

As in the 1D case, we could directly use the definition in Eqn. (19.2) to write a program or procedure that implements the 2D DFT. However, this is not even necessary. A minor rearrangement of Eqn. (19.2) into

$$G(m, n) = \frac{1}{\sqrt{N}} \cdot \sum_{v=0}^{N-1} \underbrace{\left[\frac{1}{\sqrt{M}} \cdot \sum_{u=0}^{M-1} g(u, v) \cdot e^{-i2\pi \frac{um}{M}} \right]}_{\text{1-dim. DFT of row } g(\cdot, v)} \cdot e^{-i2\pi \frac{vn}{N}} \quad (19.9)$$

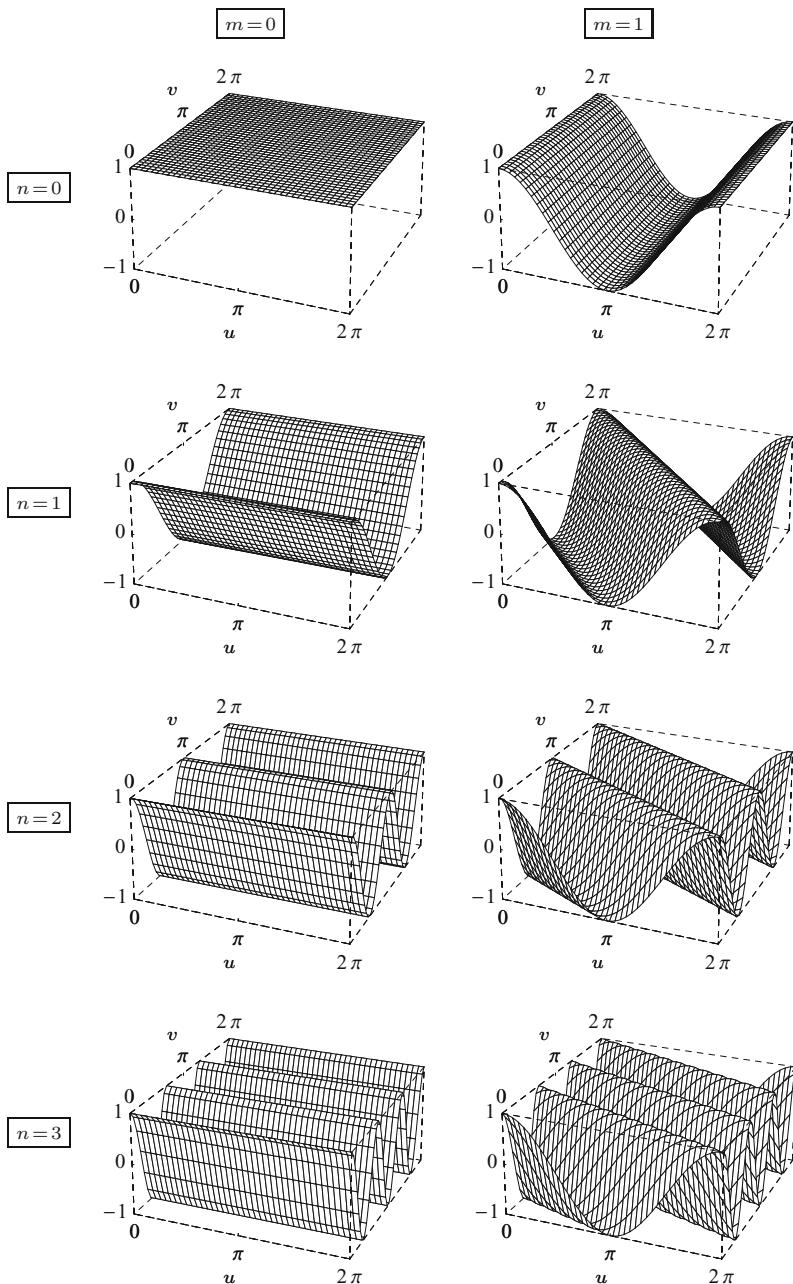
shows that its core contains a *1D DFT* (see Eqn. (18.44)) of the v th row vector $g(\cdot, v)$ that is independent of the “vertical” position v and size N (noting the fact that v and N are placed outside the square brackets in Eqn. (19.9)). If, in a first step, we *replace* each *row* vector $g(\cdot, v)$ of the original image by its 1D Fourier transform,

$$g_x(\cdot, v) \leftarrow \text{DFT}(g(\cdot, v)) \quad \text{for } 0 \leq v < N, \quad (19.10)$$

then we only need to replace each resulting *column* vector by its 1D DFT in a second step:

$$g_{xy}(u, \cdot) \leftarrow \text{DFT}(g_x(u, \cdot)) \quad \text{for } 0 \leq u < M. \quad (19.11)$$

The resulting function $g''(u, v)$ is precisely the 2D Fourier transform $G(m, n)$. Thus the *2D DFT* can be separated into a sequence of 1D



19.1 DEFINITION OF THE 2D DFT

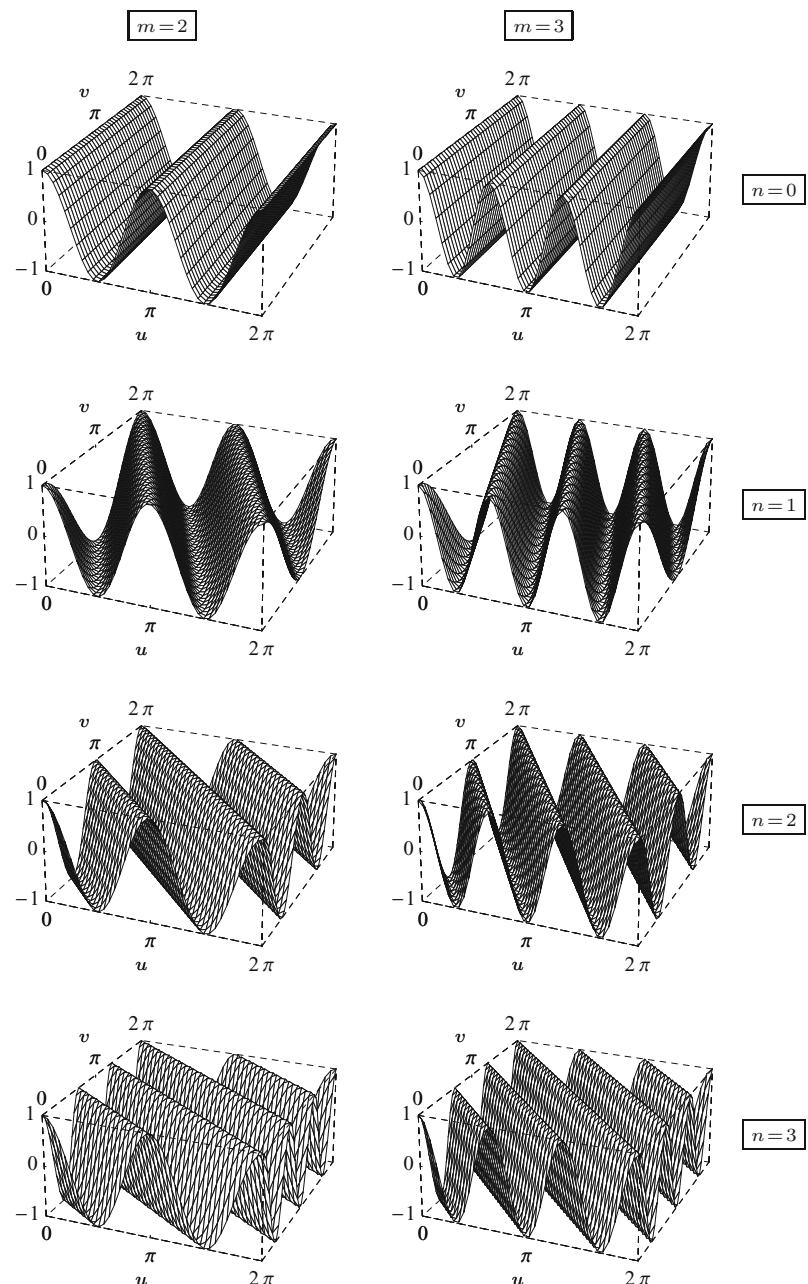
Fig. 19.1
2D cosine functions.
 $C_{m,n}^{M,N}(u, v) = \cos\left[2\pi\left(\frac{mu}{M} + \frac{nv}{N}\right)\right]$ for
 $M = N = 16$, $n = 0, \dots, 3$, $m = 0, 1$.

DFTs over the row and column vectors, respectively, as summarized in Alg. 19.1. The advantage of this is twofold: first, the 2D-DFT can be implemented more efficiently, and second, only a 1D implementation of the DFT (or the 1D FFT, as described in Ch. 18, Sec. 18.4.2) is needed to implement any multidimensional DFT.

As we can see from Eqn. (19.9), the 2D DFT could equally be performed in the opposite way, that is, by first doing a 1D DFT on all *rows* and subsequently on all *columns*. One should also note that all operations in Alg. 19.1 are done “in place”, which means that

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.2
2D cosine functions (*continued*). $C_{m,n}^{M,N}(u, v) = \cos\left[2\pi\left(\frac{um}{M} + \frac{vn}{N}\right)\right]$ for $M = N = 16$, $n = 0, \dots, 3$, $m = 2, 3$.



the original signal $g(u, v)$ is destructively modified and successively replaced by its Fourier transform $G(m, n)$ of the same size, without allocating any additional storage space. This feature is certainly desirable and also quite common, based on the fact that most 1D FFT algorithms (which should be used for implementing the DFT in practice) work “in place”.

```

1: Separable2dDft( $g$ ) ▷  $g(u, v) \in \mathbb{C}$ 
   Input:  $g$ , a 2D, discrete signal of size  $M \times N$ , with  $g(u, v) \in \mathbb{C}$ . Returns the DFT for the 2D function  $g(u, v)$ . The resulting spectrum  $G(m, n)$  has the same dimensions as  $g$ . The algorithm works “in place”, i.e.,  $g$  is modified.
2:  $(M, N) \leftarrow \text{Size}(g)$ 
3: for  $v \leftarrow 0, \dots, N - 1$  do
4:    $\mathbf{r} \leftarrow g(\cdot, v)$  ▷ extract the  $v$ th row vector of  $g$ 
5:    $g(\cdot, v) \leftarrow \text{DFT}(\mathbf{r})$  ▷ replace the  $v$ th row vector of  $g$ 
6: for  $u \leftarrow 0, \dots, M - 1$  do
7:    $\mathbf{c} \leftarrow g(u, \cdot)$  ▷ extract the  $u$ th column vector of  $g$ 
8:    $g(u, \cdot) \leftarrow \text{DFT}(\mathbf{c})$  ▷ replace the  $u$ th column vector of  $g$ 
9: return  $g$ 

```

Remark: $g(u, v) \equiv G(m, n)$ now contains the discrete 2D Fourier spectrum.

19.2 VISUALIZING THE 2D FOURIER TRANSFORM

Alg. 19.1

In-place computation of the 2D DFT as a sequence of 1D DFTs on row and column vectors.

19.2 Visualizing the 2D Fourier Transform

Unfortunately, there is no simple method for visualizing 2D complex-valued functions, such as the result of a 2D DFT. One alternative is to display the real and imaginary parts individually as 2D surfaces. Mostly, however, the absolute value of the complex functions is displayed, which in the case of the Fourier transform is $|G(m, n)|$, the *power spectrum* (see Ch. 18, Sec. 18.3.5).

19.2.1 Range of Spectral Values

For most natural images, the “spectral energy” concentrates at the lower frequencies with a clear maximum at wave numbers $(0, 0)$; that is, at the co-ordinate center (see also Sec. 19.4). The values of the power spectrum usually cover a wide range, and displaying them linearly often makes the smaller values invisible. To show the full range of spectral values, in particular the smaller values for the high frequencies, it is common to display the square root or the logarithm of the power spectrum, $\sqrt{|G(m, n)|}$ or $\log |G(m, n)|$, respectively.

19.2.2 Centered Representation of the DFT Spectrum

Analogous to the 1D case, the 2D spectrum is a periodic function in both dimensions,

$$G(m, n) = G(m + pM, n + qN), \quad (19.12)$$

for arbitrary $p, q \in \mathbb{Z}$. In the case of a real-valued 2D signal $g(u, v) \in \mathbb{R}$ (see Eqn. (18.57)), the power spectrum is also *symmetric* about the origin, that is,

$$|G(m, n)| = |G(-m, -n)|. \quad (19.13)$$

It is thus common to use a centered representation of the spectrum with coordinates m, n in the ranges

$$-\lfloor \frac{M}{2} \rfloor \leq m \leq \lfloor \frac{M-1}{2} \rfloor \quad \text{and} \quad -\lfloor \frac{N}{2} \rfloor \leq n \leq \lfloor \frac{N-1}{2} \rfloor,$$

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.3

Centering the 2D Fourier spectrum. In the original (noncentered) spectrum, the coordinate center (i.e., the region of low frequencies) is located in the upper left corner and, due to the periodicity of the spectrum, also at all other corners (a). In this case, the coefficients for the highest wave numbers (frequencies) lie at the center. Swapping the quadrants pairwise, as shown in (b), moves all low-frequency coefficients to the center and high frequencies to the periphery. A real 2D power spectrum is shown in its original form in (c) and in centered form in (d).

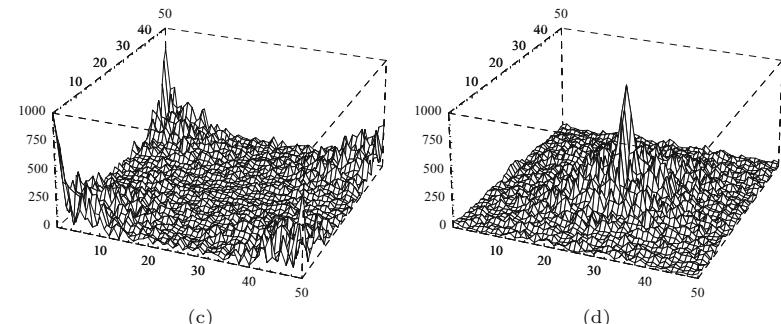
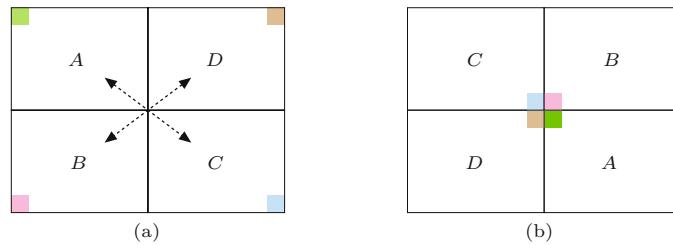
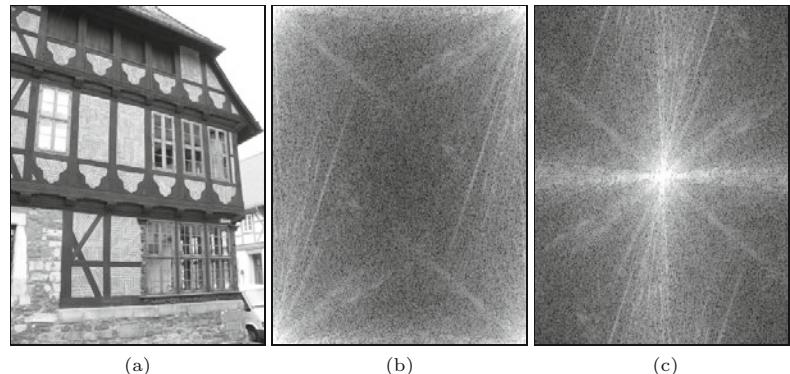


Fig. 19.4

Intensity plot of a 2D power spectrum: original image (a), noncentered spectrum (b), and centered spectrum (c).

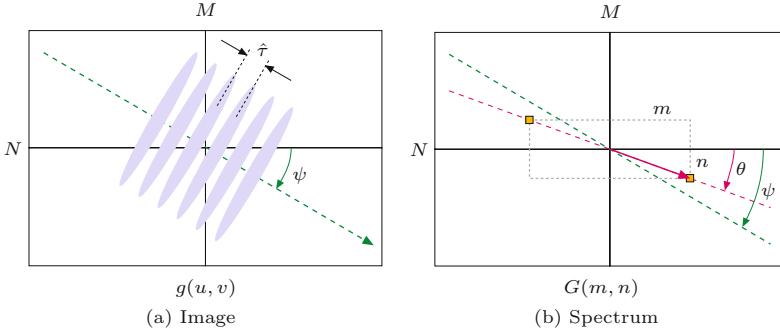


respectively. This can be easily accomplished by swapping the four quadrants of the Fourier transform, as illustrated in Fig. 19.3. In the resulting representation, the low-frequency coefficients are found at the center and the high-frequency entries along the outer boundaries. Figure 19.4 shows the plot of a 2D power spectrum as an intensity image in its original and centered form, with the intensity proportional to the logarithm of the spectral values ($\log_{10} |G(m, n)|$).

19.3 Frequencies and Orientation in 2D

19.3.1 Effective Frequency

As we could see in Figs. 19.1 and 19.2, each 2D basis function is an oriented cosine or sine function whose orientation and frequency are determined by its wave numbers m and n for the horizontal and vertical directions, respectively. If we moved along the main direction of such a basis function (i.e., perpendicular to the crest of the waves), we would follow a 1D cosine or sine function of some frequency f ,



which we call the *directional* or *effective frequency* of the waveform (see Fig. 19.5).

Recall that the wave numbers m, n specify how many full cycles the associated 2D basis function performs over a distance of M units in the horizontal direction or N units in the vertical direction. Thus, if an image of size $M \times N$ contains a periodic pattern with effective frequency $\hat{f} = 1/\hat{\tau}$ and orientation ψ , the associated frequency coefficients are found at positions

$$\binom{m}{n} = \pm \hat{f} \cdot \binom{M \cdot \cos(\psi)}{N \cdot \sin(\psi)} \quad (19.14)$$

in the corresponding 2D Fourier spectrum (see Fig. 19.5). Given the spectral position (m, n) , the effective frequency along the main direction of the wave can be derived (from the 1D case in Eqn. (18.58)) as

$$\hat{f}_{(m,n)} = \frac{1}{\tau} \cdot \sqrt{\left(\frac{m}{M}\right)^2 + \left(\frac{n}{N}\right)^2}, \quad (19.15)$$

where we assume the same spatial sampling interval along the x and y axes (i.e., $\tau = \tau_x = \tau_y$). Thus the *maximum signal frequency* in the directions of the coordinate axes is

$$\hat{f}_{(\pm \frac{M}{2}, 0)} = \hat{f}_{(0, \pm \frac{N}{2})} = \frac{1}{\tau} \cdot \sqrt{\left(\frac{1}{2}\right)^2} = \frac{1}{2\tau} = \frac{1}{2}f_s, \quad (19.16)$$

where $f_s = \frac{1}{\tau}$ denotes the sampling frequency. Notice that the effective signal frequency at the *corners* of the spectrum is

$$\hat{f}_{(\pm \frac{M}{2}, \pm \frac{N}{2})} = \frac{1}{\tau} \cdot \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \frac{1}{\sqrt{2}\tau} = \frac{1}{\sqrt{2}}f_s, \quad (19.17)$$

which is a factor $\sqrt{2}$ higher than along the coordinate axes (Eqn. (19.16)).

19.3.2 Frequency Limits and Aliasing in 2D

Figure 19.6 illustrates the relationship described in Eqns. (19.16) and (19.17). The highest permissible signal frequencies in any direction lie along the boundary of the centered 2D spectrum of size $M \times N$. Any signal with all frequency components *within* this region complies with the sampling theorem (Nyquist rule) and can thus be reconstructed without aliasing. In contrast, any spectral component

19.3 FREQUENCIES AND ORIENTATION IN 2D

Fig. 19.5

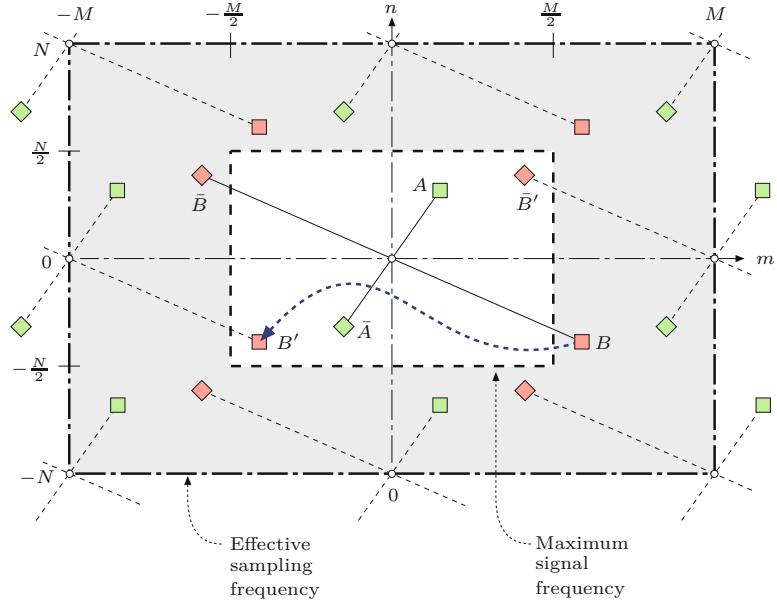
Frequency and orientation in 2D. The image (a) contains a periodic pattern with effective frequency $\hat{f} = 1/\hat{\tau}$ and orientation ψ . The frequency coefficient corresponding to this pattern is found at position $(m, n) = \pm \hat{f} \cdot (M \cos \psi, N \sin \psi)$ (see Eqn. (19.14)) in the 2D Fourier spectrum (b). Thus, if $M \neq N$, the spectral coefficients (m, n) are located at a direction (θ) different to the orientation of the image pattern (ψ).

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.6

Maximum signal frequencies and aliasing in 2D. The boundary of the $M \times N$ 2D spectrum (inner rectangle) marks the region of permissible signal frequencies along any direction.

The outer rectangle corresponds to the effective sampling frequency, which is twice the maximum signal frequency in the same direction. The signal component at spectral position a lies inside the permissible frequency range and thus causes no aliasing. In contrast, component b is outside the permissible range. Due to the periodicity of the spectrum, all components repeat (as in the 1D case) at all multiples of the sampling frequency along the m and n axis. This causes the component B to be “aliased” to a lower-frequency position B' (and \bar{B} to \bar{B}') in the visible part of the spectrum. Note that this also changes the *direction* of the corresponding wave in signal space.



outside these limits is reflected across the boundary of this box toward the coordinate center onto lower frequencies, which would appear as visual aliasing in the reconstructed image.

Apparently the lowest effective sampling frequency (Eqn. (19.15)) occurs in the directions of the coordinate axes of the sampling grid. To ensure that a certain image pattern can be reconstructed without aliasing at any orientation, the effective signal frequency \hat{f} of that pattern must be limited to $\frac{f_s}{2} = \frac{1}{2\tau}$ in every direction, again assuming that the sampling interval τ is the same along both coordinate axes.

19.3.3 Orientation

The spatial orientation of a 2D cosine or sine wave with spectral coordinates m, n (wave numbers $0 \leq m < M, 0 \leq n < N$) is

$$\psi_{(m,n)} = \text{ArcTan}\left(\frac{m}{M}, \frac{n}{N}\right) = \text{ArcTan}(mN, nM), \quad (19.18)$$

where $\psi_{(m,n)}$ for $m = n = 0$ is of course undefined.¹ Conversely, a 2D sinusoid with effective frequency \hat{f} and spatial orientation ψ is represented by the spectral coordinates

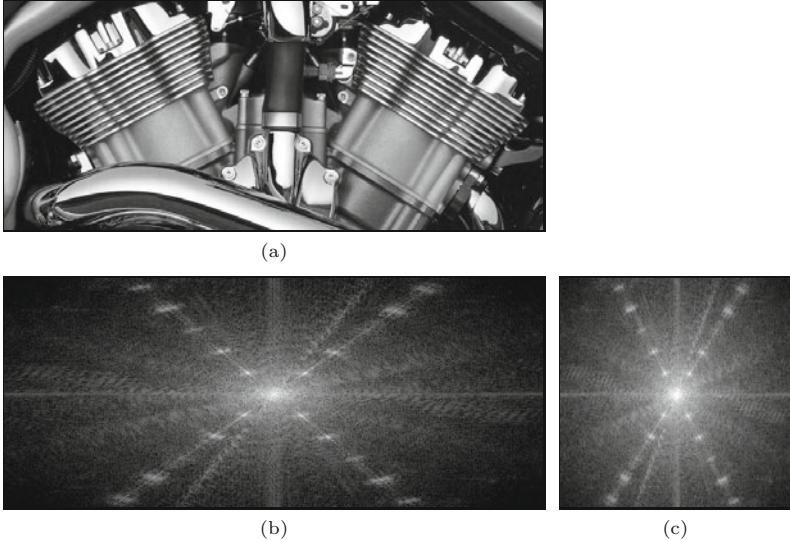
$$(m, n) = \pm \hat{f} \cdot (M \cos \psi, N \sin \psi), \quad (19.19)$$

as already shown in Fig. 19.5.

19.3.4 Normalizing the Geometry of the 2D Spectrum

From Eqn. (19.19) we can derive that in the special case of a sinusoid with spatial orientation $\psi = 45^\circ$ the corresponding spectral coefficients are found at the frequency coordinates

¹ $\text{ArcTan}(x, y)$ in Eqn. (19.18) denotes the inverse tangent function $\tan^{-1}(y/x)$ (also see Sec. F.1.6 in the Appendix).



19.3 FREQUENCIES AND ORIENTATION IN 2D

Fig. 19.7

Normalizing the 2D spectrum. Original image (a) with dominant oriented patterns that show up as clear peaks in the corresponding spectrum (b). Because the image and the spectrum are not square ($M \neq N$), orientations in the image are not the same as in the actual spectrum (b). After the spectrum is normalized to square proportions (c), we can clearly observe that the cylinders of this (Harley-Davidson *V-Rod*) engine are really arranged at a 60° angle.

$$(m, n) = \pm(\lambda M, \lambda N) \quad \text{for } -\frac{1}{2} \leq \lambda \leq +\frac{1}{2}, \quad (19.20)$$

that is, at the diagonals of the spectrum (see also Eqn. (19.17)). Unless the image (and thus the spectrum) is quadratic ($M = N$), the angle of orientation in the image and in the spectrum are not the same, though they coincide along the directions of the coordinate axes. This means that rotating an image by some angle α does turn the spectrum in the same direction but in general not by the same angle α !

To obtain identical orientations and turning angles in both the image and the spectrum, it is sufficient to scale the spectrum to square size such that the spectral resolution is the same along both frequency axes (as shown in Fig. 19.7).

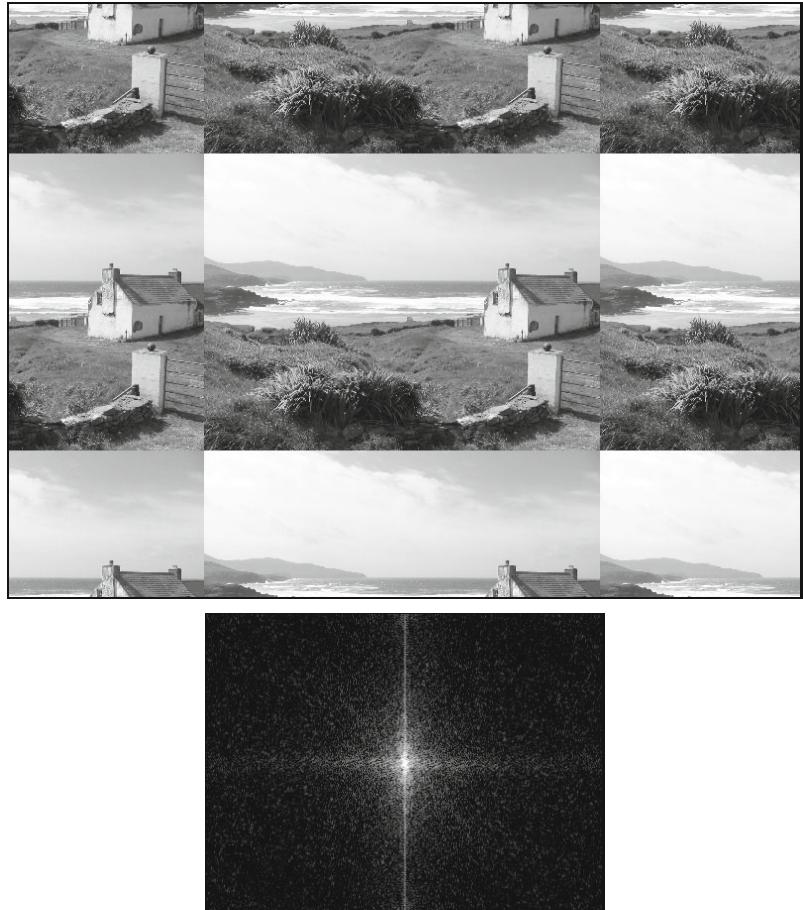
19.3.5 Effects of Periodicity

When interpreting the 2D DFT of images, one must always be aware of the fact that with any discrete Fourier transform, the signal function is implicitly assumed to be periodic in every dimension. Thus the transitions at the borders between the replicas of the image are also part of the signal, just like the interior of the image itself. If there is a large intensity difference between opposing borders of an image (e.g., between the upper and lower parts of a landscape image), then this causes strong transitions in the resulting periodic signal. Such steep discontinuities are of high bandwidth (i.e., the corresponding signal energy is spread over a wide range along the coordinate axes of the sampling grid; see Fig. 19.8). This broadband energy distribution along the main axes, which is often observed with real images, may lead to a suppression of other relevant signal components in the spectrum.

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.8

Effects of periodicity in the 2D spectrum. The discrete Fourier transform is computed under the implicit assumption that the image signal is periodic along both dimensions (top). Large differences in intensity at opposite image borders—here most notably in the vertical direction—lead to broad-band signal components that in this case appear as a bright line along the spectrum's vertical axis (bottom).



19.3.6 Windowing

One solution to this problem is to multiply the image function $g(u, v) = I(u, v)$ by a suitable *windowing function* $w(u, v)$, that is,

$$\tilde{g}(u, v) = g(u, v) \cdot w(u, v), \quad (19.21)$$

for $0 \leq u < M$, $0 \leq v < N$, prior to computing the DFT. The windowing function $w(u, v)$ should drop off continuously toward the image borders such that the transitions between image replicas are effectively eliminated. But multiplying the image with $w(u, v)$ has additional effects upon the spectrum. As we already know (from Eqn. (18.26)), a *multiplication* of two functions in signal space corresponds to a *convolution* of the corresponding spectra in frequency space, that is,

$$\tilde{G}(m, n) = G(m, n) * W(m, n). \quad (19.22)$$

To cause the least possible damage to the Fourier transform of the image, the ideal spectrum of $w(u, v)$ would be the impulse function $\delta(m, n)$. Unfortunately, this again corresponds to the constant windowing function $w(u, v) = 1$ with no windowing effect at all. In general, we can say that a broader spectrum of the windowing function

$w(u, v)$ smoothes the resulting spectrum more strongly and individual frequency components are harder to isolate.

Taking a picture is equivalent to cutting out a finite (usually rectangular) region from an infinite image plane, which can be simply modeled as a multiplication with a rectangular pulse function of width M and height N . So, in this case, the spectrum of the original intensity function is convolved with the spectrum of the rectangular pulse (box). The problem is that the spectrum of the rectangular box (see Fig. 19.9(a)) is of extremely high bandwidth and thus far off the ideal narrow impulse function.

These two examples demonstrate a dilemma: windowing functions should for one be as wide as possible to include a maximum part of the original image, and they should also drop off to zero toward the image borders but then again not too steeply to maintain a narrow windowing spectrum.

19.3.7 Common Windowing Functions

Suitable windowing functions should therefore exhibit soft transitions, and many variants have been proposed and analyzed both theoretically and for practical use (see, e.g., [34, Ch. 9, Sec. 9.3], [194, Ch. 10]). Table 19.1 lists the definitions of several popular windowing functions; the corresponding 2D (logarithmic) power spectra are displayed in Figs. 19.9 and 19.10.

The spectrum of the *rectangular pulse* function (Fig. 19.9(a)), which assigns identical weights to all image elements, exhibits a relatively narrow peak at the center, which promises little smoothing in the resulting windowed spectrum. Nevertheless, the spectral energy drops off quite slowly toward the higher frequencies, thus creating a rather wide spectrum. Not surprisingly, the behavior of the *elliptical* windowing function in Fig. 19.9(b) is quite similar. The *Gaussian* window in Fig. 19.9(c) demonstrates how the off-center spectral energy can be significantly suppressed by narrowing the windowing function, however, at the cost of a much wider peak at the center. In fact, none of the functions in Fig. 19.9 makes a good windowing function.

Obviously, the choice of a suitable windowing function is a delicate compromise since even apparently similar functions may exhibit largely different behaviors in the frequency spectrum. For example, good overall results can be obtained with the *Hanning* window (Fig. 19.10(c)) or the *Parzen* window (Fig. 19.10(d)), which are both easy to compute and frequently used in practice.

Figure 19.11 illustrates the effects of selected windowing functions upon the spectrum of an intensity image. As can be seen clearly, narrowing the windowing function leads to a suppression of the artifacts caused by the signal's implicit periodicity. At the same time, however, it also reduces the resolution of the spectrum; the spectrum becomes blurred, and individual peaks are widened.

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Table 19.1
2D windowing function definitions. The functions $w(u, v)$ have their maximum values at the image center, $w(M/2, N/2) = 1$. The values r_u , r_v , and $r_{u,v}$ used in the definitions are specified at the top of the table.

	Definitions:
$r_u = \frac{u-M/2}{M/2} = \frac{2u}{M}-1$,	$r_v = \frac{v-N/2}{N/2} = \frac{2v}{N}-1$,
$r_{u,v} = \sqrt{r_u^2 + r_v^2}$	
Elliptical window:	$w(u, v) = \begin{cases} 1 & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Gaussian window:	$w(u, v) = e^{\left(\frac{-r_{u,v}^2}{2\sigma^2}\right)}$, $\sigma = 0.3, \dots, 0.4$
Super-Gaussian window:	$w(u, v) = e^{\left(\frac{-r_{u,v}^n}{\kappa}\right)}$, $n = 6$, $\kappa = 0.3, \dots, 0.4$
Cosine² window:	$w(u, v) = \begin{cases} \cos\left(\frac{\pi}{2}r_u\right) \cdot \cos\left(\frac{\pi}{2}r_v\right) & \text{for } 0 \leq r_u, r_v \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Bartlett window:	$w(u, v) = \begin{cases} 1 - r_{u,v} & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Hanning window:	$w(u, v) = \begin{cases} 0.5 \cdot [\cos(\pi r_{u,v}) + 1] & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Parzen window:	$w(u, v) = \begin{cases} 1 - 6r_{u,v}^2 + 6r_{u,v}^3 & \text{for } 0 \leq r_{u,v} < 0.5 \\ 2 \cdot (1 - r_{u,v})^3 & \text{for } 0.5 \leq r_{u,v} < 1 \\ 0 & \text{otherwise} \end{cases}$

19.4 2D Fourier Transform Examples

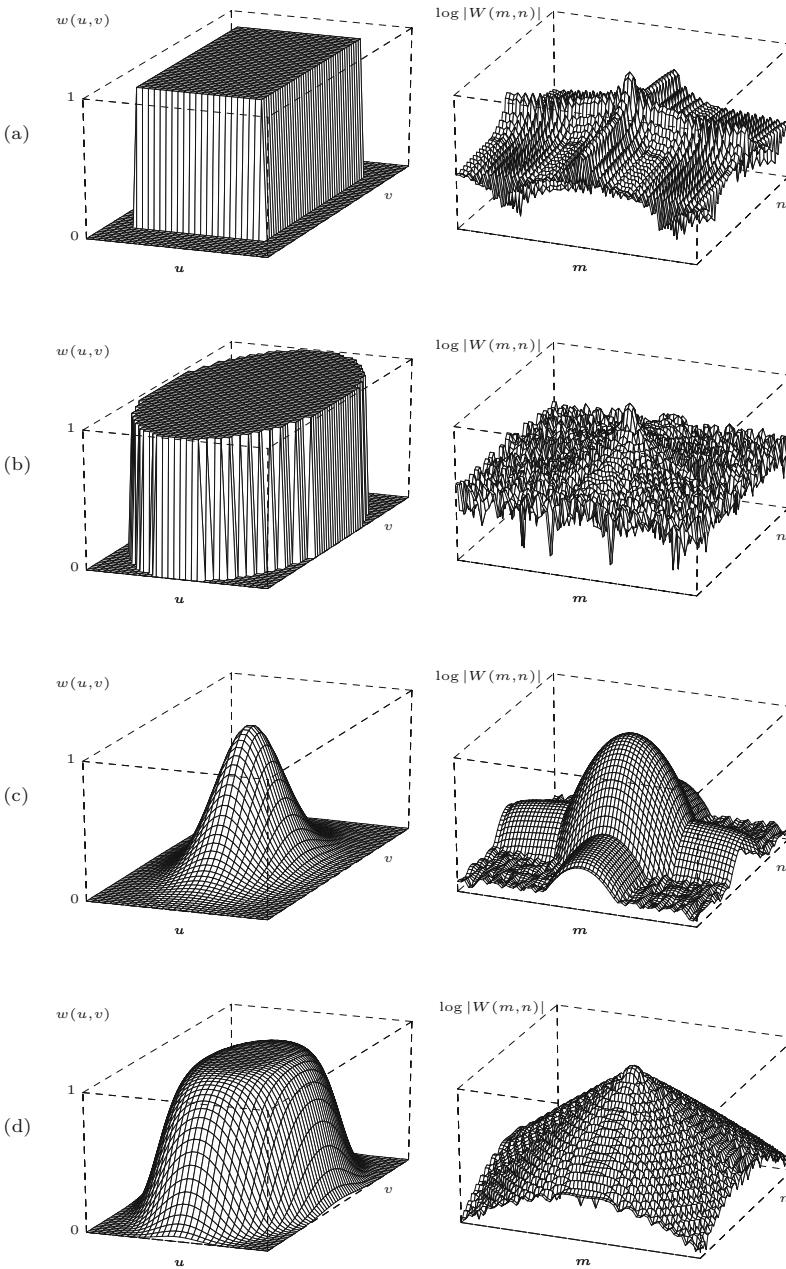
The following examples demonstrate some basic properties of the 2D DFT on real intensity images. All examples in Figs. 19.12–19.18 show a centered and squared spectrum with logarithmic intensity values (see Sec. 19.2).

Scaling

Figure 19.12 shows that scaling the image in signal space has the opposite effect in frequency space, analogous to the 1D case (see Ch. 18, Fig. 18.4).

Periodic Image Patterns

The images in Fig. 19.13 contain repetitive periodic patterns at various orientations and scales. They appear as distinct peaks at the corresponding positions (see Eqn. (19.19)) in the spectrum.



19.4 2D FOURIER TRANSFORM EXAMPLES

Fig. 19.9

Windowing functions and their logarithmic power spectra. Rectangular pulse (a), elliptical window (b), Gaussian window with $\sigma = 0.3$ (c), and super-Gaussian window of order $n = 6$ and $\kappa = 0.3$ (d). The windowing functions are deliberately of nonsquare size ($M : N = 1 : 2$).

Rotation

Figure 19.14 shows that rotating the image by some angle α rotates the spectrum in the same direction and—if the image is square—by the same angle.

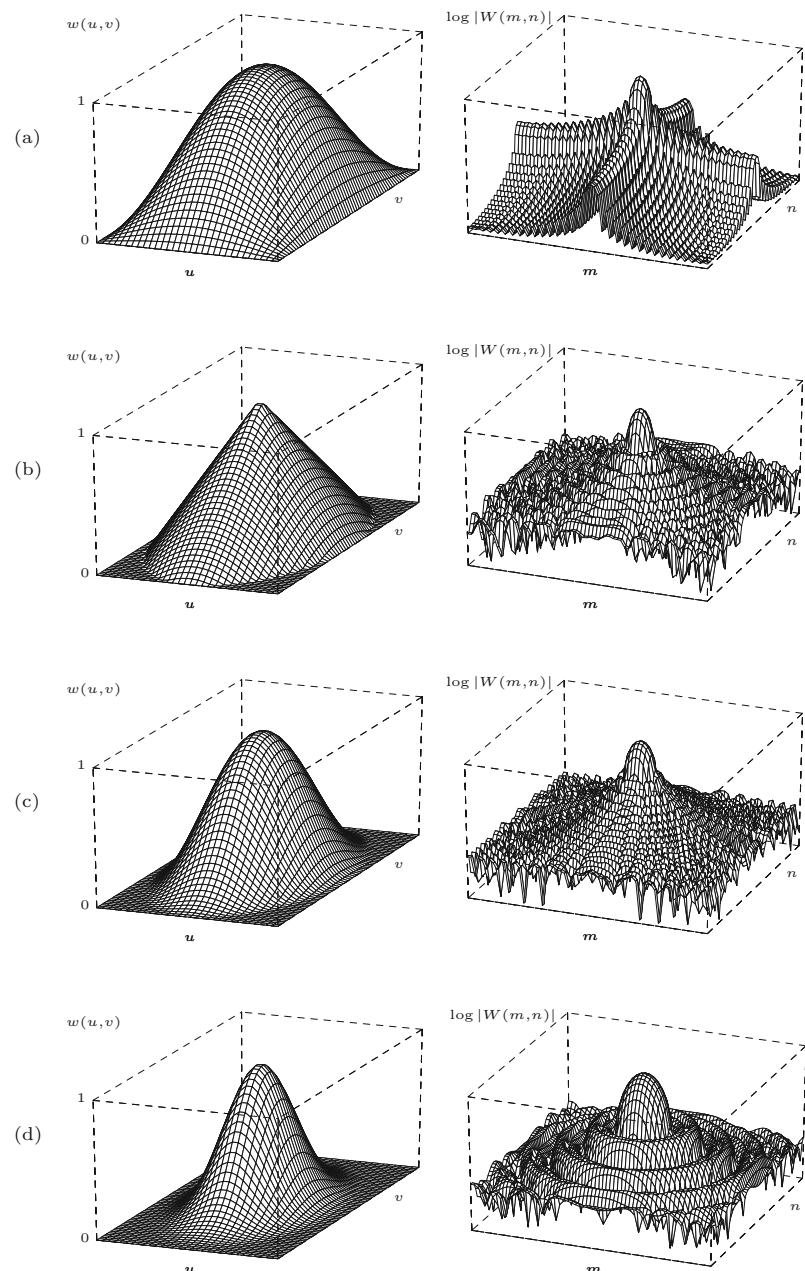
Oriented, elongated structures

Pictures of artificial objects often exhibit regular patterns or elongated structures that appear dominantly in the spectrum. The im-

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.10

Windowing functions and their logarithmic power spectra (*continued*). Cosine window (a), Bartlett window (b), Hanning window (c), and Parzen window (d).

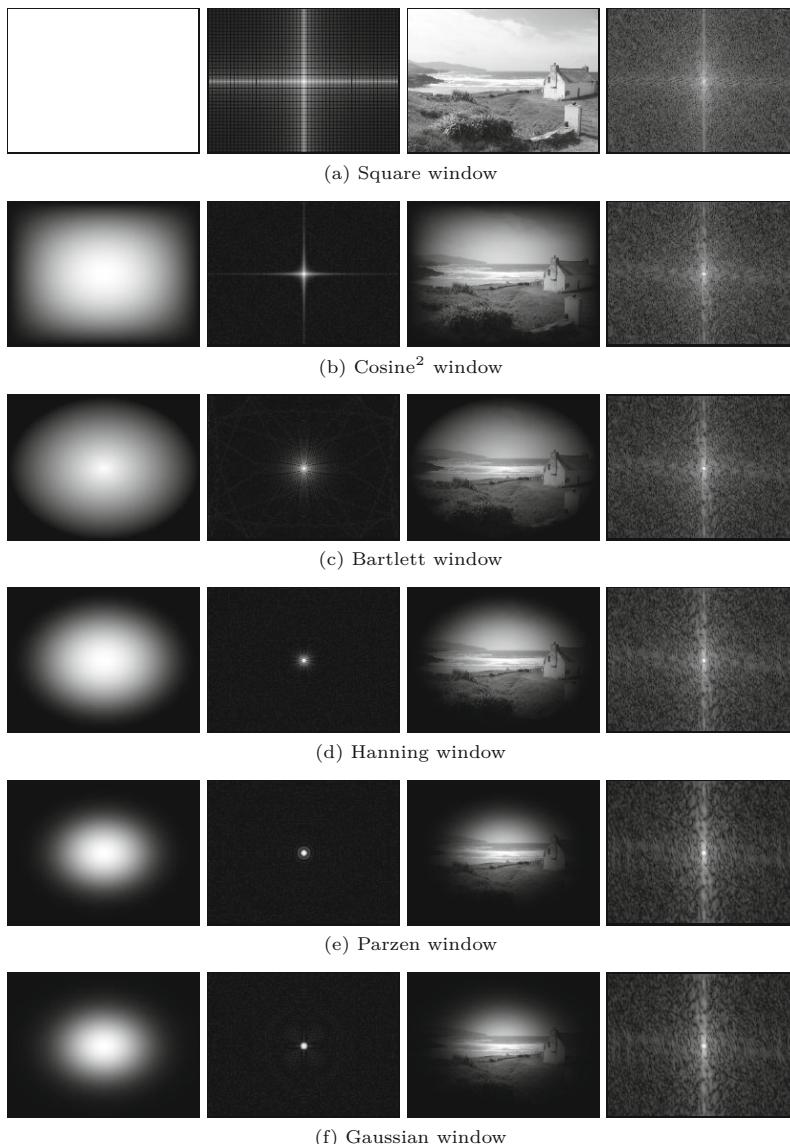


ages in Fig. 19.15 show several elongated structures that show up in the spectrum as bright streaks oriented perpendicularly to the main direction of the image patterns.

Natural images

Straight and regular structures are usually less dominant in images of natural objects and scenes, and thus the visual effects in the spectrum are not as obvious as with artificial objects. Some examples of this class of images are shown in Figs. 19.16 and 19.17.

Window function (linear) $w(u,v)$	Window spectrum (logarithmic) $\log W(m,n) $	Windowed image $g(u,v) \cdot w(u,v)$	Windowed image spectrum (log.) $\log G(m,n) * W(m,n) $
---	---	--	---



19.4 2D FOURIER TRANSFORM EXAMPLES

Fig. 19.11

Application of windowing functions on images. The plots show the windowing function $w(u,v)$, the logarithmic power spectrum of the windowing function $\log |W(m,n)|$, the windowed image $g(u,v) \cdot w(u,v)$, and the power spectrum of the windowed image $\log |G(m,n) * W(m,n)|$.

Print patterns

The regular patterns generated by the common raster print techniques (Fig. 19.18) are typical examples for periodic multidirectional structures, which stand out clearly in the corresponding Fourier spectrum.

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.12

DFT under image scaling. The rectangular pulse in the image function (a–c) creates a strongly oscillating power spectrum (d–f), as in the 1D case. Stretching the image causes the spectrum to contract and vice versa.

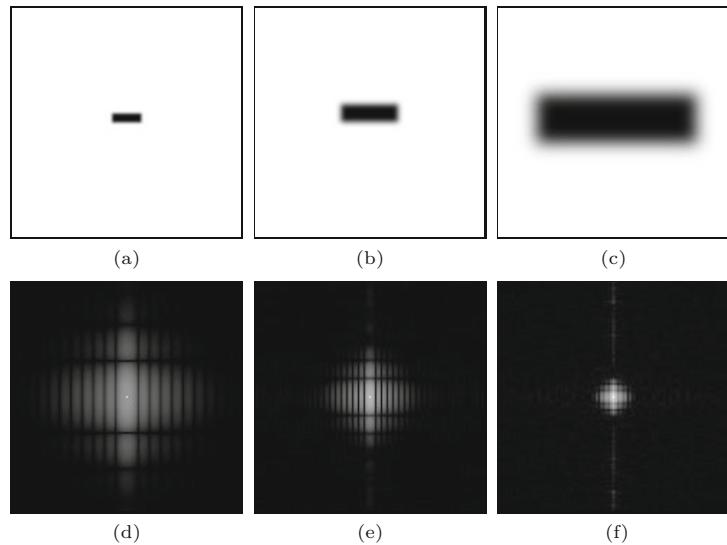
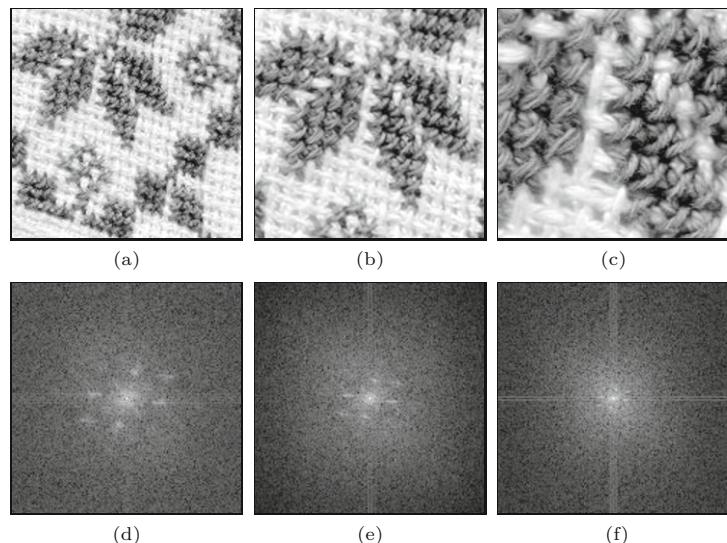


Fig. 19.13

DFT of oriented, repetitive patterns. The image function (a–c) contains patterns with three dominant orientations, which appear as pairs of corresponding frequency spots in the spectrum (c–f). Enlarging the image causes the spectrum to contract.

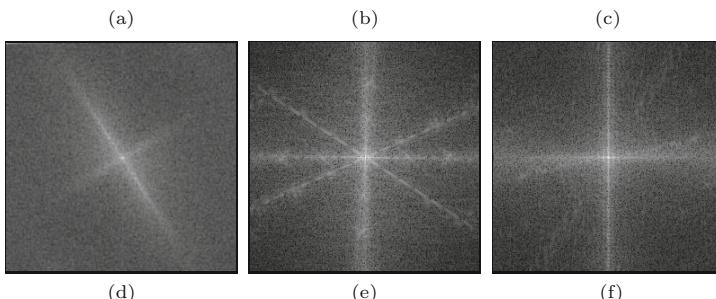
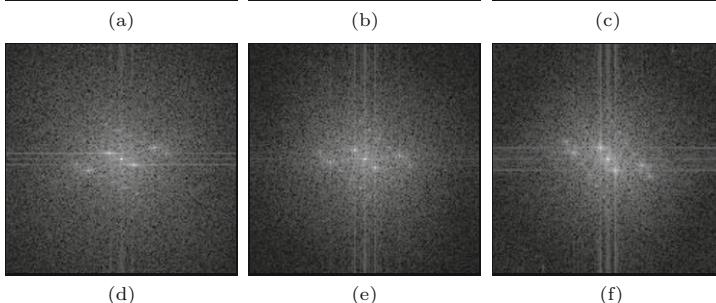
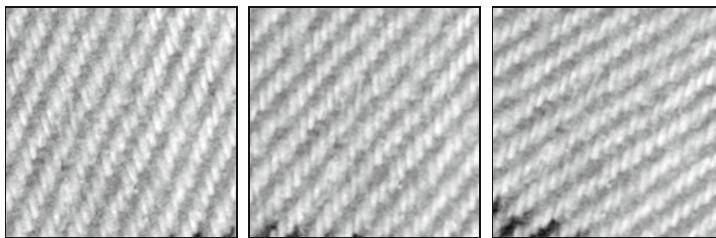


19.5 Applications of the DFT

The Fourier transform and the DFT in particular are important tools in many engineering disciplines. In digital signal and image processing, the DFT (and the FFT) is an indispensable “workhorse” with many applications in image analysis, filtering, and image reconstruction, just to name a few.

19.5.1 Linear Filter Operations in Frequency Space

Performing linear filter operations in frequency space is an interesting option because it provides an efficient way to apply filters of large spatial extent. The approach is based on the *convolution property* of the Fourier transform (see Ch. 18, Sec. 18.1.6), which states that a linear convolution in image space corresponds to a pointwise multiplication in frequency space. Thus the linear convolution $g * h \rightarrow g'$ between



an image $g(u, v)$ and a filter matrix $h(u, v)$ can be accomplished by the following steps:

$$\begin{array}{l}
 \text{image space:} \quad g(u, v) * h(u, v) = g'(u, v) \\
 \qquad\qquad\qquad\downarrow \qquad\qquad\qquad\downarrow \qquad\qquad\qquad\uparrow \\
 \qquad\qquad\qquad \text{DFT} \qquad \text{DFT} \qquad \text{DFT}^{-1} \\
 \qquad\qquad\qquad\downarrow \qquad\qquad\qquad\downarrow \qquad\qquad\qquad\uparrow \\
 \text{frequency space:} \quad G(m, n) \cdot H(m, n) \longrightarrow G'(m, n).
 \end{array} \tag{19.23}$$

First, the image g and the filter kernel² h are transformed to frequency space using the 2D DFT. The corresponding spectra G and H are then multiplied (pointwise), and the result G' is subsequently

19.5 APPLICATIONS OF THE DFT

Fig. 19.14

DFT—image rotation. The original image (a) is rotated by 15° (b) and 30° (c). The corresponding (squared) spectrum turns in the same direction and by exactly the same amount (d–f).

Fig. 19.15

DFT—superposition of image patterns. Strong, oriented subpatterns (a–c) are easy to identify in the corresponding spectrum (d–f). Notice the broadband effects caused by straight structures, such as the dark beam on the wall in (b, e).

² Note that the symbol h is used here for any 1D or 2D filter kernel and H for the corresponding Fourier spectrum. This should not be confused with the use of h , H for 1D and 2D filter kernels, respectively, in Ch. 5.

19 THE DISCRETE FOURIER TRANSFORM IN 2D

Fig. 19.16

DFT—natural image patterns. Examples of repetitive structures in natural images (a–c) that are also visible in the corresponding spectrum (d–f).

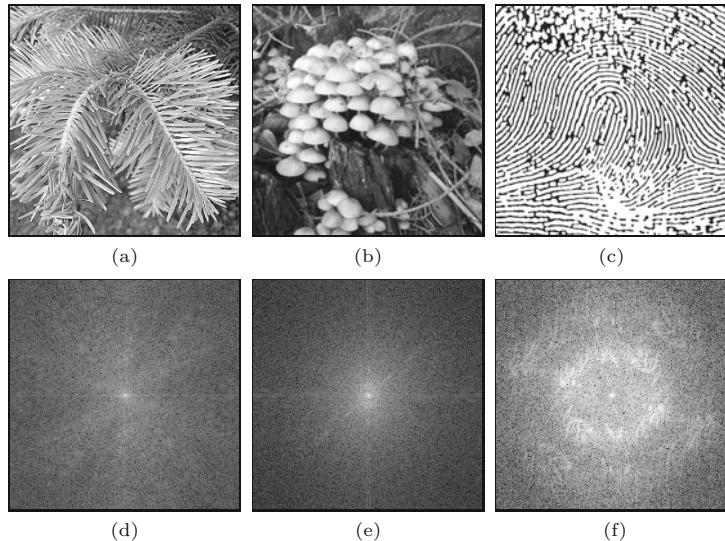
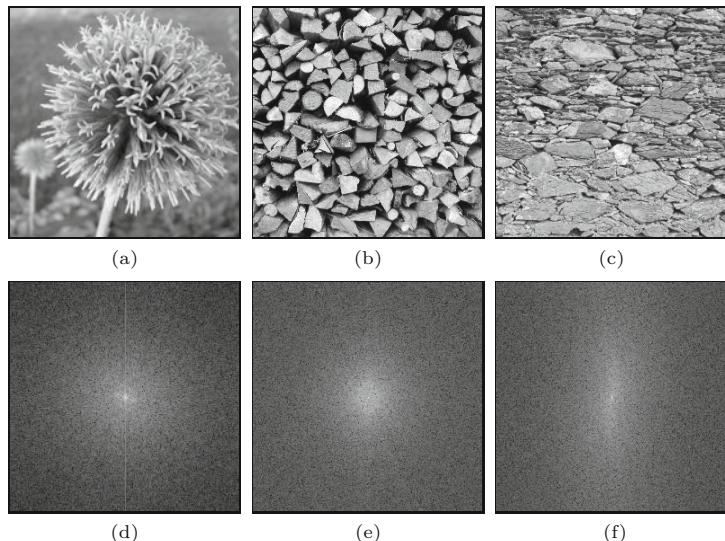


Fig. 19.17

DFT—natural image patterns with no dominant orientation.

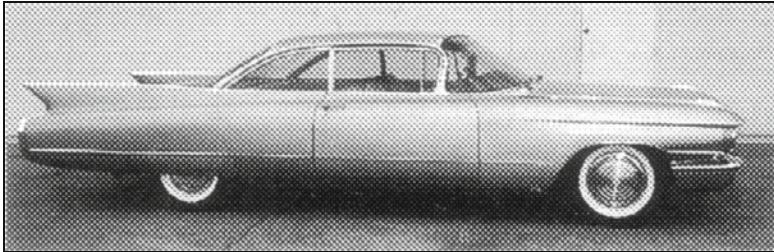
The repetitive patterns contained in these images (a–c) have no common orientation or sufficiently regular spacing to stand out locally in the corresponding Fourier spectra (d–f).



transformed back to image space using the inverse DFT, thus generating the filtered image g' .

The main advantage of this “detour” lies in its possible efficiency. A direct convolution for an image of size $M \times M$ and a filter matrix of size $N \times N$ requires $\mathcal{O}(M^2N^2)$ operations. Thus, time complexity increases quadratically with filter size, which is usually no problem for small filters but may render some larger filters too costly to implement. For example, a filter of size 50×50 already requires about 2500 multiplications and additions for every image pixel. In comparison, the transformation from image to frequency space and back can be performed in $\mathcal{O}(M \log_2 M)$ using the FFT, and the pointwise multiplication in frequency space requires M^2 operations, independent of the filter size.

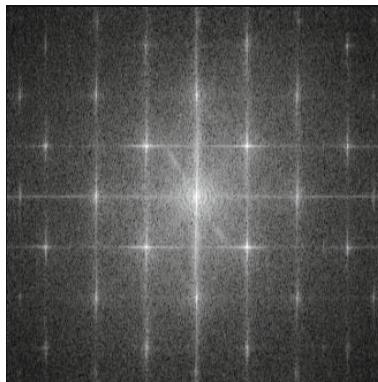
In addition, certain types of filters are easier to specify in frequency space than in image space; for example, an ideal low-pass



(a)



(b)



(c)

19.5 APPLICATIONS OF THE DFT

Fig. 19.18

DFT of a print pattern. The regular diagonally oriented raster pattern (a, b) is clearly visible in the corresponding power spectrum (c). It is possible (at least in principle) to remove such patterns by erasing these peaks in the Fourier spectrum and reconstructing the smoothed image from the modified spectrum using the inverse DFT.

filter, which can be described very compactly in frequency space. Further details on filter operations in frequency space can be found, for example, in [88, Sec. 4.4].

19.5.2 Linear Convolution and Correlation

As discussed in Chapter 5, Sec. 5.3, a linear correlation is the same as a linear convolution with a mirrored filter function. Therefore, the correlation can be computed just like the convolution operation in the frequency domain by following the steps described in Eqn. (19.23). This could be advantageous for comparing images using correlation methods (see Ch. 23, Sec. 23.1.1) because in this case the image and the “filter” matrix (i.e., the second image) are of similar size and thus usually too large to be processed in image space.

Some operations in ImageJ, such as *correlate*, *convolve*, or *deconvolve*, are also implemented in the “Fourier domain” (FD) using the 2D DFT. They can be invoked through the menu **Process** ▷ **FFT** ▷ **FD Math**.

19.5.3 Inverse Filters

Filtering in the frequency domain opens another interesting perspective: reversing the effects of a filter, at least under restricted conditions. In the following, we describe the basic idea only.

Assume we are given an image g_{blur} that has been generated from an original image g_{orig} by some linear filter, for example, motion blur induced by a moving camera. Under the assumption that this image modification can be modeled sufficiently by a linear filter function

h_{blur} , we can state that

$$g_{\text{blur}}(u, v) = (g_{\text{orig}} * h_{\text{blur}})(u, v). \quad (19.24)$$

Recalling that in frequency space this corresponds to a multiplication of the corresponding spectra, that is,

$$G_{\text{blur}}(m, n) = G_{\text{orig}}(m, n) \cdot H_{\text{blur}}(m, n) \quad (19.25)$$

it should be possible to reconstruct the original (non-blurred) image by computing the inverse Fourier transform of the expression

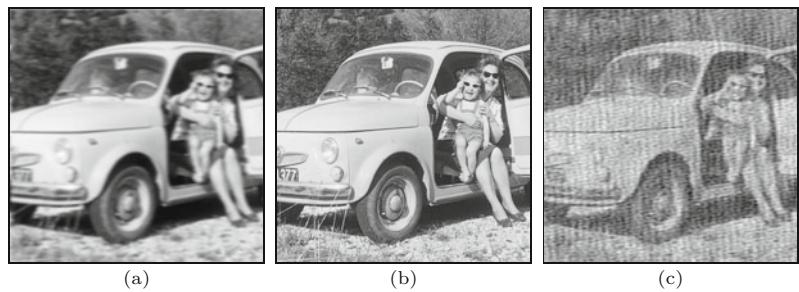
$$G_{\text{orig}}(m, n) = \frac{G_{\text{blur}}(m, n)}{H_{\text{blur}}(m, n)}. \quad (19.26)$$

Unfortunately, this “inverse filter” only works if the spectral coefficients H_{blur} are nonzero, because otherwise the resulting values are infinite. But even small values of H_{blur} , which are typical at high frequencies, lead to large coefficients in the reconstructed spectrum and, as a consequence, large amounts of image noise.

It is also important that the real filter function be accurately approximated, because otherwise the reconstructed image may strongly deviate from the original. The example in Fig. 19.19 shows an image that has been blurred by smooth horizontal motion, whose effect can easily be modeled as a linear convolution. If the filter function that caused the blurring is known exactly, then the reconstruction of the original image can be accomplished without any problems (Fig. 19.19(c)). However, as shown in Fig. 19.19(d), large errors occur if the inverse filter deviates only marginally from the real filter, which quickly renders the method useless.

Fig. 19.19

Removing motion blur by inverse filtering. Original image (a); image blurred by horizontal motion (b); reconstruction using the exact (known) filter function (c); result of the inverse filter when the filter function deviates marginally from the real filter (d).



Beyond this simple idea (which is often referred to as “deconvolution”), better methods for inverse filtering exist, such as the *Wiener filter* and related techniques (see, e.g., [88, Sec. 5.4], [128, Sec. 8.3], [126, Sec. 17.8], [43, Ch. 16]).

19.6 Exercises

Exercise 19.1. Implement the 2D DFT using the 1D DFT, as described in Sec. 19.1.2. Apply the 2D DFT to real intensity images of arbitrary size and display the results (by converting to ImageJ `FloatProcessor` images). Implement the inverse transform and verify that the back-transformed result is identical to the original image.

Exercise 19.2. Assume that the 2D Fourier spectrum of an image with size 640×480 and a spatial resolution of 72 dpi shows a dominant peak at position $\pm(100, 100)$. Determine the orientation and effective frequency (in cycles per cm) of the corresponding image pattern.

19.6 EXERCISES

Exercise 19.3. An image with size 800×600 contains a wavy intensity pattern with an effective period of 12 pixels, oriented at 30° . At which frequency coordinates will this pattern manifest itself in the discrete Fourier spectrum?

Exercise 19.4. Generalize Eqn. (19.15) and Eqns. (19.17)–(19.19) for the case where the sampling intervals are *not* identical along the x and y axes (i.e., for $\tau_x \neq \tau_y$).

Exercise 19.5. Implement the *elliptical* and the *super-Gauss* windowing functions (Table 19.1) as ImageJ plugins, and investigate the effects of these windows upon the resulting spectra. Also compare the results to the case where *no* windowing function is used.

The Discrete Cosine Transform (DCT)

The Fourier transform and the DFT are designed for processing complex-valued signals, and they always produce a complex-valued spectrum even in the case where the original signal was strictly real-valued. The reason is that neither the real nor the imaginary part of the Fourier spectrum alone is sufficient to represent (i.e., reconstruct) the signal completely. In other words, the corresponding cosine (for the real part) or sine functions (for the imaginary part) alone do not constitute a complete set of basis functions.

On the other hand, we know (see Ch. 18, Eqn. (18.21)) that a real-valued signal has a symmetric Fourier spectrum, so only one half of the spectral coefficients need to be computed without losing any signal information.

There are several spectral transformations that have properties similar to the DFT but do not work with complex function values. The discrete cosine transform (DCT) is a well known example that is particularly interesting in our context because it is frequently used for image and video compression. The DCT uses only cosine functions of various wave numbers as basis functions and operates on real-valued signals and spectral coefficients. Similarly, there is also a discrete sine transform (DST) based on a system of sine functions [128].

20.1 1D DCT

The discrete cosine transform is not, as one may falsely assume, only a “one-half” variant of the discrete Fourier transform. In the 1D case, the *forward* cosine transform for a signal $g(u)$ of length M is defined as

$$G(m) = \sqrt{\frac{2}{M}} \cdot \sum_{u=0}^{M-1} g(u) \cdot c_m \cdot \cos\left(\pi \frac{m(2u+1)}{2M}\right), \quad (20.1)$$

for $0 \leq m < M$, and the *inverse* transform is

$$g(u) = \sqrt{\frac{2}{M}} \cdot \sum_{m=0}^{M-1} G(m) \cdot c_m \cdot \cos\left(\pi \frac{m(2u+1)}{2M}\right), \quad (20.2)$$

for $0 \leq u < M$, with

$$c_m = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } m = 0, \\ 1 & \text{otherwise.} \end{cases} \quad (20.3)$$

Note that the index variables (u, m) are used differently in the forward and inverse transforms (Eqns. (20.2)–(20.1)), so the two transforms are—unlike the DFT—*not* symmetric.

20.1.1 DCT Basis Functions

One may ask how it is possible that the DCT can work without any sine functions, while they are essential in the DFT. The trick is to divide all frequencies in half such that they are spaced more densely and thus the frequency resolution in the spectrum is doubled. Comparing the cosine parts of the DFT basis functions (Eqn. (18.49)) and those of the DCT (Eqn. (20.1)),

$$\text{DFT: } C_m^M(u) = \cos\left(2\pi \frac{mu}{M}\right), \quad (20.4)$$

$$\text{DCT: } D_m^M(u) = \cos\left(\pi \frac{m(2u+1)}{2M}\right) = \cos\left(2\pi \frac{m(u+0.5)}{2M}\right), \quad (20.5)$$

one can see that, for a given wave number m , the period ($\tau_m = 2\frac{M}{m}$) of the corresponding DCT basis function is double the period of the DFT basis functions ($\tau_m = \frac{M}{m}$). Notice that the DCT basis functions are also *phase-shifted* by 0.5 units.

Figure 20.1 shows the DCT basis functions $D_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0, \dots, 7$. For example, the basis function $D_7^8(u)$ for wave number $m = 7$ performs seven full cycles over a length of $2M = 16$ units and therefore has the radial frequency $\omega = m/2 = 3.5$.

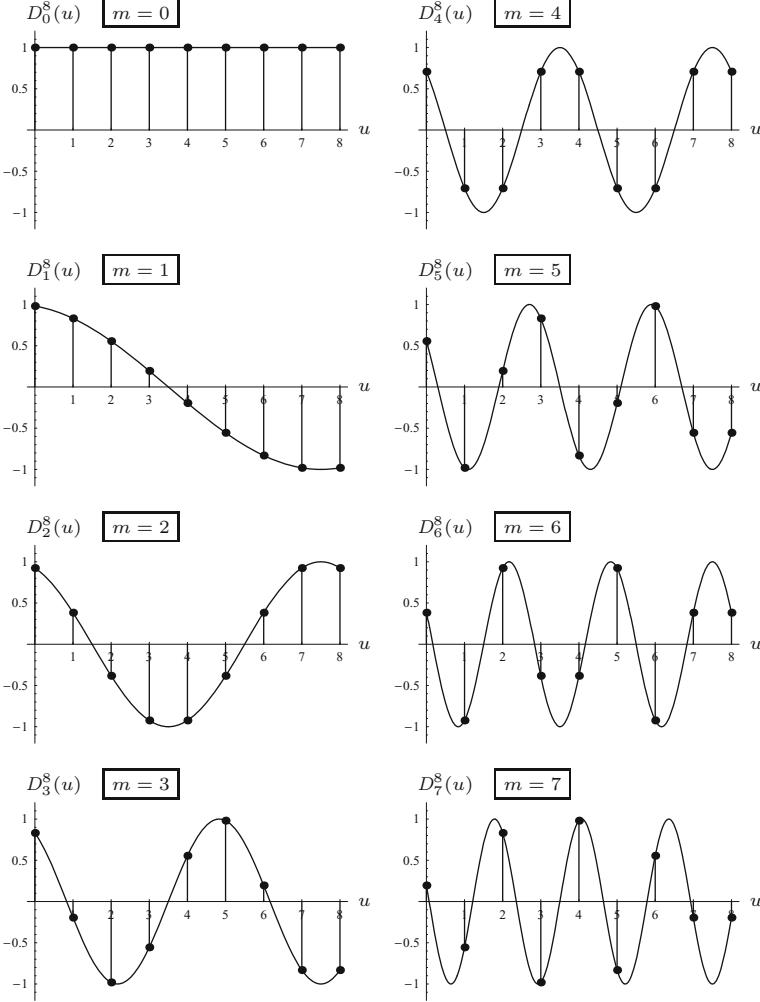
20.1.2 Implementing the 1D DCT

Since the DCT does not create any complex values and the forward and inverse transforms (Eqns. (20.1) and (20.2)) are almost identical, the whole procedure is fairly easy to implement in Java, as shown in Prog. 20.1. The only notable detail is that the factor c_m in Eqn. (20.1) is independent of the iteration variable u and can thus be calculated outside the inner summation loop (see Prog. 20.1, line 8).

Of course, much more efficient (“fast”) DCT algorithms exist. Moreover, the DCT can also be computed in $\mathcal{O}(M \log_2 M)$ time using the FFT [128, p. 152].

20.2 2D DCT

The 2D form of the DCT follows immediately from the the 1D definition (Eqns. (20.1) and (20.2)), resulting in the 2D forward transform



20.2 2D DCT

Fig. 20.1 DCT basis functions $D_0^M(u)$, ..., $D_7^M(u)$ for $M = 8$. Each plot shows both the discrete function (round dots) and the corresponding continuous function. Compared with the basis functions of the DFT (Figs. 18.11 and 18.12), all frequencies are divided in half and the DCT basis functions are phase-shifted by 0.5 units. All DCT basis functions are thus periodic over the length $2M = 16$ (as compared with M for the DFT).

$$G(m, n) = \frac{2}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} [g(u, v) \cdot c_m \cos\left(\frac{\pi(2u+1)m}{2M}\right) \cdot c_n \cos\left(\frac{\pi(2v+1)n}{2N}\right)] \quad (20.6)$$

$$= \frac{2 \cdot c_m \cdot c_n}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} [g(u, v) \cdot D_m^M(u) \cdot D_n^N(v)], \quad (20.7)$$

for $0 \leq m < M$, $0 \leq n < N$, and the inverse transform

$$g(u, v) = \frac{2}{\sqrt{MN}} \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} [G(m, n) \cdot c_m \cos\left(\frac{\pi(2u+1)m}{2M}\right) \cdot c_n \cos\left(\frac{\pi(2v+1)n}{2N}\right)] \quad (20.8)$$

$$= \frac{2}{\sqrt{MN}} \cdot \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} [G(m, n) \cdot c_m \cdot D_m^M(u) \cdot c_n \cdot D_n^N(v)], \quad (20.9)$$

for $0 \leq u < M$, $0 \leq v < N$. The coefficients c_m and c_n in Eqns. (20.7) and (20.9) are the same as in the 1D case (Eqn. (20.3)). Notice

20 THE DISCRETE COSINE TRANSFORM (DCT)

Prog. 20.1

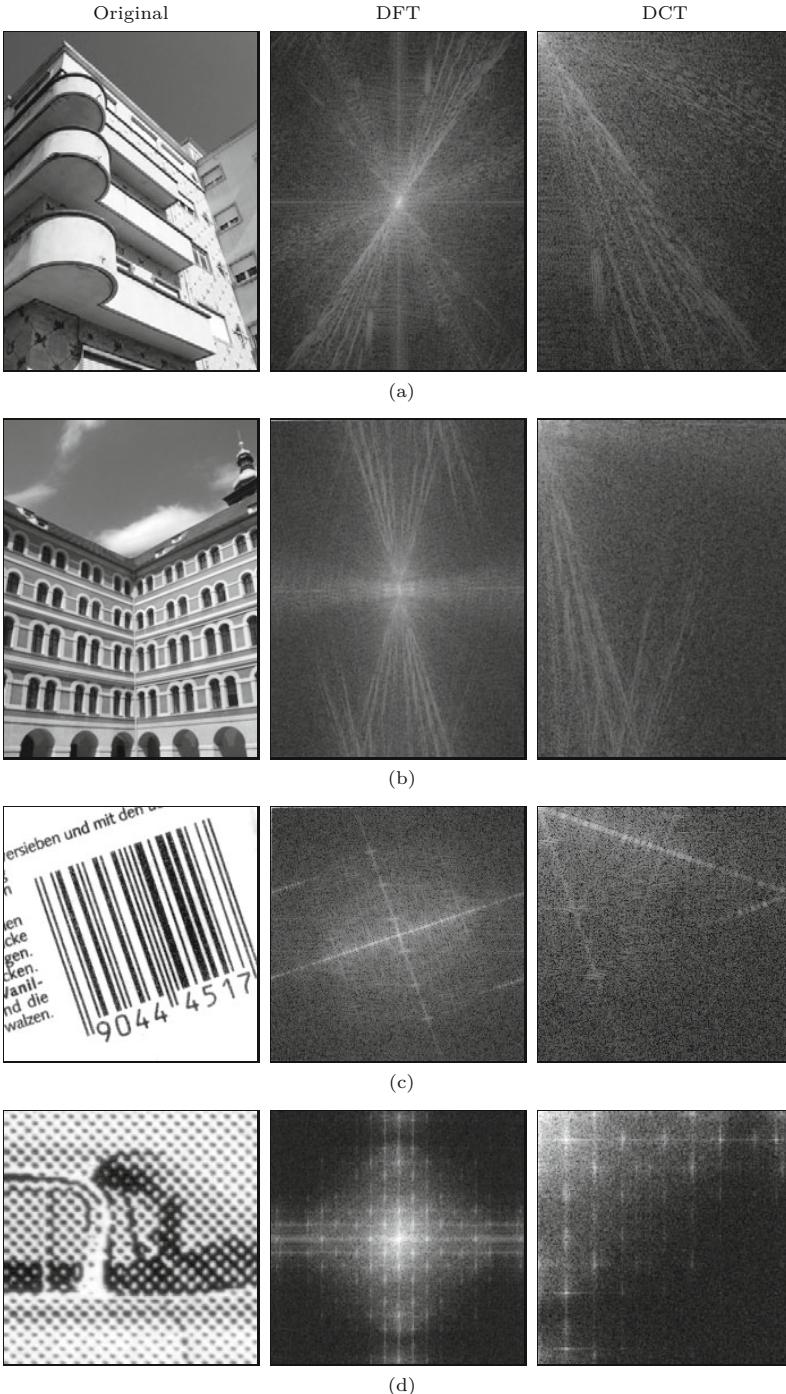
1D DCT (Java implementation). The method `DCT()` computes the forward transform for a real-valued signal vector \mathbf{g} of arbitrary length according to the definition in Eqn. (20.1). The method returns the DCT spectrum as a real-valued vector of the same length as the input vector \mathbf{g} . The inverse transform `iDCT()` computes the inverse DCT for the real-valued cosine spectrum \mathbf{G} .

```
1  double[] DCT (double[] g) { // forward DCT on signal g
2      int M = g.length;
3      double s = Math.sqrt(2.0 / M); // common scale factor
4      double[] G = new double[M];
5      for (int m = 0; m < M; m++) {
6          double cm = 1.0;
7          if (m == 0)
8              cm = 1.0 / Math.sqrt(2);
9          double sum = 0;
10         for (int u = 0; u < M; u++) {
11             double Phi = Math.PI * m * (2 * u + 1) / (2 * M);
12             sum += g[u] * cm * Math.cos(Phi);
13         }
14         G[m] = s * sum;
15     }
16     return G;
17 }
18
19
20 double[] iDCT (double[] G) { // inverse DCT on spectrum G
21     int M = G.length;
22     double s = Math.sqrt(2.0 / M); //common scale factor
23     double[] g = new double[M];
24     for (int u = 0; u < M; u++) {
25         double sum = 0;
26         for (int m = 0; m < M; m++) {
27             double cm = 1.0;
28             if (m == 0)
29                 cm = 1.0 / Math.sqrt(2);
30             double Phi = Math.PI * m * (2 * u + 1) / (2 * M);
31             sum += G[m] * cm * Math.cos(Phi);
32         }
33         g[u] = s * sum;
34     }
35     return g;
36 }
```

that in the forward transform (and only there!) the factors c_m , c_n are independent of the iteration variables u, v and can thus be placed outside the summation (as shown in Eqn. (20.7)).

20.2.1 Examples

Figure 20.2 shows several examples of the DCT in comparison with the results of the DFT. Since the DCT spectrum is (in contrast to the DFT spectrum) not symmetric, it does not get centered but is displayed in its original form with its origin at the upper left corner. The intensity corresponds to the logarithm of the absolute value in the case of the (real-valued) DCT spectrum. Similarly, the usual logarithmic power spectrum is shown for the DFT. Notice that the DCT is not simply a section of the DFT but obviously combines structures from adjacent quadrants of the Fourier spectrum.



20.2 2D DCT

Fig. 20.2

2D DFT versus DCT. Both transforms show the frequency effects of image structures in a similar fashion. In the real-valued DCT spectrum (right), all coefficients are contained in a single quadrant and the frequency resolution is doubled compared with the DFT power spectrum (center). The DFT spectrum is centered as usual, while the origin of the DCT spectrum is located at the upper left corner. Both spectral plots display logarithmic intensity values.

20.2.2 Separability

Similar to the DFT (see Ch. 19, Eqn. (19.9)), the 2D DCT can also be separated into two successive 1D transforms. To make this fact clear, the forward DCT (Eqn. (20.7)) can be expressed in the form

$$G(m, n) = \sqrt{\frac{2}{N}} \cdot \sum_{v=0}^{N-1} \underbrace{\left[\sqrt{\frac{2}{M}} \cdot \sum_{u=0}^{M-1} g(u, v) \cdot c_m \cdot D_m^M(u) \cdot c_n \cdot D_n^N(v) \right]}_{\text{1D DCT of } g(\cdot, v)}. \quad (20.10)$$

The inner expression in Eqn. (20.10) corresponds to a 1D DCT of the v th line $g(\cdot, v)$ of the 2D signal function. Thus, as with the 2D DFT, one can first apply a 1D DCT to every line of an image and subsequently a DCT to each column. Of course, one could equally follow the reverse order by doing a DCT on the columns first and then on the rows.

The DCT is often used for image compression, in particular for JPEG where the size of the transformed sub-images is fixed at 8×8 and the processing can be highly optimized. Applying the DCT to square images (or sub-images) of size $M \times M$ is indeed an important special case. Here the DCT is commonly expressed in matrix form,

$$\mathbf{G} = \mathbf{A} \cdot \mathbf{g} \cdot \mathbf{A}^\top, \quad (20.11)$$

where the matrices \mathbf{g} and \mathbf{G} (both of size $M \times M$) represent the 2D signal and the resulting DCT spectrum, respectively. \mathbf{A} is a quadratic, real-valued transformation matrix with the elements (cf. Eqn. (20.1))

$$A_{i,j} = \sqrt{\frac{2}{N}} \cdot c_i \cdot \cos\left(\pi \cdot \frac{i \cdot (2j+1)}{2M}\right), \quad (20.12)$$

with $0 \leq i, j < M$ and c_i as defined in Eqn. (20.3). The x/y separability of the DCT is easy to see in this notation. The matrix \mathbf{A} is real-valued and *orthonormal*, i.e., $\mathbf{A} \cdot \mathbf{A}^\top = \mathbf{A}^\top \cdot \mathbf{A} = \mathbf{I}$ and so its transpose \mathbf{A}^\top is identical to the inverse matrix \mathbf{A}^{-1} . Thus the associated inverse transformation from the DCT spectrum \mathbf{G} back to the signal \mathbf{g} can be carried out in the form

$$\mathbf{g} = \mathbf{A}^\top \cdot \mathbf{G} \cdot \mathbf{A}, \quad (20.13)$$

with the same matrices \mathbf{A} and \mathbf{A}^\top as used in the forward transform. For example, for $M = 4$ the DCT transformation matrix is

$$\mathbf{A} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \quad (20.14)$$

$$= \begin{pmatrix} \frac{1}{2} \cos(0) & \frac{1}{2} \cos(0) & \frac{1}{2} \cos(0) & \frac{1}{2} \cos(0) \\ \frac{1}{\sqrt{2}} \cos\left(\frac{\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{3\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{5\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{7\pi}{8}\right) \\ \frac{1}{\sqrt{2}} \cos\left(\frac{2\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{6\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{8\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{10\pi}{8}\right) \\ \frac{1}{\sqrt{2}} \cos\left(\frac{3\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{9\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{15\pi}{8}\right) & \frac{1}{\sqrt{2}} \cos\left(\frac{21\pi}{8}\right) \end{pmatrix} \quad (20.15)$$

$$\approx \begin{pmatrix} 0.50000 & 0.50000 & 0.50000 & 0.50000 \\ 0.65328 & 0.27060 & -0.27060 & -0.65328 \\ 0.50000 & -0.50000 & -0.50000 & 0.50000 \\ 0.27060 & -0.65328 & 0.65328 & -0.27060 \end{pmatrix}. \quad (20.16)$$

For the arbitrarily chosen 2D signal (i.e., “image”)

$$\mathbf{g} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 7 & 2 & 0 & 9 \\ 6 & 5 & 2 & 5 \\ 0 & 9 & 8 & 1 \end{pmatrix}, \quad (20.17)$$

for example, the DCT spectrum obtained with Eqn. (20.11) is

$$\mathbf{G} = \mathbf{A} \cdot \mathbf{g} \cdot \mathbf{A}^T \approx \begin{pmatrix} 16.00000 & -0.95671 & 0.50000 & -2.30970 \\ -2.61313 & -1.81066 & 6.57924 & 0.45711 \\ -2.00000 & -1.65642 & -8.50000 & 1.22731 \\ -1.08239 & 0.95711 & -1.10162 & 0.31066 \end{pmatrix}, \quad (20.18)$$

which is the same as the result from Eqn. (20.7) or, alternatively, Eqn. (20.10).

The matrix notation of the DCT, as shown in Eqn. (20.11) and Eqn. (20.13), is particularly useful for describing the transformation of small, fixed-size sub-images. This is an important component common in most image and video compression methods (including JPEG and MPEG) that calls for efficient implementations.

20.3 Java Implementation

A straightforward Java implementation of the one- and two-dimensional DCT is available online as part of the `imagingbook` library.¹ For efficiency reasons, the following methods generally work “in place”, i.e., the supplied data array is destructively modified by the transformation.

Dct1d (class)

This class implements the *1D* DCT (see also Prog. 20.1):

`Dct1d (int M)`

Constructor; `M` denotes the length of the expected signal.

`void DCT (double[] g)`

Calculates the DCT spectrum of the one-dimensional signal `g`. The array `g` is modified, its content being replaced by the resulting spectrum.

`void iDCT (double[] G)`

Reconstructs the original signal from the one-dimensional DCT spectrum `G`. The array `G` is modified, its content being replaced by the reconstructed signal.

Pre-calculated cosine tables are used in both the forward and inverse transformation for efficient processing.

Dct2d (class)

This class implements the *2D* DCT (by using class `Dct1d`):

`Dct2d ()`

Constructor; in this case no dimension arguments are required.

¹ Package `imagingbook.pub.dct`.

```
void DCT (float [][] g)
    Calculates the DCT spectrum of the 2D signal g. The array g
    is modified.

void iDCT (float [][] G)
    Reconstructs the original signal from the two-dimensional
    DCT spectrum G. The array G is modified.

FloatProcessor DCT (FloatProcessor g)
    Calculates the DCT spectrum of the image g and returns a
    new image with the resulting spectrum (g is not modified).

FloatProcessor iDCT (FloatProcessor G)
    Calculates the inverse DCT from the 2D spectrum G and re-
    turns the reconstructed image (G is not modified).
```

20.4 Other Spectral Transforms

Apparently, the Fourier transform is not the only way to represent a given signal in frequency space; in fact, numerous similar transforms exist. Some of these, such as the discrete cosine transform, also use sinusoidal basis functions, while others, such as the *Hadamard* transform (also known as the *Walsh* transform), build on binary 0/1-functions [43, 126].

All of these transforms are of *global* nature; i.e., the value of any spectral coefficient is equally influenced by all signal values, independent of the spatial position in the signal. Thus a peak in the spectrum could be caused by a high-amplitude event of local extent as well as by a widespread, continuous wave of low amplitude. Global transforms are therefore of limited use for the purpose of detecting or analyzing local events because they are incapable of capturing the spatial position and extent of events in a signal.

A solution to this problem is to use a set of *local*, spatially limited basis functions (“wavelets”) in place of the global, spatially fixed basis functions. The corresponding *wavelet transform*, of which several versions have been proposed, allows the simultaneous localization of repetitive signal components in both signal space *and* frequency space [158].

20.5 Exercises

Exercise 20.1. Implement an efficient (“hard-coded”) Java method for computing the 1D DCT of length $M = 8$ that operates without iterations (loops) and contains all necessary coefficients as precomputed constants.

Exercise 20.2. Consider how the implementation of the one-dimensional DCT in Prog. 20.1 could be accelerated by using a pre-calculated table of the cosine values (for a given signal length M). Hint: A table of length $4M$ is required.

Exercise 20.3. Verify by numerical computation that the DCT basis functions $D_m^M(u)$ for $0 \leq m, u < M$ (Eqn. (20.5)) are pairwise

orthogonal; i.e., the inner product of the vectors $D_m^M \cdot D_n^M$ is zero for any pair $m \neq n$.

20.5 EXERCISES

Exercise 20.4. Implement the 2D DCT (Sec. 20.2) as an ImageJ plugin for images of arbitrary size. Make use of the fact that the 2D DCT can be performed as a sequence of 1D transforms (see Eqn. (20.10)).

Exercise 20.5. Verify for the 4×4 DCT example in Eqn. (20.18) that the result of the inverse transformation in Eqn. (20.13) is really identical to the original signal \mathbf{g} in Eqn. (20.17).

Exercise 20.6. Show that the $M \times M$ matrix \mathbf{A} (with elements as defined in Eqn. (20.12)) is really orthonormal, i.e., $\mathbf{A} \cdot \mathbf{A}^\top = \mathbf{I}$.

Geometric Operations

Common to all the filters and point operations described so far is the fact that they may change the intensity function of an image but the position of each pixel, and thus the geometry of the image, remains the same. The purpose of geometric operations, which are discussed in this chapter, is to deform an image by altering its geometry. Typical examples are shifting, rotating, or scaling images, as shown in Fig. 21.1. Geometric operations are frequently needed in practical applications, for example, in virtually any modern graphical computer interface. Today we take for granted that windows and images in graphic or video applications can be zoomed continuously to arbitrary size. Geometric image operations are also important in computer graphics where textures, which are usually raster images, are deformed to be mapped onto the corresponding 3D surfaces, possibly in real time. Of course, geometric operations are not as simple as their commonality may suggest. While it is obvious, for example, that an image could be enlarged by some integer factor n simply by replicating each pixel $n \times n$ times, the results would probably not be appealing, and it also gives us no immediate idea how to handle continuous scaling, rotating images, or other image deformations. In general, geometric operations that achieve high-quality results are not trivial to implement and are also computationally demanding, even on today's fast computers.

In principle, a geometric operation transforms a given image I to a new image I' by modifying the *coordinates* of image pixels,

$$I'(x', y') \leftarrow I(x, y), \quad (21.1)$$

that is, the value of the image function I at the original location (x, y) moves to the new position (x', y') in the transformed image I' . Thus (at least in the continuous case) the *values* of the image elements do not change but only their *positions*.

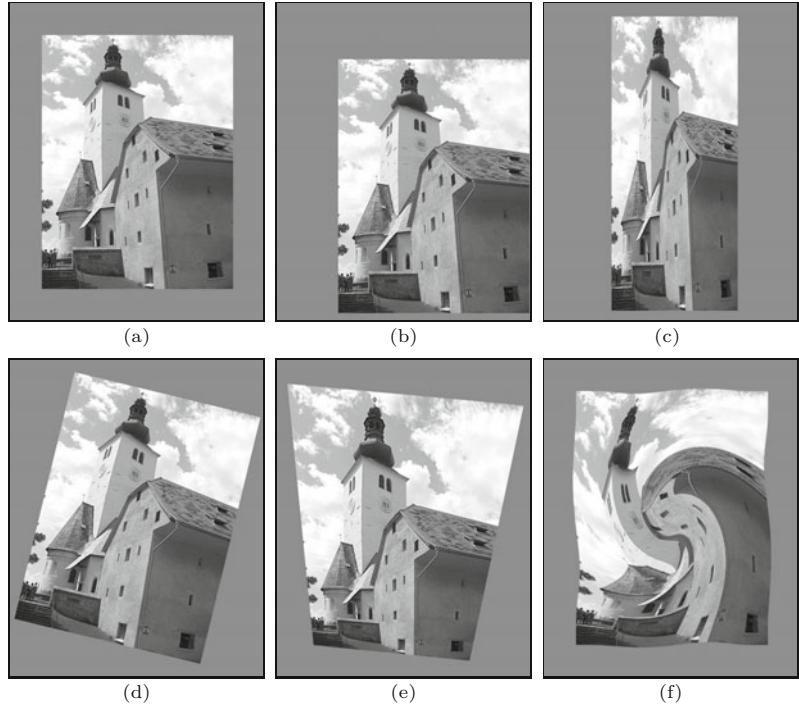
To model this process, we first need a 2D transformation function or *geometric mapping* T , for example, in the form

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2, \quad (21.2)$$

21 GEOMETRIC OPERATIONS

Fig. 21.1

Typical examples for geometric operations: original image (a), translation (b), scaling (contracting or stretching) in x and y directions (c), rotation about the center (d), projective transformation (e), and nonlinear distortion (f).



that specifies for each original 2D coordinate point $\mathbf{x} = (x, y)$ the corresponding target point $\mathbf{x}' = (x', y')$ in the new image I' ,

$$(x', y') = T(x, y). \quad (21.3)$$

Notice that the coordinates (x, y) and (x', y') specify points in the *continuous* image plane $\mathbb{R} \times \mathbb{R}$. The main problem in transforming digital images is that the pixels $I(u, v)$ are defined not on a continuous plane but on a *discrete* raster $\mathbb{Z} \times \mathbb{Z}$. Obviously, a transformed coordinate $(u', v') = T(u, v)$ produced by the mapping function $T()$ will, in general, no longer fall onto a discrete raster point. The solution to this problem is to compute intermediate pixel values for the transformed image by a process called *interpolation* (see Ch. 22), which is the second essential element in any geometric operation.

21.1 2D Coordinate Transformations

The mapping function $T()$ in Eqn. (21.3) is an arbitrary continuous function that for reasons of simplicity is often specified as two separate functions,

$$x' = T_x(x, y) \quad \text{and} \quad y' = T_y(x, y) \quad (21.4)$$

for the x and y components, respectively.

21.1.1 Simple Geometric Mappings

The simple mapping functions include translation, scaling, shearing, and rotation, defined as follows:

Translation (shift) by a vector (d_x, d_y) :

$$\begin{aligned} T_x : x' &= x + d_x & \text{or} & \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}. \end{aligned} \quad (21.5)$$

Scaling (contracting or stretching) along the x or y axis by the factor s_x or s_y , respectively:

$$\begin{aligned} T_x : x' &= s_x \cdot x & \text{or} & \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \end{aligned} \quad (21.6)$$

Shearing along the x and y axis by the factor b_x and b_y , respectively (for shearing in only one direction, the other factor is set to zero):

$$\begin{aligned} T_x : x' &= x + b_x \cdot y & \text{or} & \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \end{aligned} \quad (21.7)$$

Rotation by an angle α , with the coordinate origin being the center of rotation:

$$\begin{aligned} T_x : x' &= x \cdot \cos \alpha - y \cdot \sin \alpha & \text{or} & \quad \\ T_y : y' &= x \cdot \sin \alpha + y \cdot \cos \alpha & & \end{aligned} \quad (21.8)$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \quad (21.9)$$

Rotating the image by an angle α around an *arbitrary center point* $\mathbf{x}_c = (x_c, y_c)$ is accomplished by first translating the image by $(-x_c, -y_c)$, such that \mathbf{x}_c coincides with the origin, then rotating the image about the origin (as in Eqn. (21.9)), and finally shifting the image back by (x_c, y_c) . The resulting composite transformation is

$$\begin{aligned} T_x : x' &= x_c + (x - x_c) \cdot \cos \alpha - (y - y_c) \cdot \sin \alpha & \quad (21.10) \\ T_y : y' &= y_c + (x - x_c) \cdot \sin \alpha + (y - y_c) \cdot \cos \alpha \end{aligned}$$

or

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x - x_c \\ y - y_c \end{pmatrix}. \quad (21.11)$$

The combination of the operations listed in Eqns. (21.5)–(21.9) constitute the important class of “affine” transformations or *affine mappings* (see also Sec. 21.1.3).

21.1.2 Homogeneous Coordinates

To simplify the concatenation of linear mappings, it is advantageous to specify all operations in the form of vector-matrix multiplications, as in Eqns. (21.6)–(21.9). Note that pure translation Eqn. (21.5), which corresponds to a vector addition, cannot be formulated as a vector-matrix multiplication. Fortunately, this difficulty can be elegantly resolved with so-called *homogeneous coordinates* (see, e.g., [75, p. 204]).¹

¹ See also Sec. B.5 in the Appendix.

To turn an “ordinary” (i.e., *Cartesian*) coordinate into a homogeneous coordinate, the original vector is simply extended by an additional element with constant value 1. For example, a 2D Cartesian point $\mathbf{x} = (x, y)^\top$ converts to a 3D vector,

$$\text{hom}(\mathbf{x}) = \text{hom}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \underline{\mathbf{x}}. \quad (21.12)$$

Note that the homogeneous representation is not unique, but any scaled vector $s \cdot \underline{\mathbf{x}}$ is an equivalent homogeneous representation of the Cartesian coordinate \mathbf{x} , that is

$$\mathbf{x} = \text{hom}^{-1}(\underline{\mathbf{x}}) = \text{hom}^{-1}(s \cdot \underline{\mathbf{x}}), \quad (21.13)$$

for any nonzero $s \in \mathbb{R}$. For example, the homogeneous coordinates $\underline{\mathbf{x}}_1 = (3, 2, 1)^\top$, $\underline{\mathbf{x}}_2 = (-6, -4, -2)^\top$, and $\underline{\mathbf{x}}_3 = (30, 20, 10)^\top$ are all equivalent representations of the same Cartesian coordinate $\mathbf{x} = (3, 2)^\top$.

The reverse mapping from a 3D homogeneous coordinate $\underline{\mathbf{x}} = (\underline{x}, \underline{y}, \underline{z})^\top$ to the corresponding 2D Cartesian coordinate is denoted

$$\text{hom}^{-1}(\underline{\mathbf{x}}) = \text{hom}^{-1}\begin{pmatrix} \underline{x} \\ \underline{y} \\ \underline{z} \end{pmatrix} = \frac{1}{\underline{z}} \cdot \begin{pmatrix} \underline{x} \\ \underline{y} \end{pmatrix} = \mathbf{x} \quad (21.14)$$

With the help of homogeneous coordinates, we can now define a 2D *translation* (Eqn. (21.5)) as a vector-matrix product in the form

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \text{hom}^{-1}\left[\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \text{hom}\begin{pmatrix} x \\ y \end{pmatrix}\right] \quad (21.15)$$

$$= \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+d_x \\ y+d_y \\ 1 \end{pmatrix}, \quad (21.16)$$

which had been our motive for introducing homogeneous coordinates in the first place. As we shall see in the following sections, homogeneous coordinates allow us to write many common 2D coordinate transformations in the form

$$\underline{\mathbf{x}}' = \mathbf{A} \cdot \underline{\mathbf{x}}, \quad (21.17)$$

where \mathbf{A} is a 3×3 matrix. Note that (due to the relation in Eqn. (21.13)) multiplying the matrix \mathbf{A} by some scalar factor s yields the same transformation in terms of Cartesian coordinates, that is,

$$\mathbf{x}' = \text{hom}^{-1}[\mathbf{A} \cdot \underline{\mathbf{x}}] = \text{hom}^{-1}[s \cdot (\mathbf{A} \cdot \underline{\mathbf{x}})] = \text{hom}^{-1}[(s \cdot \mathbf{A}) \cdot \underline{\mathbf{x}}], \quad (21.18)$$

for any nonzero $s \in \mathbb{R}$.

21.1.3 Affine (Three-Point) Mapping

In general, and analogous to Eqn. (21.16), we can express any combination of 2D translation, scaling, and rotation as vector-matrix multiplication in homogeneous coordinates in the form

$$\underline{x}' = \mathbf{A}_{\text{affine}} \cdot \underline{x} \quad (21.19)$$

or $\mathbf{x}' = \text{hom}^{-1}[\mathbf{A}_{\text{affine}} \cdot \text{hom}(\mathbf{x})]$ in Cartesian coordinates, that is,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \text{hom}^{-1} \left[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \right] = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (21.20)$$

This 2D coordinate transformation is called an “affine mapping” with the six parameters a_{00}, \dots, a_{12} , where a_{02}, a_{12} specify the translation (equivalent to d_x, d_y in Eqn. (21.5)) and $a_{00}, a_{01}, a_{10}, a_{11}$ aggregate the scaling, shearing, and rotation coefficients (see Eqns. (21.6)–(21.9)). For example, the affine transformation matrix for a rotation about the origin by an angle α is specified by the matrix

$$\mathbf{A}_{\text{rot}} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (21.21)$$

In this way, compound transformations can be constructed easily by consecutive matrix multiplications (from right to left). For example, the transformation matrix for a rotation by α about a given center point $\mathbf{x}_c = (x_c, y_c)^\top$ (see Eqn. (21.11)), composed by a translation to the origin followed by a rotation and another translation, is

$$\mathbf{A} = \underbrace{\begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation by } (x_c, y_c)^\top} \cdot \underbrace{\begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{rotation by } \alpha \text{ (about the origin)}} \cdot \underbrace{\begin{pmatrix} 1 & 0 & -x_c \\ 0 & 1 & -y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation by } (-x_c, -y_c)^\top} \quad (21.22)$$

$$= \begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix}^{-1} \quad (21.23)$$

$$= \begin{pmatrix} \cos \alpha & -\sin \alpha & x_c \cdot (1 - \cos \alpha) + y_c \cdot \sin \alpha \\ \sin \alpha & \cos \alpha & y_c \cdot (1 - \cos \alpha) - x_c \cdot \sin \alpha \\ 0 & 0 & 1 \end{pmatrix}. \quad (21.24)$$

Of course, the result is the same as in Eqn. (21.10).

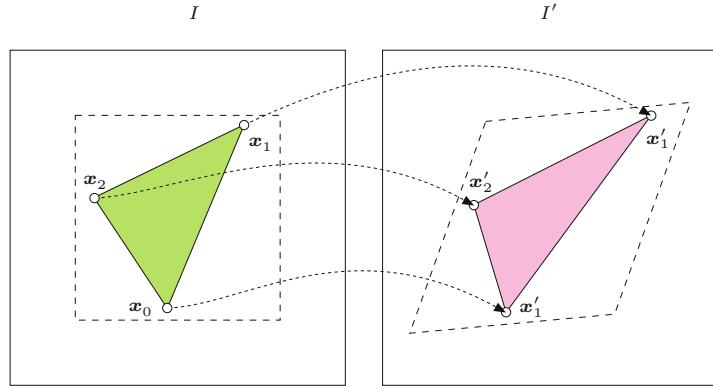
Note that multiplying two affine transformation matrices always yields another affine transformation. Also, an affine transformation maps straight lines to straight lines, triangles to triangles, and rectangles to parallelograms, as illustrated in Fig. 21.2. The distance ratio between points on a straight line remains unchanged by this type of mapping function.

Affine transformation parameters from three point pairs

The six parameters of the 2D affine mapping (Eqn. (21.20)) are uniquely determined by three pairs of corresponding points $(\mathbf{x}_0, \mathbf{x}'_0)$, $(\mathbf{x}_1, \mathbf{x}'_1)$, $(\mathbf{x}_2, \mathbf{x}'_2)$, with the first point $\mathbf{x}_i = (x_i, y_i)$ of each pair located in the original image and the corresponding point $\mathbf{x}'_i = (x'_i, y'_i)$ located in the target image. From these six coordinate values, the

Fig. 21.2

Affine mapping. An affine 2D transformation is uniquely specified by three pairs of corresponding points; for example, $(\mathbf{x}_0, \mathbf{x}'_0)$, $(\mathbf{x}_1, \mathbf{x}'_1)$, and $(\mathbf{x}_2, \mathbf{x}'_2)$.



six transformation parameters a_{00}, \dots, a_{12} are derived by solving the system of linear equations

$$\begin{aligned} x'_0 &= a_{00} \cdot x_0 + a_{01} \cdot y_0 + a_{02}, & y'_0 &= a_{10} \cdot x_0 + a_{11} \cdot y_0 + a_{12}, \\ x'_1 &= a_{00} \cdot x_1 + a_{01} \cdot y_1 + a_{02}, & y'_1 &= a_{10} \cdot x_1 + a_{11} \cdot y_1 + a_{12}, \\ x'_2 &= a_{00} \cdot x_2 + a_{01} \cdot y_2 + a_{02}, & y'_2 &= a_{10} \cdot x_2 + a_{11} \cdot y_2 + a_{12}, \end{aligned} \quad (21.25)$$

provided that the points (vectors) $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$ are linearly independent (i.e., that they do not lie on a common straight line). Since Eqn. (21.25) consists of two independent sets of linear 3×3 equations for x'_i and y'_i , the solution can be written in closed form as

$$\begin{aligned} a_{00} &= \frac{1}{d} \cdot [y_0(x'_1 - x'_2) &+ y_1(x'_2 - x'_0) &+ y_2(x'_0 - x'_1)], \\ a_{01} &= \frac{1}{d} \cdot [x_0(y'_1 - y'_2) &+ x_1(y'_2 - y'_0) &+ x_2(y'_0 - y'_1)], \\ a_{10} &= \frac{1}{d} \cdot [y_0(y'_1 - y'_2) &+ y_1(y'_2 - y'_0) &+ y_2(y'_0 - y'_1)], \\ a_{11} &= \frac{1}{d} \cdot [x_0(y'_1 - y'_2) &+ x_1(y'_2 - y'_0) &+ x_2(y'_0 - y'_1)], \\ a_{02} &= \frac{1}{d} \cdot [x_0(y_2 x'_1 - y_1 x'_2) &+ x_1(y_0 x'_2 - y_2 x'_0) &+ x_2(y_1 x'_0 - y_0 x'_1)], \\ a_{12} &= \frac{1}{d} \cdot [x_0(y_2 y'_1 - y_1 y'_2) &+ x_1(y_0 y'_2 - y_2 y'_0) &+ x_2(y_1 y'_0 - y_0 y'_1)], \end{aligned} \quad (21.26)$$

with $d = x_0(y_2 - y_1) + x_1(y_0 - y_2) + x_2(y_1 - y_0)$.

Inverse affine mapping

The inverse of the affine transformation, which is often required in practice (see Sec. 21.2.2), can be calculated by simply applying the inverse of the transformation matrix $\mathbf{A}_{\text{affine}}$ (Eqn. (21.20)) in homogeneous coordinate space, that is,

$$\underline{\mathbf{x}} = \mathbf{A}_{\text{affine}}^{-1} \cdot \underline{\mathbf{x}}' \quad (21.27)$$

or $\mathbf{x} = \text{hom}^{-1} [\mathbf{A}_{\text{affine}}^{-1} \cdot \text{hom}(\mathbf{x}')]$ in Cartesian coordinates, that is,

$$\begin{pmatrix} x \\ y \end{pmatrix} = \text{hom}^{-1} \left[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \right] \quad (21.28)$$

$$= \text{hom}^{-1} \left[\underbrace{\frac{1}{a_{00}a_{11}-a_{01}a_{10}} \cdot \begin{pmatrix} a_{11} & -a_{01} & a_{01}a_{12}-a_{02}a_{11} \\ -a_{10} & a_{00} & a_{02}a_{10}-a_{00}a_{12} \\ 0 & 0 & a_{00}a_{11}-a_{01}a_{10} \end{pmatrix}}_{\mathbf{A}_{\text{affine}}^{-1}} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \right] \quad (21.29)$$

$$= \frac{1}{a_{00}a_{11}-a_{01}a_{10}} \cdot \begin{pmatrix} a_{11} & -a_{01} & a_{01}a_{12}-a_{02}a_{11} \\ -a_{10} & a_{00} & a_{02}a_{10}-a_{00}a_{12} \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}. \quad (21.30)$$

Since the bottom row of $\mathbf{A}_{\text{affine}}^{-1}$ in Eqn. (21.29) consists of the elements $(0, 0, 1)$, the inverse mapping is again an affine transformation. Of course, the inverse of the affine mapping can also be found directly (i.e., without inverting the transformation matrix) from the given point coordinates $(\mathbf{x}_i, \mathbf{x}'_i)$ by using Eqns. (21.25) and (21.26) with *interchanged* source and target coordinates.

21.1.4 Projective (Four-Point) Mapping

In contrast to the affine transformation, which provides a mapping between arbitrary triangles, the projective transformation is a linear mapping between arbitrary *quadrilaterals* (Fig. 21.3). This is particularly useful for deforming images controlled by mesh partitioning, as described in Sec. 21.1.7. To map from an arbitrary sequence of four 2D points $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ to a set of corresponding points $(\mathbf{x}'_0, \mathbf{x}'_1, \mathbf{x}'_2, \mathbf{x}'_3)$, the transformation requires eight degrees of freedom, two more than needed for the affine transformation. Analogous to the affine transformation (Eqn. (21.20)), a projective transformation can be expressed as a linear mapping in homogeneous coordinates,

$$\underline{\mathbf{x}}' = \mathbf{A}_{\text{proj}} \cdot \underline{\mathbf{x}} \quad (21.31)$$

or $\mathbf{x}' = \text{hom}^{-1}[\mathbf{A}_{\text{proj}} \cdot \text{hom}(\mathbf{x})]$ in Cartesian coordinates, that is,

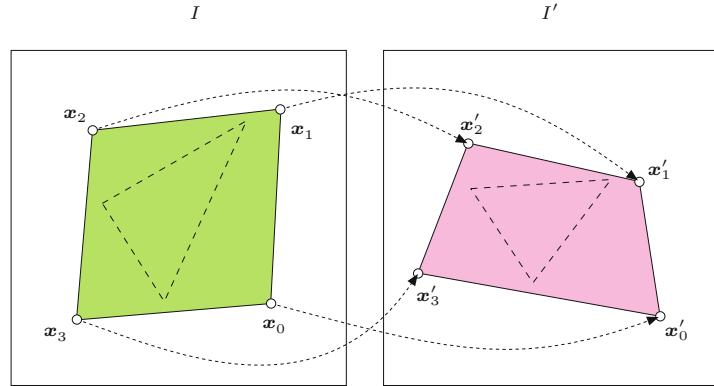
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \text{hom}^{-1} \left[\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \right] \quad (21.32)$$

$$= \frac{1}{a_{20} \cdot x + a_{21} \cdot y + 1} \cdot \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (21.33)$$

with the two additional elements (parameters) a_{20} and a_{21} in the transformation matrix \mathbf{A}_{proj} . Because x, y appear in the denominator of the fraction in Eqn. (21.33), the projective mapping is generally *nonlinear* in Cartesian coordinates. Despite this nonlinearity, straight lines are preserved under this transformation. In fact, this is the most general transformation that maps straight lines to straight lines in 2D, and it actually maps any N th-order algebraic curve onto another N th-order algebraic curve. In particular, circles and ellipses always transform into other second-order curves (i.e., conic sections). Unlike the affine transformation, however, parallel lines do not generally map to parallel lines under a projective transformation (cf. Fig.

Fig. 21.3

Projective mapping. Four pairs of corresponding 2D points, $(\mathbf{x}_0, \mathbf{x}'_0), (\mathbf{x}_1, \mathbf{x}'_1), (\mathbf{x}_2, \mathbf{x}'_2), (\mathbf{x}_3, \mathbf{x}'_3)$ uniquely specify a projective transformation. Straight lines are again mapped to straight lines, and a rectangle is mapped to some quadrilateral. In general, neither parallelism between straight lines nor the distance ratio is preserved.



21.3) and the distance ratios between points on a line are not preserved. The projective mapping is therefore sometimes referred to as “perspective” or “pseudo-perspective”.

Projective transformation parameters from four point pairs

Given four pairs of corresponding 2D points, $(\mathbf{x}_0, \mathbf{x}'_0), \dots, (\mathbf{x}_3, \mathbf{x}'_3)$, with one point $\mathbf{x}_i = (x_i, y_i)^\top$ in the source image and the second point $\mathbf{x}'_i = (x'_i, y'_i)^\top$ in the target image, the eight unknown transformation parameters a_{00}, \dots, a_{21} can be found by solving a system of linear equations. Multiplying Eqn. (21.33) by the common denominator on the right hand side gives

$$\begin{aligned} x' \cdot (a_{20} \cdot x + a_{21} \cdot y + 1) &= a_{00} \cdot x + a_{01} \cdot y + a_{02}, \\ y' \cdot (a_{20} \cdot x + a_{21} \cdot y + 1) &= a_{10} \cdot x + a_{11} \cdot y + a_{12}, \end{aligned} \quad (21.34)$$

and thus

$$\begin{aligned} a_{20} \cdot x \cdot x' + a_{21} \cdot y \cdot x' + x' &= a_{00} \cdot x + a_{01} \cdot y + a_{02}, \\ a_{20} \cdot x \cdot y' + a_{21} \cdot y \cdot y' + y' &= a_{10} \cdot x + a_{11} \cdot y + a_{12}, \end{aligned} \quad (21.35)$$

for any pair of corresponding points $\mathbf{x} = (x, y)^\top$ and $\mathbf{x}' = (x', y')^\top$. By slightly rearranging Eqn. (21.35) and inserting the (known) source and target point coordinates (x_i, y_i) and (x'_i, y'_i) , respectively, we obtain one such pair of linear equations

$$\begin{aligned} x'_i &= a_{00} \cdot x_i + a_{01} \cdot y_i + a_{02} - a_{20} \cdot x_i \cdot x'_i - a_{21} \cdot y_i \cdot x'_i, \\ y'_i &= a_{10} \cdot x_i + a_{11} \cdot y_i + a_{12} - a_{20} \cdot x_i \cdot y'_i - a_{21} \cdot y_i \cdot y'_i, \end{aligned} \quad (21.36)$$

for each point pair $i = 0, \dots, 3$ and the eight unknowns a_{00}, \dots, a_{21} . Combining the resulting eight equations in the usual matrix notation yields

$$\begin{pmatrix} x'_0 \\ y'_0 \\ x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0 x'_0 & -y_0 x'_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -x_0 y'_0 & -y_0 y'_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1 x'_1 & -y_1 x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1 y'_1 & -y_1 y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2 x'_2 & -y_2 x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2 y'_2 & -y_2 y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3 x'_3 & -y_3 x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3 y'_3 & -y_3 y'_3 \end{pmatrix} \cdot \begin{pmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \end{pmatrix}, \quad (21.37)$$

or

$$\mathbf{b} = \mathbf{M} \cdot \mathbf{a}. \quad (21.38)$$

Note that all elements of the vector $\mathbf{b} = (x'_0, \dots, y'_3)^\top$ and the matrix \mathbf{M} are obtained from the specified point coordinates and are thus constants. The unknown parameters $\mathbf{a} = (a_{00}, \dots, a_{21})^\top$ can be found by solving the system of linear equations in Eqn. (21.38) with standard numerical methods, for example, the Gauss algorithm [35, p. 276]. It is recommended to use proven numerical software for this purpose.²

If we want to use *more than four* corresponding point pairs to recover the eight parameters of a projective transformation, the system of linear equations in Eqn. (21.37) becomes overdetermined, that is, the system has more equations than unknowns. In general, n pairs of corresponding points yield a stack of $2n$ equations, so the vector \mathbf{b} in Eqn. (21.37) has the length $2n$ and the matrix \mathbf{M} is of size $2n \times 8$ (vector \mathbf{a} remains the same). Overdetermined systems like this can be solved in a least-squares sense (minimizing $\|\mathbf{M} \cdot \mathbf{a} - \mathbf{b}\|$), for example, using the singular-value (SVD) or QR decomposition of \mathbf{M} [96, 145].³ Other solutions for the multi-point case are discussed later in this section (see p. 524).

Inverse projective mapping

In general, any *linear* transformation of the form $\underline{x}' = \mathbf{A} \cdot \underline{x}$ (in homogeneous coordinates \underline{x} , \underline{x}') can be inverted by applying the inverse of the matrix \mathbf{A} , that is,

$$\underline{x} = \mathbf{A}^{-1} \cdot \underline{x}' \quad (21.39)$$

provided that \mathbf{A} is nonsingular ($\det(\mathbf{A}) \neq 0$). The inverse of a 3×3 matrix \mathbf{A} is comparatively easy to find in closed form using the relation

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \cdot \text{adj}(\mathbf{A}), \quad (21.40)$$

with the determinant $\det(\mathbf{A})$ and the *adjugate* matrix $\text{adj}(\mathbf{A})$ (see, e.g., [35, pp. 251, 260], [145, p. 219]). In particular, for a real-valued 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}, \quad (21.41)$$

the determinant can be calculated as

$$\begin{aligned} \det(\mathbf{A}) = & a_{00} a_{11} a_{22} + a_{01} a_{12} a_{20} + a_{02} a_{10} a_{21} \\ & - a_{00} a_{12} a_{21} - a_{01} a_{10} a_{22} - a_{02} a_{11} a_{20}, \end{aligned} \quad (21.42)$$

and the 3×3 adjugate matrix is

$$\text{adj}(\mathbf{A}) = \begin{pmatrix} a_{11}a_{22} - a_{12}a_{21} & a_{02}a_{21} - a_{01}a_{22} & a_{01}a_{12} - a_{02}a_{11} \\ a_{12}a_{20} - a_{10}a_{22} & a_{00}a_{22} - a_{02}a_{20} & a_{02}a_{10} - a_{00}a_{12} \\ a_{10}a_{21} - a_{11}a_{20} & a_{01}a_{20} - a_{00}a_{21} & a_{00}a_{11} - a_{01}a_{10} \end{pmatrix}. \quad (21.43)$$

² See Sec. B.7.1 in the Appendix.

³ See Sec. B.7.2 in the Appendix.

In the special case of a projective mapping, the coefficient $a_{22} = 1$ (Eqn. (21.32)), which slightly simplifies the calculation.

Since scalar multiples of homogeneous vectors are all equivalent in Cartesian space (see Eqn. (21.18)), the multiplication by the constant factor $1/\det(\mathbf{A})$ in Eqn. (21.40) can be omitted. Thus, to invert a linear 2D transformation specified by a 3×3 matrix \mathbf{A} , we only need to multiply the homogeneous coordinate vector with the adjugate matrix $\text{adj}(\mathbf{A})$, that is,

$$\underline{x} = \mathbf{A}^{-1} \cdot \underline{x}' \equiv \text{adj}(\mathbf{A}) \cdot \underline{x}'. \quad (21.44)$$

Returning to Cartesian coordinates, the inverse transformation can be written as

$$\mathbf{x} = \text{hom}^{-1}[\text{adj}(\mathbf{A}) \cdot \text{hom}(\mathbf{x}')]. \quad (21.45)$$

This method can be used to invert any linear transformation in 2D, including the affine and projective mapping functions described already. Consequently, the inversion of the *affine* transformation shown earlier (see Eqn. (21.29)) is only a special case of this general method.

Of course, matrix inversion may also be implemented with standard linear algebra software, which is not only less error-prone but also offers better numerical stability (see also Sec. B.1 in the Appendix).

Projective mapping via the unit square

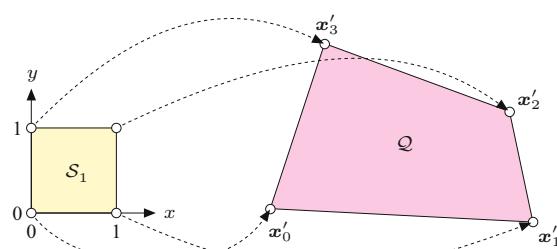
An alternative method for finding the projective mapping parameters for a given set of image points is to use a two-stage mapping through the unit square \mathcal{S}_1 , which avoids iteratively solving a system of equations [256, p. 55] [105]. The projective mapping, shown in Fig. 21.4, from the four corner points of the unit square \mathcal{S}_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_0, \dots, \mathbf{x}'_3)$ with

$$\begin{aligned} (0, 0) &\mapsto \mathbf{x}'_0, & (1, 1) &\mapsto \mathbf{x}'_2, \\ (1, 0) &\mapsto \mathbf{x}'_1, & (0, 1) &\mapsto \mathbf{x}'_3, \end{aligned} \quad (21.46)$$

reduces the system of equations in Eqn. (21.37) to

Fig. 21.4

Projective mapping from the unit square \mathcal{S}_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_0, \dots, \mathbf{x}'_3)$.



$$\begin{aligned}
x'_0 &= a_{02}, \\
y'_0 &= a_{12}, \\
x'_1 &= a_{00} + a_{02} - a_{20} \cdot x'_1, \\
y'_1 &= a_{10} + a_{12} - a_{20} \cdot y'_1, \\
x'_2 &= a_{00} + a_{01} + a_{02} - a_{20} \cdot x'_2 - a_{21} \cdot x'_2, \\
y'_2 &= a_{10} + a_{11} + a_{12} - a_{20} \cdot y'_2 - a_{21} \cdot y'_2, \\
x'_3 &= a_{01} + a_{02} - a_{21} \cdot x'_3, \\
y'_3 &= a_{11} + a_{12} - a_{21} \cdot y'_3.
\end{aligned} \tag{21.47}$$

This set of equations has the following closed-form solution for the eight unknown transformation parameters $a_{00}, a_{01}, \dots, a_{21}$:

$$a_{20} = \frac{(x'_0 - x'_1 + x'_2 - x'_3) \cdot (y'_3 - y'_2) - (y'_0 - y'_1 + y'_2 - y'_3) \cdot (x'_3 - x'_2)}{(x'_1 - x'_2) \cdot (y'_3 - y'_2) - (x'_3 - x'_2) \cdot (y'_1 - y'_2)}, \tag{21.48}$$

$$a_{21} = \frac{(y'_0 - y'_1 + y'_2 - y'_3) \cdot (x'_1 - x'_2) - (x'_0 - x'_1 + x'_2 - x'_3) \cdot (y'_1 - y'_2)}{(x'_1 - x'_2) \cdot (y'_3 - y'_2) - (x'_3 - x'_2) \cdot (y'_1 - y'_2)} \tag{21.49}$$

and

$$a_{00} = x'_1 - x'_0 + a_{20} x'_1, \quad a_{01} = x'_3 - x'_0 + a_{21} x'_3, \quad a_{02} = x'_0, \tag{21.50}$$

$$a_{10} = y'_1 - y'_0 + a_{20} y'_1, \quad a_{11} = y'_3 - y'_0 + a_{21} y'_3, \quad a_{12} = y'_0. \tag{21.51}$$

By calculating the inverse of the corresponding 3×3 transformation matrix (Eqn. (21.40)), the mapping may be *reversed* to transform an arbitrary quadrilateral to the unit square. A mapping T between two arbitrary quadrilaterals,

$$\mathcal{Q} \xrightarrow{T} \mathcal{Q}',$$

can thus be implemented by combining a reversed mapping and a forward mapping via the unit square. As illustrated in Fig. 21.5, the transformation of an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ to a second quadrilateral $\mathcal{Q}' = (\mathbf{x}'_0, \mathbf{x}'_1, \mathbf{x}'_2, \mathbf{x}'_3)$ is accomplished in two steps involving the linear transformations T_1 and T_2 between the two quadrilaterals and the unit square \mathcal{S}_1 , that is,

$$\mathcal{Q} \xleftarrow{T_1} \mathcal{S}_1 \xrightarrow{T_2} \mathcal{Q}'. \tag{21.52}$$

The parameters for the projective transformations T_1 and T_2 are obtained by inserting the corresponding point coordinates of \mathcal{Q} and \mathcal{Q}' (\mathbf{x}_i and \mathbf{x}'_i , respectively) into Eqns. (21.48)–(21.51). The complete transformation T is then the concatenation of the two transformations T_1^{-1} and T_2 , that is,

$$\mathbf{x}' = T(\mathbf{x}) = T_2(T_1^{-1}(\mathbf{x})), \tag{21.53}$$

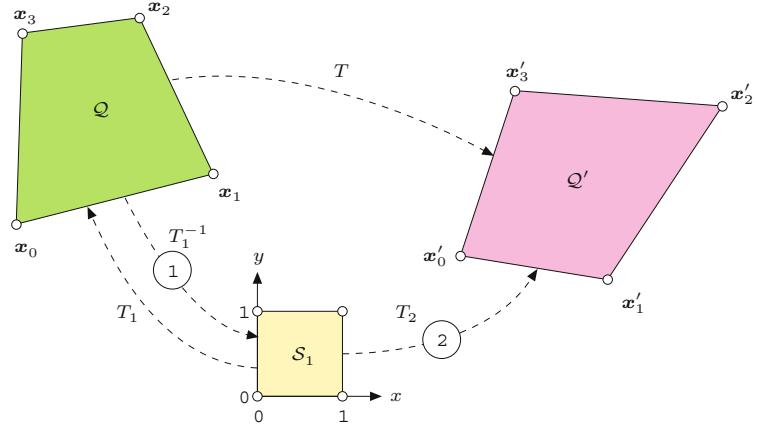
or, expressed in matrix notation (using homogeneous coordinates),

$$\underline{\mathbf{x}}' = \mathbf{A} \cdot \underline{\mathbf{x}} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1} \cdot \underline{\mathbf{x}}. \tag{21.54}$$

Of course, the matrix $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ needs to be calculated only once for a particular transformation and can then be used repeatedly for mapping any other image points \mathbf{x}_i .

Fig. 21.5

Two-step projective transformation between arbitrary quadrilaterals. In the first step, quadrilateral \mathcal{Q} is transformed to the unit square S_1 by the inverse mapping function T_1^{-1} . In the second step, T_2 transforms the square S_1 to the target quadrilateral \mathcal{Q}' . The complete mapping T results from the concatenation of the mappings T_1^{-1} and T_2 .



Example

The source and the target quadrilaterals \mathcal{Q} and \mathcal{Q}' , respectively, are specified by the following coordinate points:

$$\begin{aligned}\mathcal{Q} : \quad & x_0 = (2, 5), \quad x_1 = (4, 6), \quad x_2 = (7, 9), \quad x_3 = (5, 9); \\ \mathcal{Q}' : \quad & x'_0 = (4, 3), \quad x'_1 = (5, 2), \quad x'_2 = (9, 3), \quad x'_3 = (7, 5).\end{aligned}$$

Using Eqns. (21.48)–(21.51), the transformation parameters (matrices) for the projective mappings from the unit S_1 square to the quadrilaterals $\mathbf{A}_1 : S_1 \mapsto \mathcal{Q}$ and $\mathbf{A}_2 : S_1 \mapsto \mathcal{Q}'$ are obtained as

$$\mathbf{A}_1 = \begin{pmatrix} 3.33 & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.33 & -0.50 & 1.00 \end{pmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}.$$

Concatenating the inverse mapping \mathbf{A}_1^{-1} with \mathbf{A}_2 (by matrix multiplication), we get the complete mapping $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ with

$$\mathbf{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \text{and} \quad \mathbf{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}.$$

The library method `makeMapping()` in class `ProjectiveMapping` (see Sec. 21.3) is an implementation of this two-step technique.

Projective transformation parameters from more than four point pairs

The projective transformation in Eqn. (21.32) describes a mapping between pairs of arbitrary quadrilaterals in the 2D plane. This geometric relation is also known under the terms *projective isomorphism* or *homography*. The concept is frequently encountered in computer vision, because the transformations between two views of a planar 3D point set can be modeled as a homography (with only 8 degrees of freedom) in the 2D image plane, which is important, for example, for camera calibration, and 3D surface reconstruction. In this context, it is often necessary to estimate the homography parameters from a larger set of 2D point matches, for example, from multiple points

assumed to be located on a planar 3D surface. This is the same problem as finding the projective mapping between sets of $n > 4$ corresponding point pairs in 2D.

21.1 2D COORDINATE TRANSFORMATIONS

Several approaches to “homography estimation” exist, including linear and (iterative) nonlinear methods. The simplest and most common is the direct linear transform (DLT) method [56,103], which requires solving a system of $2n$ homogenous linear equations, typically done by singular value decomposition (SVD).

21.1.5 Bilinear Mapping

Similar to the projective transformation (Eqn. (21.32)), the bilinear mapping function

$$\begin{aligned} T_x : x' &= a_0 \cdot x + a_1 \cdot y + a_2 \cdot x \cdot y + a_3, \\ T_y : y' &= b_0 \cdot x + b_1 \cdot y + b_2 \cdot x \cdot y + b_3, \end{aligned} \quad (21.55)$$

is specified with four pairs of corresponding points and has eight parameters $(a_0, \dots, a_3, b_0, \dots, b_3)$. The transformation is nonlinear because of the mixed term $x \cdot y$ and cannot be described by a linear transformation, even with homogeneous coordinates. In contrast to the projective transformation, the straight lines are not preserved in general but map onto quadratic curves. Similarly, circles are not mapped to ellipses by a bilinear transform.

A bilinear mapping is uniquely specified by four corresponding pairs of 2D points $(\mathbf{x}_0, \mathbf{x}'_0), \dots, (\mathbf{x}_3, \mathbf{x}'_3)$. In the general case, for a bilinear mapping between arbitrary quadrilaterals, the coefficients $a_0, \dots, a_3, b_0, \dots, b_3$ (Eqn. (21.55)) are found as the solution of two separate systems of equations, each with four unknowns:

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & x_0 \cdot y_0 & 1 \\ x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \quad \text{or } \mathbf{x} = \mathbf{M} \cdot \mathbf{a}, \quad (21.56)$$

$$\begin{pmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & x_0 \cdot y_0 & 1 \\ x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad \text{or } \mathbf{y} = \mathbf{M} \cdot \mathbf{b}. \quad (21.57)$$

These equations can again be solved using standard numerical methods. In the special case of bilinearly mapping the unit square \mathcal{S}_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_0, \dots, \mathbf{x}'_3)$, the parameters a_0, \dots, a_3 and b_0, \dots, b_3 are found as

$$a_0 = x'_1 - x'_0, \quad b_0 = y'_1 - y'_0, \quad (21.58)$$

$$a_1 = x'_3 - x'_0, \quad b_1 = y'_3 - y'_0, \quad (21.59)$$

$$a_2 = x'_0 - x'_1 + x'_2 - x'_3, \quad b_2 = y'_0 - y'_1 + y'_2 - y'_3, \quad (21.60)$$

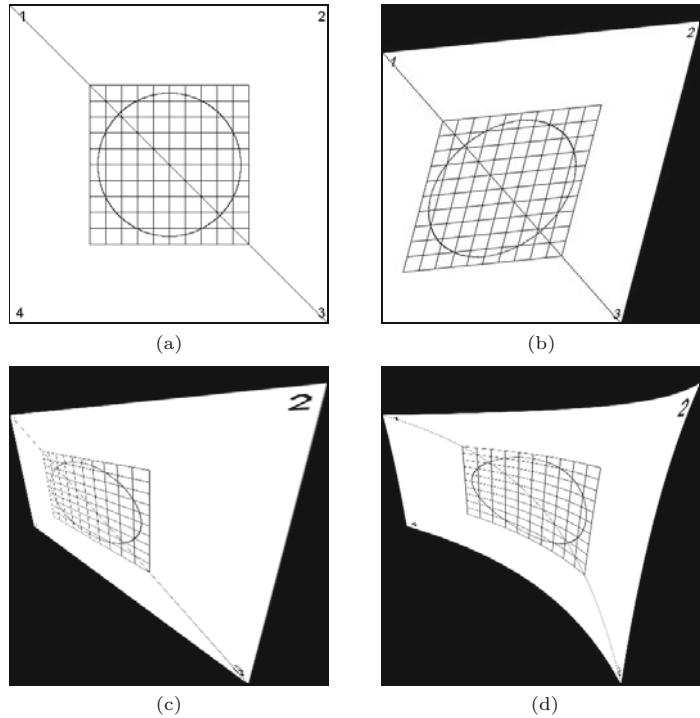
$$a_3 = x'_0, \quad b_3 = y'_0. \quad (21.61)$$

[Figure 21.6](#) shows results of the affine, projective, and bilinear transformations applied to a simple test pattern. The affine transformation ([Fig. 21.6\(b\)](#)) is specified by mapping to the triangle 1-2-3, while the four points of the quadrilateral 1-2-3-4 define the projective and the bilinear transforms ([Fig. 21.6\(c,d\)](#)).

21 GEOMETRIC OPERATIONS

Fig. 21.6

Geometric transformations compared: original image (a), affine transformation with respect to the triangle 1-2-3 (b), projective transformation (c), and bilinear transformation (d).



21.1.6 Other Nonlinear Image Transformations

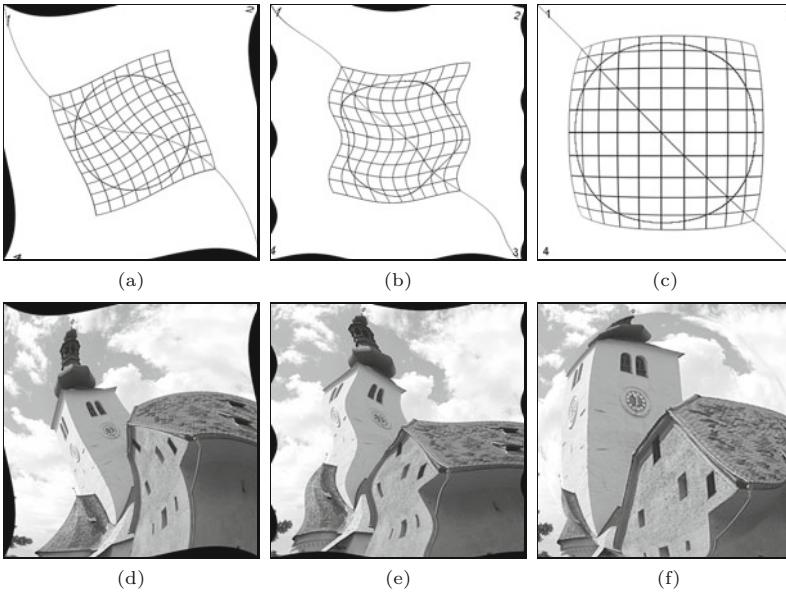
The bilinear transformation discussed in the previous section is only one example of a nonlinear mapping in 2D that cannot be expressed as a simple matrix-vector multiplication in homogeneous coordinates. Many other types of nonlinear deformations exist; for example, to implement various artistic effects for creative imaging. This type of image deformation is often called “image warping”. Depending on the type of transformation used, the derivation of the *inverse* transformation function—which is required for the practical computation of the mapping using *target-to-source mapping* (see Sec. 21.2.2)—is not always easy or may even be impossible. In the following three examples, we therefore look straight at the inverse maps

$$\mathbf{x} = T^{-1}(\mathbf{x}') \quad (21.62)$$

without really bothering about the corresponding *forward* transformations.

“Twirl” transformation

The twirl mapping causes the image to be rotated around a given anchor point $\mathbf{x}_c = (x_c, y_c)$ with a space-variant rotation angle, which has a fixed value α at the center \mathbf{x}_c and decreases linearly with the radial distance from the center. The image remains unchanged outside the limiting radius r_{\max} . The associated (*inverse*) mapping is defined as



21.1 2D COORDINATE TRANSFORMATIONS

Fig. 21.7

Various nonlinear image deformations: *twirl* (a, d), *ripple* (b, e), and *sphere* (c, f) transformations. The size of the original images is 400×400 pixels.

(a)

(b)

(c)

(d)

(e)

(f)

$$T_x^{-1}: x = \begin{cases} x_c + r \cdot \cos(\beta) & \text{for } r \leq r_{\max}, \\ x' & \text{for } r > r_{\max}, \end{cases} \quad (21.63)$$

$$T_y^{-1}: y = \begin{cases} y_c + r \cdot \sin(\beta) & \text{for } r \leq r_{\max}, \\ y' & \text{for } r > r_{\max}, \end{cases} \quad (21.64)$$

with

$$r = \sqrt{d_x^2 + d_y^2}, \quad d_x = x' - x_c, \quad (21.65)$$

$$\beta = \text{ArcTan}(d_x, d_y) + \alpha \cdot \left(\frac{r_{\max} - r}{r_{\max}} \right), \quad d_y = y' - y_c. \quad (21.66)$$

Figure 21.7(a, d) shows a twirl mapping with the anchor point x_c placed at the image center. The limiting radius r_{\max} is half the length of the image diagonal, and the rotation angle is $\alpha = 43^\circ$ at the center.

“Ripple” transformation

The ripple transformation causes a local wavelike displacement of the image along both the x and y directions. The parameters of this mapping function are the period lengths $\tau_x, \tau_y \neq 0$ (in pixels) and the corresponding amplitude values a_x, a_y for the displacement in both directions:

$$T_x^{-1}: x = x' + a_x \cdot \sin\left(\frac{2\pi \cdot y'}{\tau_x}\right), \quad (21.67)$$

$$T_y^{-1}: y = y' + a_y \cdot \sin\left(\frac{2\pi \cdot x'}{\tau_y}\right). \quad (21.68)$$

An example for the ripple mapping with $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$, and $a_y = 15$ is shown in Fig. 21.7(b, e).

Spherical transformation

The spherical deformation imitates the effect of viewing the image through a transparent hemisphere or lens placed on top of the image.

The parameters of this transformation are the position $\mathbf{x}_c = (x_c, y_c)$ of the lens center, the radius of the lens r_{\max} and its refraction index ρ . The corresponding mapping functions are defined as

$$T_x^{-1}: x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{for } r \leq r_{\max}, \\ 0 & \text{for } r > r_{\max}, \end{cases} \quad (21.69)$$

$$T_y^{-1}: y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{for } r \leq r_{\max}, \\ 0 & \text{for } r > r_{\max}, \end{cases} \quad (21.70)$$

with

$$\begin{aligned} r &= \sqrt{d_x^2 + d_y^2}, & \beta_x &= \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_x}{\sqrt{(d_x^2 + z^2)}}\right), & d_x &= x' - x_c, \\ z &= \sqrt{r_{\max}^2 - r^2}, & \beta_y &= \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_y}{\sqrt{(d_y^2 + z^2)}}\right), & d_y &= y' - y_c. \end{aligned} \quad (21.71)$$

[Figure 21.7\(c, f\)](#) shows a spherical transformation with the lens positioned at the image center. The lens radius r_{\max} is set to half of the image width, and the refraction index is $\rho = 1.8$.

See Exercise 21.4 for additional examples of nonlinear geometric transformations.

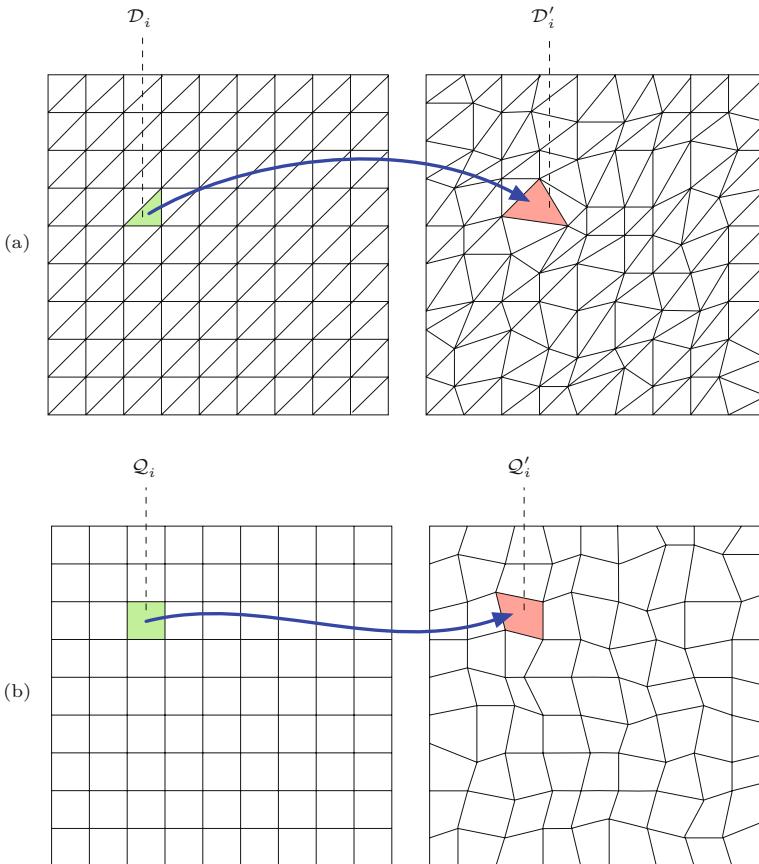
21.1.7 Piecewise Image Transformations

All the geometric transformations discussed so far are *global* (i.e., the same mapping function is applied to all pixels in the given image). It is often necessary to deform an image such that a larger number of n original image points $\mathbf{x}_0, \dots, \mathbf{x}_n$ are precisely mapped onto a given set of target points $\mathbf{x}'_0, \dots, \mathbf{x}'_n$. For $n = 3$, this problem can be solved with an affine mapping (see Sec. 21.1.3), and for $n = 4$ we could use a projective or bilinear mapping (see Secs. 21.1.4 and 21.1.5). A precise global mapping of $n > 4$ points requires a more complicated function $T(\mathbf{x})$ (e.g., a 2D n th-order polynomial or a spline function).

An alternative is to use *local* or *piecewise* transformations, where the image is partitioned into disjoint patches that are transformed separately, applying an individual mapping function to each patch. In practice, it is common to partition the image into a *mesh* of triangles or quadrilaterals, as illustrated in [Fig. 21.8](#).

For a *triangular* mesh partitioning ([Fig. 21.8\(a\)](#)), the transformation between each pair of triangles $\mathcal{D}_i \rightarrow \mathcal{D}'_i$ could be accomplished with an *affine* mapping, whose parameters must be computed individually for every patch. Similarly, the *projective* transformation would be suitable for mapping each patch in a mesh partitioning composed of *quadrilaterals* \mathcal{Q}_i ([Fig. 21.8\(b\)](#)). Since both the affine and the projective transformations preserve the straightness of lines, we can be certain that no holes or overlaps will arise and the deformation will appear continuous between adjacent mesh patches.

Local transformations of this type are frequently used; for example, to register aerial and satellite images or to undistort images for panoramic stitching. In computer graphics, similar techniques are used to map texture images onto polygonal 3D surfaces in the rendered 2D image. Another popular application of this technique is



21.2 RESAMPLING THE IMAGE

Fig. 21.8

Mesh partitioning examples. Almost arbitrary image deformations can be implemented by partitioning the image plane into nonoverlapping triangles $\mathcal{D}_i, \mathcal{D}'_i$ (a) or quadrilaterals $\mathcal{Q}_i, \mathcal{Q}'_i$ (b) and applying simple local transformations. Every patch in the resulting mesh is transformed separately with the required transformation parameters derived from the corresponding three or four corner points, respectively.

“morphing” [256], which performs a stepwise geometric transformation from one image to another while simultaneously blending their intensity (or color) values.⁴

21.2 Resampling the Image

In the discussion of geometric transformations, we have so far considered the 2D image coordinates as being continuous (i.e., real-valued). In reality, the picture elements in digital images reside at discrete (i.e., integer-valued) coordinates, and thus transferring a discrete image into another discrete image without introducing significant losses in quality is a nontrivial subproblem in the implementation of geometric transformations.

Based on the original image $I(u, v)$ and some (continuous) geometric transformations $T(x, y)$, the aim is to create a transformed image $I'(u', v')$ where all coordinates are discrete (i.e., $u, v \in \mathbb{Z}$ and

⁴ Image morphing has also been implemented in ImageJ as a plugin (*iMorph*) by Hajime Hirase (<http://rsb.info.nih.gov/ij/plugins/morph.html>).

$u', v' \in \mathbb{Z}$).⁵ This can be accomplished in one of two ways, which differ by the mapping direction and are commonly referred to as *source-to-target* or *target-to-source* mapping, respectively.

21.2.1 Source-to-Target Mapping

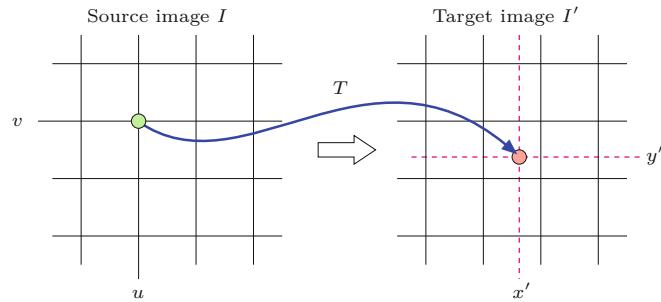
In this approach, which appears quite natural at first sight, we compute for every pixel (u, v) of the original (*source*) image I the corresponding transformed position

$$(x', y') = T(u, v) \quad (21.72)$$

in the target image I' . In general, the result will *not* coincide with any of the raster points, as illustrated in Fig. 21.9. Subsequently, we would have to decide in which pixel in the target image I' the original intensity or color value from $I(u, v)$ should be stored. We could perhaps even think of somehow distributing this value onto all adjacent pixels.

Fig. 21.9

Source-to-target mapping. For each discrete pixel position (u, v) in the source image I , the corresponding (continuous) target position (x', y') is found by applying the geometric transformation $T(u, v)$. In general, the target position (x', y') does not coincide with any discrete raster point. The source pixel value $I(u, v)$ is subsequently transferred to one of the adjacent target pixels.

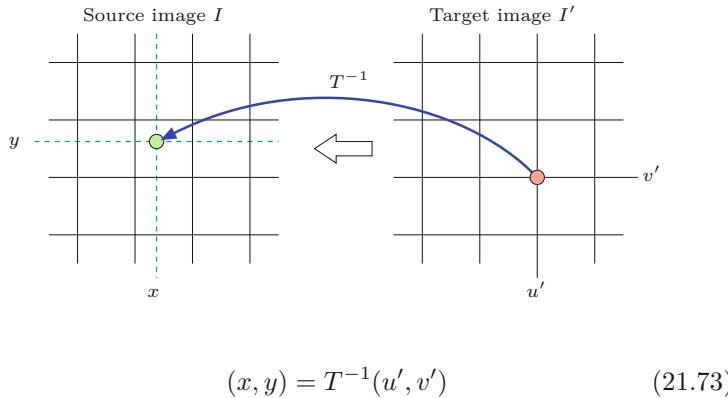


The problem with the source-to-target method is that, depending on the geometric transformation T , some elements in the target image I' may never be “hit” at all (i.e., never receive a source pixel value)! This happens, for example, when the image is enlarged (even slightly) by the geometric transformation. The resulting holes in the target image would be difficult to close in a subsequent processing step. Conversely, one would have to consider (e.g., when the image is shrunk) that a single element in the target image I' may be hit by multiple source pixels and thus image content may get lost. In the light of all these complications, source-to-target mapping is not really the method of choice.

21.2.2 Target-to-Source Mapping

This method avoids most difficulties encountered in the source-to-target mapping by simply reversing the image generation process. For every discrete pixel position (u', v') in the *target* image, we determine the corresponding (continuous) point

⁵ Remark on notation: We mostly use (u, v) or (u', v') to denote *discrete* (integer) coordinates and (x, y) or (x', y') for *continuous* (real-valued) coordinates.



in the source image plane using the inverse geometric transformation T^{-1} . Of course, the coordinate (x, y) again does not fall onto a raster point in general (Fig. 21.10) and thus we have to decide from which of the neighboring source pixels to extract the resulting target pixel value. This problem of interpolating among intensity values is discussed in detail in Chapter 22.

The major advantage of the target-to-source method is that all pixels in the target image I' (and only these) are computed and filled exactly once such that no holes or multiple hits can occur. This, however, requires the *inverse* geometric transformation T^{-1} to be available, which is no disadvantage in most cases since the forward transformation T itself is never really needed. Due to its simplicity, which is also demonstrated in Alg. 21.1, *target-to-source* mapping is the common method for geometrically transforming 2D images.

```

1: TransformImage ( $I, T$ )
   Input:  $I$ , source image;  $T$ , continuous mapping  $\mathbb{R}^2 \mapsto \mathbb{R}^2$ .
   Returns the transformed image.
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $I' \leftarrow \text{duplicate}(I)$                                  $\triangleright$  create the target image
4: for all  $(u, v) \in M \times N$  do           $\triangleright$  loop over all target pixels
5:    $(x, y) \leftarrow T^{-1}(u, v)$ 
6:    $I'(u, v) \leftarrow \text{GetInterpolatedValue}(I, x, y)$ 
7: return  $I'$ 
```

21.3 JAVA IMPLEMENTATION

Fig. 21.10

Target-to-source mapping. For each discrete pixel position (u', v') in the target image I' , the corresponding continuous source position (x, y) is found by applying the inverse mapping function $T^{-1}(u', v')$. The new pixel value $I'(u', v')$ is determined by interpolating the pixel values in the source image within some neighborhood of (x, y) .

Alg. 21.1

Geometric image transformation using target-to-source mapping. Given are the original (source) image I and the continuous coordinate transformation T . $\text{GetInterpolatedValue}(I, x, y)$ returns the interpolated value of the source image I at the continuous position (x, y) .

21.3 Java Implementation

In plain ImageJ, only a few simple geometric operations are provided as methods for the `ImageProcessor` class, such as rotation and flipping.⁶ This section describes the implementation of the transformations described in this chapter, which is openly available as part of the `imagingbook` library.⁷

⁶ Additional operations, including affine transformations, are available as plugin classes as part of the optional `TransformJ` package [162].

⁷ Package `imagingbook.pub.geometry.mappings`.

21.3.1 General Mappings (Class Mapping)

The abstract class `Mapping` is the superclass for all subsequent transformations. All subclasses of `Mapping` are required to implement the method `applyTo(double[] pnt)`, which applies the associated transformation to a given coordinate point and returns the transformed point. The actual transformations are implemented by its concrete sub-classes. The `applyTo()` method is defined in multiple versions with different signatures:

`double[] applyTo (double[] pnt)`

Applies this transformation to the 2D point (of type `double[]`) and returns the transformed coordinate.

`Point2D applyTo (Point2D pnt)`

Applies this transformation to the 2D point (of type `Point2D`) and returns the transformed coordinate.

`Point2D[] applyTo (Point2D[] pnts)`

Applies this transformation to a sequence of the 2D points (of type `Point2D`) and returns a sequence of transformed coordinates.

In addition, the `Mapping` class can also be used to transform entire images:

`double[] applyTo (ImageProcessor source, ImageProcessor target, PixelInterpolator.Method im)`

Transforms the input image `source` onto the output image `target` by target-to-source mapping, using the pixel interpolation method `im`.

`double[] applyTo (ImageProcessor ip, PixelInterpolator.Method im)`

Transforms the input image `ip` destructively, using the pixel interpolation method `im`.

`double[] applyTo (ImageInterpolator source, ImageProcessor target)`

Transforms the input image (specified by the interpolator `source`) onto the output image `target` by target-to-source mapping.

Other methods defined by class `Mapping`:

`Mapping duplicate ()`

Returns a copy of this mapping.

`Mapping getInverse ()`

Returns the inverse of this mapping if available. Otherwise an `UnsupportedOperationException` is thrown.

21.3.2 Linear Mappings

Linear transformations are implemented by class `LinearMapping`,⁸ with sub-classes including

`AffineMapping,` `Scaling,`

`ProjectiveMapping,` `Shear,`

`Rotation,` `Translation.`

⁸ Package `imagingbook.pub.geometry.mappings.linear`.

21.3.3 Nonlinear Mappings

Selected nonlinear transformations are implemented by the following subclasses of `Mapping`:⁹

BilinearMapping,	ShereMapping,
RippleMapping,	TwirlMapping.

21.3.4 Sample Applications

The following two ImageJ plugins show two simple examples of the use of the classes in Secs. 21.3.2 and 21.3.3 for implementing geometric operations and pixel interpolation (see Ch. 22 for details). Note that these plugins can be applied to any type of image.

Example 1: image rotation

The example in Prog. 21.1 shows a plugin (`Transform_Rotate`) to rotate an image by 15°. First (in line 16) the geometric mapping object (`map`) is created as an instance of class `Rotation`, with the supplied angle being converted from degrees to radians. The actual transformation of the image is performed by invoking the method `applyTo()` in line 17.

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import imagingbook.pub.geometry.interpolators.pixel.
    PixelInterpolator;
5 import imagingbook.pub.geometry.mappings.Mapping;
6 import imagingbook.pub.geometry.mappings.linear.Rotation;
7
8 public class Transform_Rotate implements PlugInFilter {
9     static double angle = 15; // rotation angle (in degrees)
10
11     public int setup(String arg, ImagePlus imp) {
12         return DOES_ALL;
13     }
14
15     public void run(ImageProcessor ip) {
16         Mapping map = new Rotation((2*Math.PI*angle)/360);
17         map.applyTo(ip, PixelInterpolator.Method.Bicubic);
18     }
19 }
```

Prog. 21.1

Image rotation example using the `Rotation` class (ImageJ plugin).

Example 2: projective transformation

The second example in Prog. 21.2 illustrates the implementation of a projective transformation. The geometric mapping T is defined by two corresponding quadrilaterals $P = p_0, \dots, p_3$ and $Q = q_0, \dots, q_3$, respectively. In a real application, these points would probably be specified interactively or given as the result of a mesh partitioning.

⁹ Package `imagingbook.pub.geometry.mappings.nonlinear`.

21 GEOMETRIC OPERATIONS

Prog. 21.2

Projective image transformation example using the `ProjectiveMapping` class (ImageJ plugin).

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import imagingbook.pub.geometry.interpolators.pixel.
    PixelInterpolator;
5 import imagingbook.pub.geometry.mappings.Mapping;
6 import imagingbook.pub.geometry.mappings.linear.
    ProjectiveMapping;
7 import java.awt.Point;
8 import java.awt.geom.Point2D;
9
10 public class Transform_Projective implements PlugInFilter {
11
12     public int setup(String arg, ImagePlus imp) {
13         return DOES_ALL;
14     }
15
16     public void run(ImageProcessor ip) {
17         Point2D p0 = new Point(0, 0);
18         Point2D p1 = new Point(400, 0);
19         Point2D p2 = new Point(400, 400);
20         Point2D p3 = new Point(0, 400);
21
22         Point2D q0 = new Point(0, 60);
23         Point2D q1 = new Point(400, 20);
24         Point2D q2 = new Point(300, 400);
25         Point2D q3 = new Point(30, 200);
26
27         Mapping map = new
28             ProjectiveMapping(p0, p1, p2, p3, q0, q1, q2, q3);
29
30         map.applyTo(ip, PixelInterpolator.Method.Bilinear);
31     }
32 }
```

The transformation object `map` (representing the forward transformation T) is created by calling the associated constructor `ProjectiveMapping()` in line 28. The mapping is applied to the input image (line 30), as in the previous example, except for the use of *bilinear* pixel interpolation.

21.4 Exercises

Exercise 21.1. Show that a straight line $y = kx + d$ in 2D is mapped to another straight line under a projective transformation, as defined in Eqn. (21.32).

Exercise 21.2. Show that parallel lines remain parallel under affine transformation (Eqn. (21.20)).

Exercise 21.3. Design a nonlinear geometric transformation similar to the ripple transformation (Eqn. (21.67)) that uses a *sawtooth* function instead of a sinusoid for the distortions in the horizontal



(a) Original image

(b) Radial wave ($a = 10.0, \tau = 38$)(c) Clover ($a = 0.2, N = 8$)(d) Spiral ($a = 0.01$)(e) Angular wave ($a = 0.1, \tau = 38$)(f) Tapestry ($a = 5.0, \tau_x = \tau_y = 30$)

21.4 EXERCISES

Fig. 21.11

Examples of the nonlinear geometric transformations defined in Exercise 21.4. The reference point \mathbf{x}_c is always taken at the image center.

and vertical directions. Use the class `TwirlMapping` as a template for your implementation.

Exercise 21.4. Implement one or more of the following nonlinear geometric transformations (see Fig. 21.11):

- A. **Radial wave** transformation: This transformation simulates an omni-directional wave which originates from a fixed center point \mathbf{x}_c (see Fig. 21.11(b)). The inverse transformation (applied to a target image point $\mathbf{x}' = (x', y')$) is

$$T^{-1}: \mathbf{x} = \begin{cases} \mathbf{x}_c & \text{for } r = 0, \\ \mathbf{x}_c + \frac{r+\delta}{r} \cdot (\mathbf{x}' - \mathbf{x}_c) & \text{for } r > 0, \end{cases} \quad (21.74)$$

with $r = \|\mathbf{x}' - \mathbf{x}_c\|$ and $\delta = a \cdot \sin(2\pi r/\tau)$. Parameter a specifies the *amplitude* (strength) of the distortion and τ is the *period* (width) of the radial wave (in pixel units).

- B. **Clover** transformation: This transformation distorts the image in the form of a N -leafed clover shape (see Fig. 21.11(c)). The associated inverse transformation is the same as in Eqn. (21.74) but uses

$$\delta = a \cdot r \cdot \cos(N \cdot \alpha), \quad \text{with } \alpha = \angle(\mathbf{x}' - \mathbf{x}_c) \quad (21.75)$$

instead. Again $r = \|\mathbf{x}' - \mathbf{x}_c\|$ is the radius of the target image point \mathbf{x}' from the designated center point \mathbf{x}_c . Parameter a specifies the amplitude of the distortion and N is the number of radial “leaves”.

- C. **Spiral** transformation: This transformation (see Fig. 21.11(d)) is similar to the *twirl* transformation in Eqns. (21.63)–(21.64), defined by the inverse transformation

$$T^{-1}: \mathbf{x} = \mathbf{x}_c + r \cdot \begin{pmatrix} \cos(\beta) \\ \sin(\beta) \end{pmatrix}, \quad (21.76)$$

with $\beta = \angle(\mathbf{x}' - \mathbf{x}_c) + a \cdot r$ and $r = \|\mathbf{x}' - \mathbf{x}_c\|$ denoting the distance from the target point \mathbf{x}' and the center point \mathbf{x}_c . The angle β increases linearly with r ; parameter a specifies the “velocity” of the spiral.

- D. **Angular wave** transformation: This is another variant of the *twirl* transformation in Eqns. (21.63)–(21.64). Its inverse transformation is the same as for the spiral mapping in Eqn. (21.76), but in this case

$$\beta = \angle(\mathbf{x}' - \mathbf{x}_c) + a \cdot \sin\left(\frac{2\pi r}{\tau}\right). \quad (21.77)$$

Thus the angle β is modified by a sine function with amplitude a (see Fig. 21.11(e)).

- E. **Tapestry** transformation: In this case the inverse transformation of a target point $\mathbf{x}' = (x', y')$ is

$$T^{-1}: \mathbf{x} = \mathbf{x}' + a \cdot \begin{pmatrix} \sin\left(\frac{2\pi}{\tau_x} \cdot (x' - x_c)\right) \\ \sin\left(\frac{2\pi}{\tau_y} \cdot (y' - y_c)\right) \end{pmatrix}, \quad (21.78)$$

again with the center point $\mathbf{x}_c = (x_c, y_c)$. Parameter a specifies the distortion’s amplitude and τ_x, τ_y are the wavelengths (measured in pixel units) along the x and y axis, respectively (see Fig. 21.11(f)).

Exercise 21.5. Implement an interactive program (plugin) that performs projective rectification (see Sec. 21.1.4) of a selected quadrilateral, as shown in Fig. 21.12. Make your program perform the following steps:

1. Let the user mark the source quad in the source image I as a polygon-shaped *region of interest* (ROI) with at least four points $\mathbf{x}_0, \dots, \mathbf{x}_3$. In ImageJ this is easily done with the built-in polygon selection tool (see Prog. 21.3 for handling ROI points).
2. Create an output image I' of fixed size (i.e., proportional to A4 or Letter paper size).
3. The target rectangle is defined by the four corners $\mathbf{x}'_0, \dots, \mathbf{x}'_3$ of the output image. The source and target points are associated 1:1, that is, the four corresponding point pairs are $\langle \mathbf{x}_0, \mathbf{x}'_0 \rangle, \dots, \langle \mathbf{x}_3, \mathbf{x}'_3 \rangle$.

- From the four point pairs, create an instance of **Projective-Mapping**, as demonstrated in Prog. 21.2.
- Test the obtained mapping by applying **A** to the specified source points x_0, \dots, x_3 . Make sure they project exactly to the specified target points x'_0, \dots, x'_3 .
- Apply the obtained mapping from the source to the target image using the method¹⁰

```
void applyTo(ImageProcessor source,
             ImageProcessor target, InterpolationMethod im).
```

- Show the resulting output image.



21.4 EXERCISES

Fig. 21.12

Projective rectification example (see Exercise 21.5). Source image and user-defined selection (a); transformed output image (b).

¹⁰ Defined in class `imagingbook.pub.geometry.mappings.Mapping`.

21 GEOMETRIC OPERATIONS

Prog. 21.3

ImageJ plugin demonstrating the extraction of vertex points from a user-selected polygon-ROI (region of interest). Notice that (in line 21) the region of interest (ROI) is obtained from the associated `ImagePlus` instance (to which a reference is kept in line 16) and not from the supplied `ImageProcessor` object. ImageJ's ROI coordinates are integer positions in general.

```
1 import java.awt.Point;
2 import java.awt.Polygon;
3 import java.awt.geom.Point2D;
4
5 import ij.ImagePlus;
6 import ij.gui.PolygonRoi;
7 import ij.gui.Roi;
8 import ij.plugin.filter.PlugInFilter;
9 import ij.process.ImageProcessor;
10
11 public class Get_Roi_Points implements PlugInFilter {
12
13     ImagePlus im = null;
14
15     public int setup(String args, ImagePlus im) {
16         this.im = im; // keep a reference to im
17         return DOES_ALL + ROI_REQUIRED;
18     }
19
20     public void run(ImageProcessor source) {
21         Roi roi = im.getRoi();
22         if (!(roi instanceof PolygonRoi)) {
23             IJ.error("Polygon selection required!");
24             return;
25         }
26
27         Polygon poly = roi.getPolygon();
28
29         // copy polygon vertices to a point array:
30         Point2D[] pts = new Point2D[poly.npoints];
31         for (int i = 0; i < poly.npoints; i++) {
32             pts[i] = new Point(poly.xpoints[i], poly.ypoints[i]);
33         }
34
35         ... // use the ROI points in pts
36
37     }
38
39 }
```

Pixel Interpolation

Interpolation is the process of estimating the intermediate values of a sampled function or signal at continuous positions or the attempt to reconstruct the original continuous function from a set of discrete samples. In the context of geometric operations this task arises from the fact that discrete pixel positions in one image are generally not mapped to discrete raster positions in the other image under some continuous geometric transformation T (or T^{-1} , respectively). The concrete goal is to obtain an optimal estimate for the value of the 2D image function $I(x, y)$ at any continuous position $(x, y) \in \mathbb{R}^2$ to implement the function

$$\text{GetInterpolatedValue}(I, x, y),$$

which we defined in Chapter 21 (see Alg. 21.1). Ideally the interpolated image should preserve as much detail (i.e., sharpness) as possible without causing visible artifacts such as ringing or moiré patterns.

22.1 Simple Interpolation Methods

To illustrate the problem, we first attend to the 1D case (see Fig. 22.1). Several simple, ad-hoc methods exist for interpolating the values of a discrete function $g(u)$, with $u \in \mathbb{Z}$, at arbitrary continuous positions $x \in \mathbb{R}$. The simplest of all interpolation methods is to round the continuous coordinate x to the closest integer u_x and use the associated sample $g(u_x)$ as the interpolated value, that is,

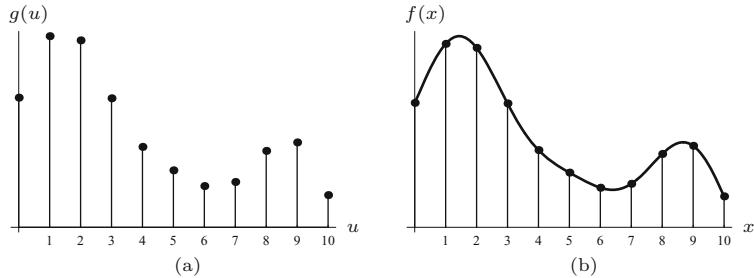
$$\tilde{g}(x) \leftarrow g(u_x), \quad (22.1)$$

with $u_x = \text{round}(x) = \lfloor x + 0.5 \rfloor$. A typical result of this so-called *nearest-neighbor interpolation* is shown in Fig. 22.2(a).

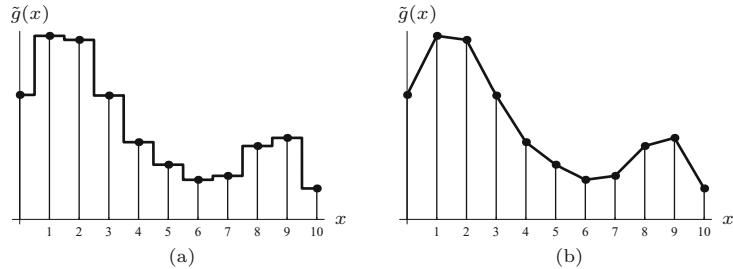
Another simple method is *linear interpolation*. Here the estimated value is the sum of the two closest samples $g(u_0)$ and $g(u_0 + 1)$, with $u_0 = \lfloor x \rfloor$. The weight of each sample is proportional to its closeness to the continuous position x , that is,

Fig. 22.1

Interpolating a discrete function in 1D. Given the discrete function values $g(u)$ (a), the goal is to estimate the original function $f(x)$ at arbitrary continuous positions $x \in \mathbb{R}$ (b).


Fig. 22.2

Simple interpolation methods. The *nearest-neighbor interpolation* (a) simply selects the discrete sample $g(u)$ closest to the given continuous coordinate x as the interpolating value $\tilde{g}(x)$. Under *linear interpolation* (b), the result is a piecewise linear function connecting adjacent samples $g(u)$ and $g(u + 1)$.



$$\begin{aligned}\tilde{g}(x) &= g(u_x) + (x - u_x) \cdot (g(u_x + 1) - g(u_x)) \\ &= g(u_x) \cdot (1 - (x - u_x)) + g(u_x + 1) \cdot (x - u_x).\end{aligned}\quad (22.2)$$

As shown in Fig. 22.2(b), the result is a piecewise linear function made up of straight line segments between consecutive sample values.

22.1.1 Ideal Low-Pass Filter

Obviously the results of these simple interpolation methods do not well approximate the original continuous function (Fig. 22.1). But how can we obtain a better approximation from the discrete samples only when the original function is unknown? This may appear hopeless at first, because the discrete samples $g(u)$ could possibly originate from any continuous function $f(x)$ with identical values at the discrete sample positions.

We find an intuitive answer to this question (once again) by looking at the functions in the spectral domain. If the original function $f(x)$ was discretized in accordance with the *sampling theorem* (see Ch. 18, Sec. 18.2.1), then $f(x)$ must have been “band limited”—it could not contain any signal components with frequencies higher than half the sampling frequency ω_s . This means that the reconstructed signal can only contain a limited set of frequencies and thus its trajectory between the discrete sample values is not arbitrary but naturally constrained.

In this context, absolute units of measure are of no concern since in a digital signal all frequencies relate to the sampling frequency. In particular, if we take $\tau_s = 1$ as the (unitless) sampling interval, the resulting sampling frequency is

$$\omega_s = 2 \cdot \pi \cdot f_s = 2 \cdot \pi \cdot \frac{1}{\tau_s} = 2 \cdot \pi \quad (22.3)$$

and thus the maximum signal frequency is $\omega_{\max} = \frac{\omega_s}{2} = \pi$. To isolate the frequency range $-\omega_{\max} \dots \omega_{\max}$ in the corresponding (periodic)

Fourier spectrum, we multiply the spectrum $G(\omega)$ by a square windowing function $\Pi_\pi(\omega)$ of width $\pm\omega_{\max} = \pm\pi$,

$$\tilde{G}(\omega) = G(\omega) \cdot \Pi_\pi(\omega) = G(\omega) \cdot \begin{cases} 1 & \text{for } -\pi \leq \omega \leq \pi, \\ 0 & \text{otherwise.} \end{cases} \quad (22.4)$$

This is called an *ideal low-pass filter*, which cuts off all signal components with frequencies greater than π and keeps all lower-frequency components unchanged. In the signal domain, the operation in Eqn. (22.4) corresponds (see Eqn. (18.27)) to a *linear convolution* with the inverse Fourier transform of the windowing function $\Pi_\pi(\omega)$, which is the *Sinc* function, defined as

$$\text{Sinc}(x) = \frac{\sin(\pi x)}{\pi x}, \quad (22.5)$$

and shown in Fig. 22.3 (see also Ch. 18, Table 18.1). This correspondence, which was already discussed in Chapter 18, Sec. 18.1.6, between convolution in the signal domain and simple multiplication in the frequency domain is summarized in Fig. 22.4.

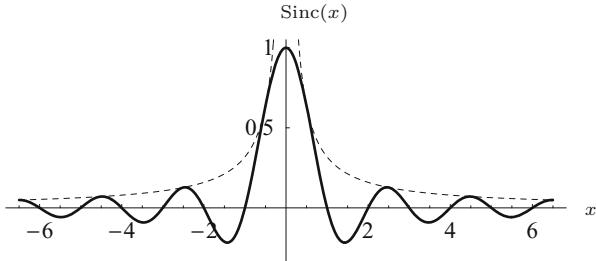


Fig. 22.3

Sinc function in 1D. The function $\text{Sinc}(x)$ has the value 1 at the origin and zero values at all integer positions. The dashed line plots the amplitude $|\frac{1}{\pi x}|$ of the underlying sine function.

So theoretically $\text{Sinc}(x)$ is the ideal interpolation function for reconstructing a frequency-limited continuous signal. To compute the interpolated value for the discrete function $g(u)$ at an arbitrary position x_0 , the Sinc function is shifted to x_0 (such that its origin lies at x_0), multiplied with all sample values $g(u)$, with $u \in \mathbb{Z}$, and the results are summed—that is, $g(u)$ and $\text{Sinc}(x)$ are *convolved*. The reconstructed value of the continuous function at position x_0 is thus

$$\tilde{g}(x_0) = [\text{Sinc} * g](x_0) = \sum_{u=-\infty}^{\infty} \text{Sinc}(x_0 - u) \cdot g(u), \quad (22.6)$$

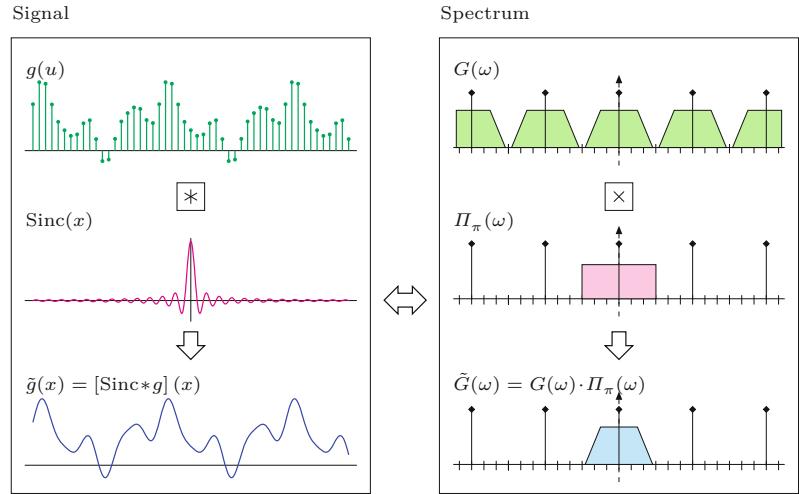
where $*$ is the linear convolution operator (see Ch. 5, Sec. 5.3.1). If the discrete signal $g(u)$ is *finite* with length N (as is usually the case), it is assumed to be *periodic* (i.e., $g(u) = g(u + kN)$ for all $k \in \mathbb{Z}$).¹ In this case, Eqn. (22.6) modifies to

$$\tilde{g}(x_0) = \sum_{u=-\infty}^{\infty} \text{Sinc}(x_0 - u) \cdot g(u \bmod N). \quad (22.7)$$

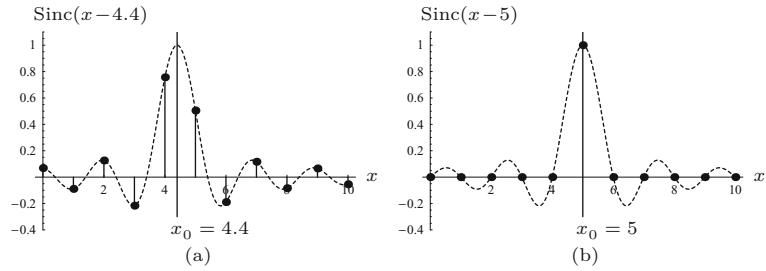
¹ This assumption is explained by the fact that a discrete Fourier spectrum implicitly corresponds to a periodic signal (also see Ch. 18, Sec. 18.2.2).

Fig. 22.4

Interpolation of a discrete signal—relation between signal and frequency space. The discrete signal $g(u)$ in signal space (left) corresponds to the periodic Fourier spectrum $G(\omega)$ in frequency space (right). The spectrum $\hat{G}(\omega)$ of the continuous signal is isolated from $G(\omega)$ by point-wise multiplication (\times) with the square function $\Pi_\pi(\omega)$, which constitutes an ideal low-pass filter (right). In signal space (left), this operation corresponds to a linear convolution ($*$) with the function $\text{Sinc}(x)$.


Fig. 22.5

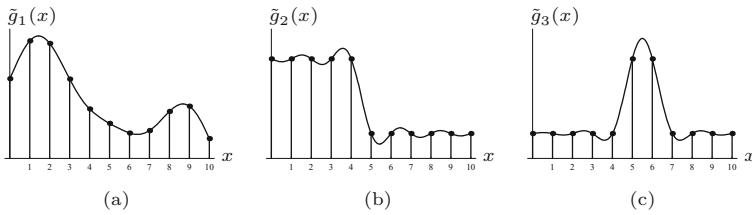
Interpolation by convolving with the Sinc function. The Sinc function is shifted by aligning its origin with the interpolation points $x_0 = 4.4$ (a) and $x_0 = 5$ (b). The values of the shifted Sinc function (dashed curve) at the integral positions are the weights (coefficients) for the corresponding sample values $g(u)$. When the function is interpolated at some *integral* position, such as $x_0 = 5$ (b), only the sample value $g(x_0) = g(5)$ is considered and weighted with 1, while all other samples coincide with the zero positions of the Sinc function and thus do not contribute to the result.



It may be surprising that the ideal interpolation of a discrete function $g(u)$ at a position x_0 apparently involves not only a few neighboring sample points but, in general, *infinitely many* values of $g(u)$ whose weights decrease continuously with their distance from the given interpolation point x_0 (at the rate $|\frac{1}{\pi(x_0-u)}|$). Figure 22.5 shows two examples for interpolating the function $g(u)$ at positions $x_0 = 4.4$ and $x_0 = 5$. If the function is interpolated at some integral position, such as $x_0 = 5$, the sample $g(u)$ at $u = x_0$ receives the weight 1, while all other samples coincide with the zero positions of the Sinc function and are thus ignored. Consequently, the resulting interpolation values are identical to the sample values $g(u)$ at all discrete positions $x = u$.

If a continuous signal is properly frequency limited (by half the sampling frequency $\frac{\omega_s}{2}$), it can be exactly reconstructed from the discrete signal by interpolation with the Sinc function, as Fig. 22.6(a) demonstrates. Problems occur, however, around local high-frequency signal events, such as rapid transitions or pulses, as shown in Fig. 22.6(b,c). In those situations, the Sinc interpolation causes strong overshooting or “ringing” artifacts, which are perceived as visually disturbing. For practical applications, the Sinc function is therefore not suitable as an interpolation kernel—not only because of its infinite extent (and the resulting noncomputability).

A good interpolation function implements a low-pass filter that, on the one hand, introduces minimal blurring by maintaining the



22.2 INTERPOLATION BY CONVOLUTION

Fig. 22.6

Sinc interpolation applied to various signal types. The reconstructed function in (a) is identical to the continuous, band-limited original. The results for the step function (b) and the pulse function (c) show the strong ringing caused by Sinc (ideal low-pass) interpolation.

maximum signal bandwidth but, on the other hand, also delivers a good reconstruction at rapid signal transitions. In this regard, the Sinc function is an extreme choice—it implements an ideal low-pass filter and thus preserves a maximum bandwidth and signal continuity but gives inferior results at signal transitions. At the opposite extreme, nearest-neighbor interpolation (see Fig. 22.2) can perfectly handle steps and pulses but generally fails to produce a continuous signal reconstruction between sample points. The design of an interpolation function thus always involves a trade-off, and the quality of the results often depends on the particular application and subjective judgment. In the following, we discuss some common interpolation functions that come close to this goal and are therefore frequently used in practice.

22.2 Interpolation by Convolution

As we saw earlier in the context of Sinc interpolation (Eqn. (22.5)), the reconstruction of a continuous signal can be described as a linear convolution operation. In general, we can express interpolation as a convolution of the given discrete function $g(u)$ with some continuous *interpolation kernel* $w(x)$ as

$$\tilde{g}(x_0) = [w * g](x_0) = \sum_{u=-\infty}^{\infty} w(x_0 - u) \cdot g(u). \quad (22.8)$$

The Sinc interpolation in Eqn. (22.6) is obviously only a special case with $w(x) = \text{Sinc}(x)$. Similarly, the 1D *nearest-neighbor interpolation* (Eqn. (22.1), Fig. 22.2(a)) can be expressed as a linear convolution with the kernel

$$w_{\text{nn}}(x) = \begin{cases} 1 & \text{for } -0.5 \leq x < 0.5, \\ 0 & \text{otherwise,} \end{cases} \quad (22.9)$$

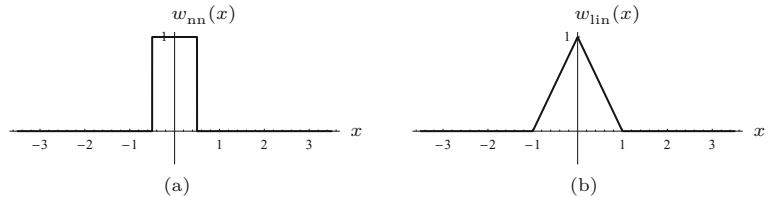
and the *linear interpolation* (see Eqn. (22.2), Fig. 22.2(b)) with the kernel

$$w_{\text{lin}}(x) = \begin{cases} 1-x & \text{for } |x| < 1, \\ 0 & \text{for } |x| \geq 1. \end{cases} \quad (22.10)$$

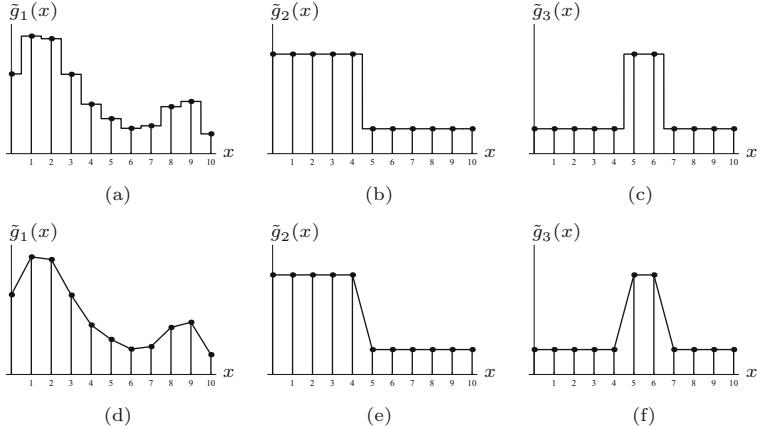
Both interpolation kernels $w_{\text{nn}}(x)$ and $w_{\text{lin}}(x)$ are shown in Fig. 22.7, and results for various function types are plotted in Fig. 22.8.

Fig. 22.7

Convolution kernels for the nearest-neighbor interpolation $w_{\text{nn}}(x)$ and the linear interpolation $w_{\text{lin}}(x)$.


Fig. 22.8

Interpolation examples (1D): nearest-neighbor interpolation (a–c), linear interpolation (d–f).



22.3 Cubic Interpolation

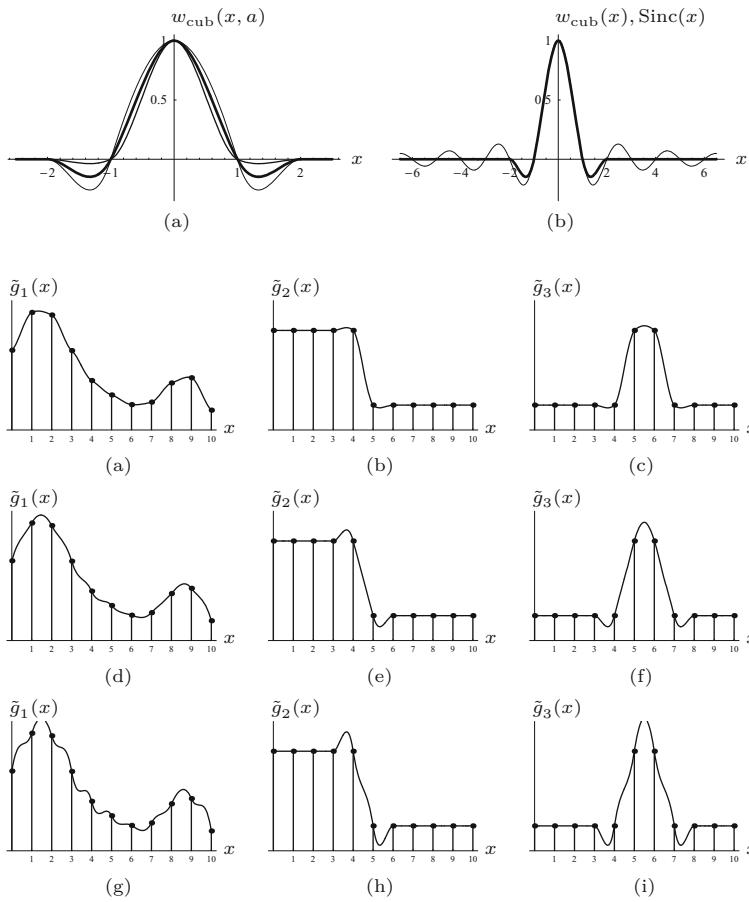
The Sinc function is not a useful interpolation kernel in practice, because of its infinite extent and the ringing artifacts caused by its slowly decaying oscillations. Therefore several interpolation methods employ a truncated version of the Sinc function or an approximation of it, thereby making the convolution kernel more compact and reducing the ringing. A frequently used approximation of a truncated Sinc function is the so-called cubic interpolation, whose convolution kernel is defined as the piecewise cubic polynomial

$$w_{\text{cub}}(x, a) = \begin{cases} (-a+2) \cdot |x|^3 + (a-3) \cdot |x|^2 + 1 & \text{for } 0 \leq |x| < 1, \\ -a \cdot |x|^3 + 5a \cdot |x|^2 - 8a \cdot |x| + 4a & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (22.11)$$

Parameter a can be used to adjust the steepness of the spline function and thus the perceived “sharpness” of the interpolation (see Fig. 22.9(a)). For the standard value $a = 1$, Eqn. (22.11) simplifies to

$$w_{\text{cub}}(x) = \begin{cases} |x|^3 - 2 \cdot |x|^2 + 1 & \text{for } 0 \leq |x| < 1, \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (22.12)$$

The comparison of the Sinc function and the cubic interpolation kernel $w_{\text{cub}}(x) = w_{\text{cub}}(x, -1)$ in Fig. 22.9(b) shows that many high-value coefficients outside $x = \pm 2$ are truncated and thus relatively large errors can be expected. However, because of the compactness of the cubic function, this type of interpolation can be calculated



22.3 CUBIC INTERPOLATION

Fig. 22.9

Cubic interpolation kernel. Function $w_{\text{cub}}(x, a)$ with control parameter a set to $a = 0.25$ (dashed curve), $a = 1$ (continuous curve), and $a = 1.75$ (dotted curve) (a). Cubic function $w_{\text{cub}}(x)$ and Sinc function compared (b).

Fig. 22.10

Cubic interpolation examples. Parameter a in Eqn. (22.11) controls the amount of signal overshoot or perceived sharpness: $a = 0.25$ (a–c), standard setting $a = 1$ (d–f), $a = 1.75$ (g–i). Notice in (d) the ripple effects incurred by interpolating with the standard settings in smooth signal regions.

very efficiently. Since $w_{\text{cub}}(x) = 0$ for $|x| \geq 2$, only *four* discrete values $g(u)$ need to be accounted for in the convolution operation (Eqn. (22.8)) at any continuous position $x \in \mathbb{R}$, that is,

$$g(u_0-1), g(u_0), g(u_0+1), g(u_0+2), \quad \text{with } u_0 = \lfloor x_0 \rfloor.$$

This reduces the 1D cubic interpolation to the expression

$$\tilde{g}(x_0) = \sum_{u=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} w_{\text{cub}}(x_0 - u) \cdot g(u). \quad (22.13)$$

Figure 22.10 shows the results of cubic interpolation with different settings of the control parameter a . Notice that the cubic reconstruction obtained with the popular standard setting ($a = 1$) exhibits substantial overshooting at edges as well as strong ripple effects in the continuous parts of the signal (Fig. 22.10(d)). With $a = 0.5$, the expression in Eqn. (22.11) corresponds to a *Catmull-Rom* spline [44] (see also Sec. 22.4), which produces significantly better results than the standard setup (with $a = 1$), particularly in smooth signal regions (see Fig. 22.12(a–c)).

22.4 Spline Interpolation

The cubic interpolation kernel (Eqn. (22.11)) described in the previous section is a piecewise cubic polynomial function, also known as a *cubic spline* in computer graphics. In its general form, this function takes not only one but *two* control parameters (a, b) [164],²

$$w_{\text{cs}}(x, a, b) = \quad (22.14)$$

$$\frac{1}{6} \cdot \begin{cases} (-6a - 9b + 12) \cdot |x|^3 \\ \quad + (6a + 12b - 18) \cdot |x|^2 - 2b + 6 & \text{for } 0 \leq |x| < 1, \\ (-6a - b) \cdot |x|^3 + (30a + 6b) \cdot |x|^2 \\ \quad + (-48a - 12b) \cdot |x| + 24a + 8b & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases}$$

Equation (22.14) describes a family of smooth, C^1 -continuous functions (i.e., with continuous first derivatives) with no visible discontinuities or sharp corners. For $b = 0$, the function $w_{\text{cs}}(x, a, b)$ specifies a one-parameter family of so-called *cardinal splines* equivalent to the cubic interpolation function $w_{\text{cub}}(x, a)$ in Eqn. (22.11),

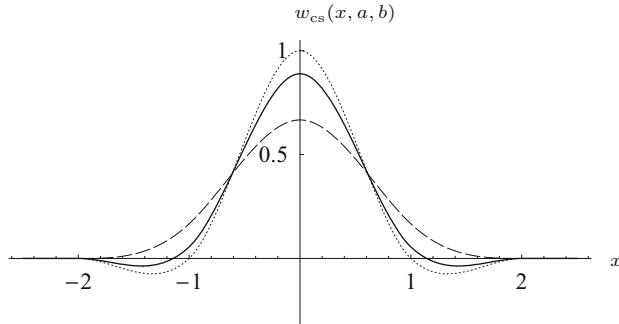
$$w_{\text{cs}}(x, a, 0) = w_{\text{cub}}(x, a), \quad (22.15)$$

and for the standard setting $a = 1$ (Eqn. (22.12)) in particular

$$w_{\text{cs}}(x, 1, 0) = w_{\text{cub}}(x, 1) = w_{\text{cub}}(x). \quad (22.16)$$

Figure 22.11 shows three additional examples of this function type that are important in the context of interpolation: *Catmull-Rom splines*, *cubic B-splines*, and the *Mitchell-Netravali* function. All three functions are briefly described in the following sections. The actual calculation of the interpolated signal follows exactly the same scheme as used for the cubic interpolation described in Eqn. (22.13).

Fig. 22.11
Examples of cubic spline functions as defined in Eqn. (22.14): *Catmull-Rom spline* $w_{\text{cs}}(x, 0.5, 0)$ (dotted line), *cubic B-spline* $w_{\text{cs}}(x, 0, 1)$ (dashed line), and *Mitchell-Netravali* function $w_{\text{cs}}(x, \frac{1}{3}, \frac{1}{3})$ (solid line).



22.4.1 Catmull-Rom Interpolation

With the control parameters set to $a = 0.5$ and $b = 0$, the function in Eqn. (22.14) is a *Catmull-Rom spline* [44], as already mentioned in Sec. 22.3:

² In [164], the parameters a and b were originally named C and B , respectively, with $B \equiv b$ and $C \equiv a$.

$$w_{\text{crm}}(x) = w_{\text{cs}}(x, 0.5, 0) \quad (22.17)$$

$$= \frac{1}{2} \cdot \begin{cases} 3 \cdot |x|^3 - 5 \cdot |x|^2 + 2 & \text{for } 0 \leq |x| < 1, \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases}$$

Examples of signals interpolated with this kernel are shown in Fig. 22.12(a–c). The results are similar to ones produced by cubic interpolation (with $a = 1$, see Fig. 22.10) with regard to sharpness, but the Catmull-Rom reconstruction is clearly superior in smooth signal regions (compare, e.g., Fig. 22.10(d) vs. Fig. 22.12(a)).

22.4.2 Cubic B-spline Approximation

With parameters set to $a = 0$ and $b = 1$, Eqn. (22.14) corresponds to a cubic B-spline function of the form

$$\begin{aligned} w_{\text{cbs}}(x) &= w_{\text{cs}}(x, 0, 1) \quad (22.18) \\ &= \frac{1}{6} \cdot \begin{cases} 3 \cdot |x|^3 - 6 \cdot |x|^2 + 4 & \text{for } 0 \leq |x| < 1, \\ -|x|^3 + 6 \cdot |x|^2 - 12 \cdot |x| + 8 & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases} \end{aligned}$$

This function is positive everywhere and, when used as an interpolation kernel, causes a pure smoothing effect similar to a Gaussian smoothing filter (see Fig. 22.12(d–f)). The B-spline function in Eqn. (22.18) is C^2 -continuous, that is, its first *and* second derivatives are continuous. Notice that—in contrast to all previously described interpolation methods—the reconstructed function does *not* pass through all discrete sample points. Thus, to be precise, the reconstruction with cubic B-splines is not called an *interpolation* but an *approximation* of the signal.

22.4.3 Mitchell-Netravali Approximation

The design of an optimal interpolation kernel is always a trade-off between high bandwidth (sharpness) and good transient response (low ringing). Catmull-Rom interpolation, for example, emphasizes high sharpness, whereas cubic B-spline interpolation blurs but creates no ringing. Based on empirical tests, Mitchell and Netravali [164] proposed a cubic interpolation kernel as described in Eqn. (22.14) with parameter settings $a = \frac{1}{3}$ and $b = \frac{1}{3}$, and the resulting interpolation function

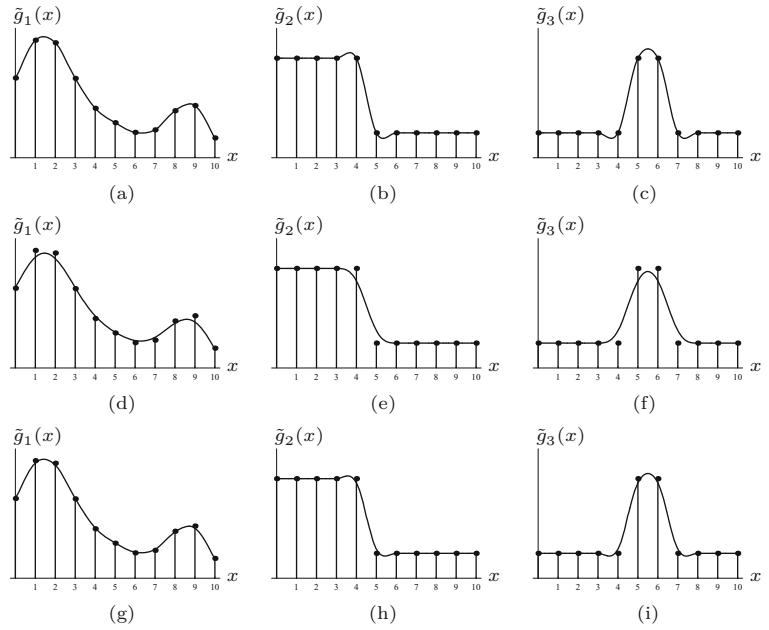
$$\begin{aligned} w_{\text{mn}}(x) &= w_{\text{cs}}(x, \frac{1}{3}, \frac{1}{3}) \quad (22.19) \\ &= \frac{1}{18} \cdot \begin{cases} 21 \cdot |x|^3 - 36 \cdot |x|^2 + 16 & \text{for } 0 \leq |x| < 1, \\ -7 \cdot |x|^3 + 36 \cdot |x|^2 - 60 \cdot |x| + 32 & \text{for } 1 \leq |x| < 2, \\ 0 & \text{for } |x| \geq 2. \end{cases} \end{aligned}$$

This function is the weighted sum of a Catmull-Rom spline in Eqn. (22.17) and a cubic B-spline in Eqn. (22.18).³ The examples in Fig.

³ See also Exercise 22.1.

Fig. 22.12

Cardinal spline reconstruction examples: *Catmull-Rom* interpolation (a–c), *cubic B-spline* approximation (d–f), and *Mitchell-Netravali* approximation (g–i).



22.12(g–i) show that this method is a good compromise, creating little overshoot, high edge sharpness, and good signal continuity in smooth regions. Since the resulting function does not pass through the original sample points, the Mitchell-Netravali method is again an *approximation* and not an interpolation.

22.4.4 Lanczos Interpolation

The Lanczos⁴ interpolation belongs to the family of “windowed Sinc” methods. In contrast to the methods described in the previous sections, these do *not* use a polynomial (or other) approximation of the Sinc function but the Sinc function *itself* combined with a suitable window function $\psi(x)$; that is, an interpolation kernel of the form

$$w(x) = \psi(x) \cdot \text{Sinc}(x). \quad (22.20)$$

The particular window functions for the Lanczos interpolation are defined as

$$\psi_{Ln}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/n)}{\pi x/n} & \text{for } 0 < |x| < n, \\ 0 & \text{for } |x| \geq n, \end{cases} \quad (22.21)$$

where $n \in \mathbb{N}$ denotes the *order* of the filter [176, 237]. Notice that the window function is again a truncated Sinc function! For the Lanczos filters of order $n = 2, 3$, which are the most commonly used in image processing, the corresponding window functions are

⁴ Cornelius Lanczos (1893–1974).

$$\psi_{L2}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/2)}{\pi x/2} & \text{for } 0 < |x| < 2, \\ 0 & \text{for } |x| \geq 2, \end{cases} \quad (22.22)$$

$$\psi_{L3}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/3)}{\pi x/3} & \text{for } 0 < |x| < 3, \\ 0 & \text{for } |x| \geq 3. \end{cases} \quad (22.23)$$

Both window functions are shown in Fig. 22.13(a,b). The 1D interpolation kernels w_{L2} and w_{L3} are obtained as the product of the Sinc function (Eqn. (22.5)) and the associated window function (Eqn. (22.21)), that is,

$$w_{L2}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ 2 \cdot \frac{\sin(\pi x/2) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 2, \\ 0 & \text{for } |x| \geq 2, \end{cases} \quad (22.24)$$

and

$$w_{L3}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ 3 \cdot \frac{\sin(\pi x/3) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 3, \\ 0 & \text{for } |x| \geq 3, \end{cases} \quad (22.25)$$

respectively. In general, for Lanczos interpolation of order n , we get

$$w_{Ln}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ n \cdot \frac{\sin(\pi x/n) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < n, \\ 0 & \text{for } |x| \geq n. \end{cases} \quad (22.26)$$

Figure 22.13(c,d) shows the resulting interpolation kernels together with the original Sinc function. The function $w_{L2}(x)$ is quite similar to the Catmull-Rom kernel $w_{crm}(x)$ (Eqn. (22.17), Fig. 22.11), so the results can be expected to be similar as well, as shown in Fig. 22.14(a–c) (cf. Fig. 22.12(a–c)). Notice, however, the relatively poor reconstruction in the smooth signal regions (Fig. 22.14(a)) and the strong ringing introduced in the constant high-amplitude regions (Fig. 22.14(b)). The “3-tap” kernel $w_{L3}(x)$ reduces these artifacts and produces steeper edges, at the cost of increased overshoot (Fig. 22.12(d–f)).

In summary, although Lanczos interpolators have seen revived interest and popularity in recent years, they do not seem to offer much (if any) advantage over other established methods, particularly the cubic, Catmull-Rom, or Mitchell-Netravali interpolations. While these are based on efficiently computable polynomial functions, Lanczos interpolation requires trigonometric functions which are relatively costly to compute, unless some form of tabulation is used.

22.5 Interpolation in 2D

So far we have only looked at interpolating (or reconstructing) 1D signals from discrete samples. Images are 2D signals but, as we

Fig. 22.13
1D Lanczos interpolation kernels. Lanczos window functions ψ_{L2} (a), ψ_{L3} (b), and the corresponding interpolation kernels w_{L2} (c), w_{L3} (d). The original Sinc function (dotted curve) is shown for comparison.

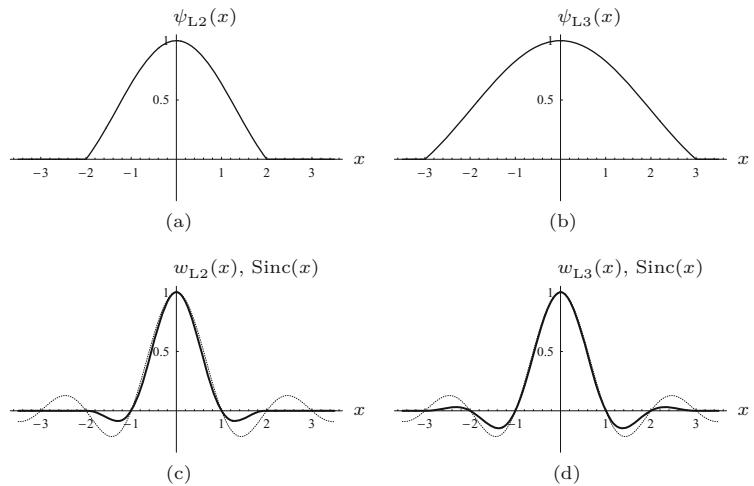
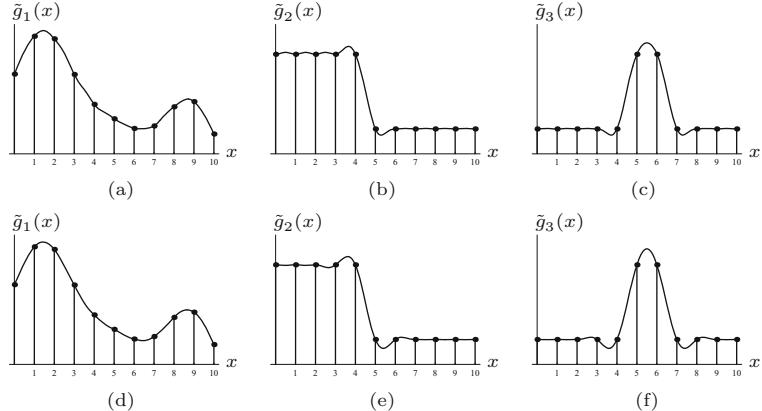


Fig. 22.14
Lanczos interpolation examples: Lanczos-2 (a–c), Lanczos-3 (d–f). Note the ringing in the flat (constant) regions caused by Lanczos-2 interpolation in the left part of (b). The Lanczos-3 interpolator shows less ringing (e) but produces steeper edges at the cost of increased overshoot (e, f).



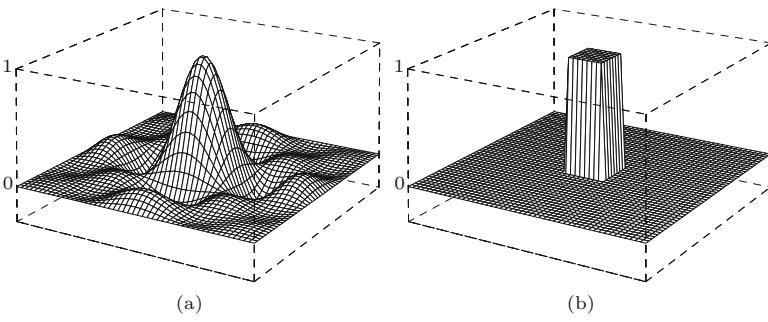
shall see in this section, the techniques for interpolating images are very similar and can be derived from the 1D approach. In particular, “ideal” (low-pass filter) interpolation requires a 2D Sinc function defined as

$$\text{SINC}(x, y) = \text{Sinc}(x) \cdot \text{Sinc}(y) = \frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi y)}{\pi y}, \quad (22.27)$$

which is shown in Fig. 22.15(a). Just as in 1D, the 2D Sinc function is not a practical interpolation function for various reasons. In the following, we look at some common interpolation methods for images, particularly the nearest-neighbor, bilinear, bicubic, and Lanczos interpolations, whose 1D versions were described in the previous sections.

22.5.1 Nearest-Neighbor Interpolation in 2D

The position (u_x, v_y) of the pixel closest to a given continuous point (x, y) is found by independently rounding the x and y coordinates to discrete values, that is,



22.5 INTERPOLATION IN 2D

Fig. 22.15
Interpolation kernels in 2D. Sinc kernel $\text{SINC}(x, y)$ (a) and nearest-neighbor kernel $W_{\text{nn}}(x, y)$ (b) for $-3 \leq x, y \leq 3$.

$$\tilde{I}(x, y) = I(u_x, v_y), \quad (22.28)$$

with $u_x = \text{round}(x) = \lfloor x + 0.5 \rfloor$ und $v_y = \text{round}(y) = \lfloor y + 0.5 \rfloor$.

As in the 1D case, the interpolation in 2D can be described as a linear convolution (linear filter). The 2D kernel for the nearest-neighbor interpolation is, analogous to Eqn. (22.9), defined as

$$W_{\text{nn}}(x, y) = \begin{cases} 1 & \text{for } -0.5 \leq x, y < 0.5, \\ 0 & \text{otherwise.} \end{cases} \quad (22.29)$$

This function is shown in Fig. 22.15(b). Nearest-neighbor interpolation is known for its strong blocking effects (Fig. 22.16(b)) and thus is rarely used for geometric image operations. However, in some situations, this effect may be intended; for example, if an image is to be enlarged by replicating each pixel without any smoothing.

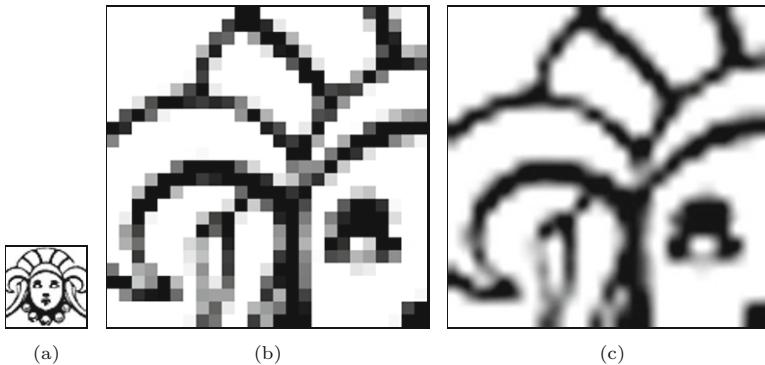


Fig. 22.16
Image enlargement example. Original (a); 8× enlargement using nearest-neighbor interpolation (b) and bicubic interpolation (c).

22.5.2 Bilinear Interpolation

The 2D counterpart to the linear interpolation in 1D (see Sec. 22.1) is the so-called *bilinear* interpolation,⁵ whose operation is illustrated in Fig. 22.17. For the given interpolation point (x, y) , we first find the four closest (surrounding) pixel values,

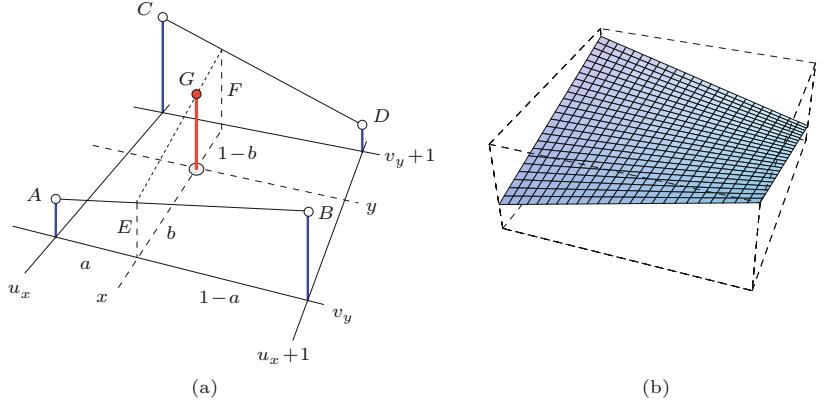
$$\begin{aligned} A &= I(u_x, v_y), & B &= I(u_x + 1, v_y), \\ C &= I(u_x, v_y + 1), & D &= I(u_x + 1, v_y + 1), \end{aligned} \quad (22.30)$$

⁵ Not to be confused with the bilinear *mapping* (transformation) described in Chapter 21, Sec. 21.1.5.

Fig. 22.17

Bilinear interpolation. For a given position (x, y) , the interpolated value is computed from the values A, B, C, D of the four closest pixels in two steps

(a). First the intermediate values E and F are computed by linear interpolation in the horizontal direction between A, B and C, D , respectively, where $a = x - u_x$ is the distance to the nearest pixel to the left of x . Subsequently, the intermediate values E, F are interpolated in the vertical direction, where $b = y - v_y$ is the distance to the nearest pixel below y . An example for the resulting surface between four adjacent pixels is shown in (b).



where $u_x = \lfloor x \rfloor$ and $v_x = \lfloor y \rfloor$. Then the pixel values A, B, C, D are interpolated in horizontal and subsequently in vertical direction. The intermediate values E, F are calculated from the distance $a = (x - u_x)$ of the specified interpolation position (x, y) from the discrete raster coordinate u_x as

$$E = A + (x - u_x) \cdot (B - A) = A + a \cdot (B - A), \quad (22.31)$$

$$F = C + (x - u_x) \cdot (D - C) = C + a \cdot (D - C), \quad (22.32)$$

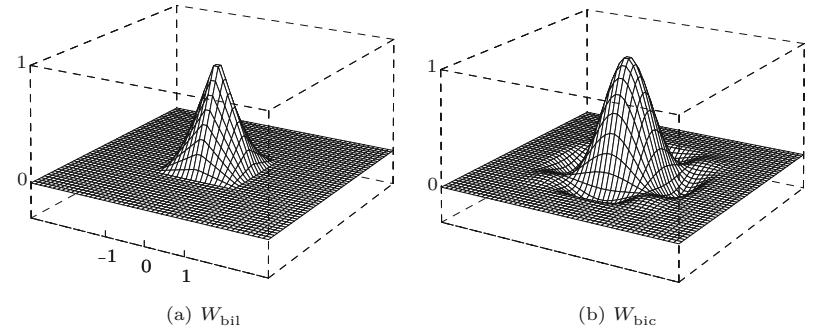
and the final interpolation value G is computed from the vertical distance $b = y_0 - v_y$ as

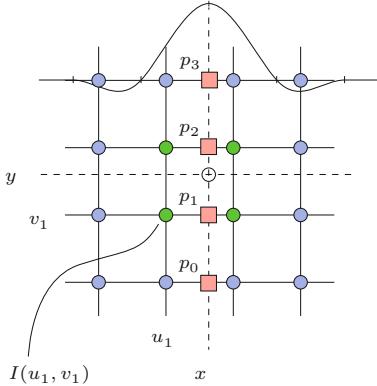
$$\begin{aligned} \tilde{I}(x, y) &= G = E + (y - v_y) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a - 1)(b - 1)A + a(1 - b)B + (1 - a)bC + abD. \end{aligned} \quad (22.33)$$

Expressed as a linear convolution filter, the corresponding 2D kernel $W_{\text{bil}}(x, y)$ is the product of the two 1D kernels $w_{\text{lin}}(x)$ and $w_{\text{lin}}(y)$ (Eqn. (22.10)), that is,

$$\begin{aligned} W_{\text{bilin}}(x, y) &= w_{\text{lin}}(x) \cdot w_{\text{lin}}(y) \\ &= \begin{cases} 1 - x - y + x \cdot y & \text{for } 0 \leq |x|, |y| < 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (22.34)$$

In this function (plotted in Fig. 22.18), we can recognize the bilinear term that gives this method its name.

Fig. 22.18
 2D interpolation kernels. bilinear kernel $W_{\text{bil}}(x, y)$ (a)
 and bicubic kernel $W_{\text{bic}}(x, y)$ (b) for $-3 \leq x, y \leq 3$.




22.5.3 Bicubic and Spline Interpolation in 2D

The convolution kernel for the 2D cubic interpolation is also defined as the product of the corresponding 1D kernels (Eqn. (22.12)),

$$W_{\text{bic}}(x, y) = w_{\text{cub}}(x) \cdot w_{\text{cub}}(y). \quad (22.35)$$

The resulting kernel is plotted in Fig. 22.18(b). Due to the decomposition into 1D kernels (Eqn. (22.13)), the computation of the bicubic interpolation is *separable* in x, y and can thus be expressed as

$$\tilde{I}(x, y) = \sum_{\substack{v=\lfloor y \rfloor - 1 \\ \lfloor y \rfloor - 1}}^{\lfloor y \rfloor + 2} \left[\sum_{\substack{u=\lfloor x \rfloor - 1 \\ \lfloor x \rfloor - 1}}^{\lfloor x \rfloor + 2} I(u, v) \cdot W_{\text{bic}}(x - u, y - v) \right] \quad (22.36)$$

$$= \sum_{j=0}^3 \left[w_{\text{cub}}(y - v_j) \cdot \underbrace{\sum_{i=0}^3 I(u_i, v_j) \cdot w_{\text{cub}}(x - u_i)}_{p_j} \right], \quad (22.37)$$

with $u_i = \lfloor x_0 \rfloor - 1 + i$ and $v_j = \lfloor y_0 \rfloor - 1 + j$. The quantity p_j is the intermediate result of the cubic interpolation in the x direction in line j , as illustrated in Fig. 22.19. Equation (22.37) describes a simple and efficient procedure for computing the bicubic interpolation using only a 1D kernel $w_{\text{cub}}(x)$. The interpolation is based on a 4×4 neighborhood of pixels and requires a total of $16 + 4 = 20$ additions and multiplications.

This method, which is summarized in Alg. 22.1, can be used to implement any x/y -separable 2D interpolation kernel of size 4×4 , such as the 2D *Catmull-Rom* interpolation (Eqn. (22.17)) with

$$W_{\text{crm}}(x, y) = w_{\text{crm}}(x) \cdot w_{\text{crm}}(y) \quad (22.38)$$

or the *Mitchell-Netravali* interpolation (Eqn. (22.19)) with

$$W_{\text{mn}}(x, y) = w_{\text{mn}}(x) \cdot w_{\text{mn}}(y). \quad (22.39)$$

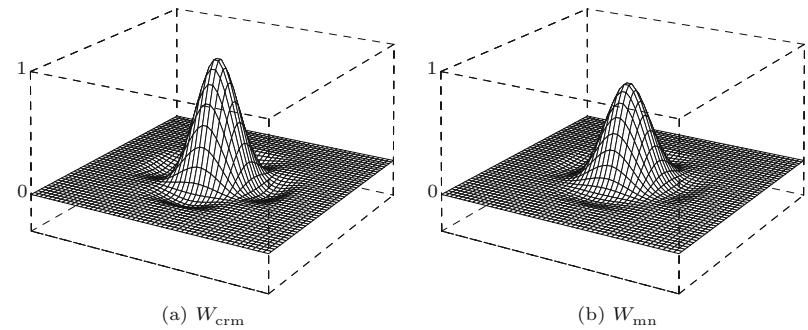
22.5 INTERPOLATION IN 2D

Fig. 22.19

Bicubic interpolation in two steps. The discrete image I (pixel positions correspond to raster lines) is to be interpolated at some continuous position (x, y) . In step 1 (left), a 1D interpolation is performed in the horizontal direction with $w_{\text{cub}}(x)$ over four pixels $I(u_i, v_j)$ in four lines. One intermediate result p_j (marked \square) is computed for each line j . In step 2 (right), the result $\tilde{I}(x_0, y_0)$ is computed by a single cubic interpolation in the vertical direction over the intermediate results p_0, \dots, p_3 . In total, $16 + 4 = 20$ interpolation steps are required.

22 PIXEL INTERPOLATION

Fig. 22.20
2D spline interpolation kernels: Catmull-Rom kernel $W_{\text{crm}}(x, y)$ (a), Mitchell-Netravali kernel $W_{\text{mn}}(x, y)$ (b), for $-3 \leq x, y \leq 3$.



Alg. 22.1

Bicubic interpolation of image I at position (x, y) . The 1D cubic function $w_{\text{cub}}(\cdot)$ (Eqn. (22.11)) is used for the separate interpolation in the x and y directions based on a neighborhood of 4×4 pixels. See Prog. 22.1 for a straightforward implementation in Java.

```

1: BicubicInterpolation( $I, x, y, a$ )
   Input:  $I$ , original image;  $x, y \in \mathbb{R}$ , continuous position;  $a$ , control parameter. Returns the interpolated image value at position  $(x, y)$ .
2:  $q \leftarrow 0$ 
3: for  $j \leftarrow 0, \dots, 3$  do ▷ iterate over 4 lines
4:    $v \leftarrow \lfloor y \rfloor - 1 + j$ 
5:    $p \leftarrow 0$ 
6:   for  $i \leftarrow 0, \dots, 3$  do ▷ iterate over 4 columns
7:      $u \leftarrow \lfloor x \rfloor - 1 + i$ 
8:      $p \leftarrow p + I(u, v) \cdot w_{\text{cub}}(x-u, a)$  ▷ see Eq. 22.11
9:    $q \leftarrow q + p \cdot w_{\text{cub}}(y-v, a)$ 
10: return  $q$ 

```

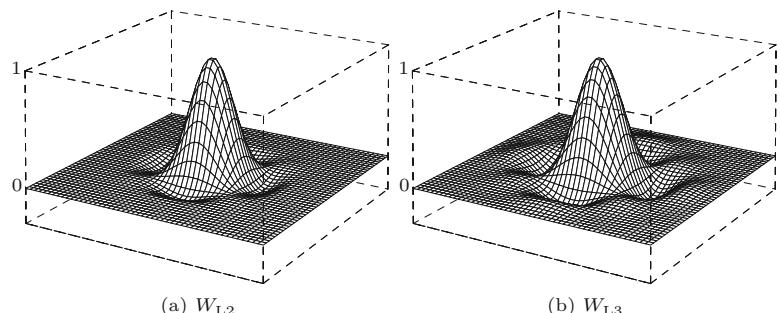
22.5.4 Lanczos Interpolation in 2D

The kernels for the 2D Lanczos interpolation are also x/y -separable into 1D kernels (see Eqns. (22.24) and (22.25), respectively), that is,

$$W_{\text{Ln}}(x, y) = w_{\text{Ln}}(x) \cdot w_{\text{Ln}}(y). \quad (22.40)$$

The resulting kernels for orders $n = 2$ and $n = 3$ are shown in Fig. 22.21. Because of the separability the 2D Lanczos interpolation can be computed, similar to the bicubic interpolation, separately in the x and y directions. Like the bicubic kernel, the 2-tap Lanczos kernel W_{L2} (Eqn. (22.24)) is zero outside the interval $-2 \leq x, y \leq 2$, and thus the procedure described in Eqn. (22.37) and Alg. 22.1 can be used with only a small modification (replace w_{cub} by w_{L2}).

Fig. 22.21
2D Lanczos kernels for $n = 2$ and $n = 3$: kernels $W_{\text{L2}}(x, y)$ (a) and $W_{\text{L3}}(x, y)$ (b), with $-3 \leq x, y \leq 3$.



1: **SeparableInterpolation**(I, x, y, w, n)

Input: I , original image; $x, y \in \mathbb{R}$, continuous position; w , a 1D interpolation kernel of extent $\pm n$ ($n \geq 1$).

Returns the interpolated image value at position (x, y) using the composite interpolation kernel $W(x, y) = w(x) \cdot w(y)$.

```

2:    $q \leftarrow 0$ 
3:   for  $j \leftarrow 0, \dots, 2n-1$  do                                 $\triangleright$  iterate over  $2n$  lines
4:      $v \leftarrow \lfloor y \rfloor - n + 1 + j$                                  $\triangleright = v_j$ 
5:      $p \leftarrow 0$ 
6:     for  $i \leftarrow 0, \dots, 2n-1$  do                                 $\triangleright$  iterate over  $2n$  columns
7:        $u \leftarrow \lfloor x \rfloor - n + 1 + i$                                  $\triangleright = u_i$ 
8:        $p \leftarrow p + I(u, v) \cdot w(x - u)$ 
9:      $q \leftarrow q + p \cdot w(y - v)$ 
10:    return  $q$ 
```

22.5 INTERPOLATION IN 2D

Alg. 22.2

General interpolation with a separable interpolation kernel $W(x, y) = w_n(x) \cdot w_n(y)$ of extent $\pm n$ (i.e., the 1D kernel $w_n(x)$ is zero for $x < -n$ and $x > n$, with $n \in \mathbb{N}$). Note that procedure **BicubicInterpolation** in Alg. 22.1 is a special instance of this algorithm (with $n = 2$).

Compared to Eqn. (22.37), the larger Lanczos kernel W_{L3} (Eqn. (22.25)) requires two additional pixel rows and columns. The calculation of the interpolated pixel value at position (x, y) thus has the form

$$\tilde{I}(x, y) = \sum_{\substack{v= \\ \lfloor y \rfloor - 2}}^{\lfloor y \rfloor + 3} \left[\sum_{\substack{u= \\ \lfloor x \rfloor - 2}}^{\lfloor x \rfloor + 3} I(u, v) \cdot W_{L3}(x - u, y - v) \right] \quad (22.41)$$

$$= \sum_{j=0}^5 \left[w_{L3}(y - v_j) \cdot \sum_{i=0}^5 I(u_i, v_j) \cdot w_{L3}(x - u_i) \right], \quad (22.42)$$

with $u_i = \lfloor x \rfloor - 2 + i$ and $v_j = \lfloor y \rfloor - 2 + j$. Thus the L3 Lanczos interpolation in 2D uses a support region of $6 \times 6 = 36$ pixels from the original image, 20 pixels more than the bicubic interpolation.

In general, the expression for a 2D Lanczos interpolator L_n of arbitrary order $n \geq 1$ is

$$\tilde{I}(x, y) = \sum_{\substack{v= \\ \lfloor y \rfloor - n + 1}}^{\lfloor y \rfloor + n} \left[\sum_{\substack{u= \\ \lfloor x \rfloor - n + 1}}^{\lfloor x \rfloor + n} [I(u, v) \cdot W_{Ln}(x - u, y - v)] \right] \quad (22.43)$$

$$= \sum_{j=0}^{2n-1} \left[w_{Ln}(y - v_j) \cdot \sum_{i=0}^{2n-1} [I(u_i, v_j) \cdot w_{Ln}(x - u_i)] \right], \quad (22.44)$$

with $u_i = \lfloor x \rfloor - n + 1 + i$ and $v_j = \lfloor y \rfloor - n + 1 + j$. The size of this interpolator's support region is $2n \times 2n$ pixels. How the expression in Eqn. (22.44) could be computed is shown in Alg. 22.2, which actually describes a general interpolation procedure that can be used with any separable interpolation kernel $W(x, y) = w_n(x) \cdot w_n(y)$ of extent $\pm n$.

22.5.5 Examples and Discussion

Figures 22.22 and 22.23 compare the interpolation methods described in this section: nearest-neighbor, bilinear, bicubic Catmull-Rom, cubic B-spline, Mitchell-Netravali, and Lanczos interpolation. In both figures, the original images are rotated counter-clockwise by 15° . A

gray background is used to visualize the edge overshoot produced by some of the interpolators.

Nearest-neighbor interpolation (Fig. 22.22(b)) creates no new pixel values but forms, as expected, coarse blocks of pixels with the same intensity.

The effect of the *bilinear* interpolation (Fig. 22.22(c)) is local smoothing over four neighboring pixels. The weights for these four pixels are positive, and thus no result can be smaller than the smallest neighboring pixel value or greater than the greatest neighboring pixel value. In other words, bilinear interpolation cannot create any over- or undershoot at edges.

This is not the case for the *bicubic* interpolation (Fig. 22.22(d)): some of the coefficients in the bicubic interpolation kernel are negative, which makes pixels near edges clearly brighter or darker, respectively, thus increasing the perceived sharpness. In general, bicubic interpolation produces clearly better results than the bilinear method at comparable computing cost, and it is thus widely accepted as the standard technique and used in most image manipulation programs. By adjusting the control parameter a (Eqn. (22.11)), the bicubic kernel can be easily tuned to fit the need of particular applications. For example, the *Catmull-Rom* method (Fig. 22.22(e)) can be implemented with the bicubic interpolation by setting $a = 0.5$ (Eqns. (22.17) and (22.38)).

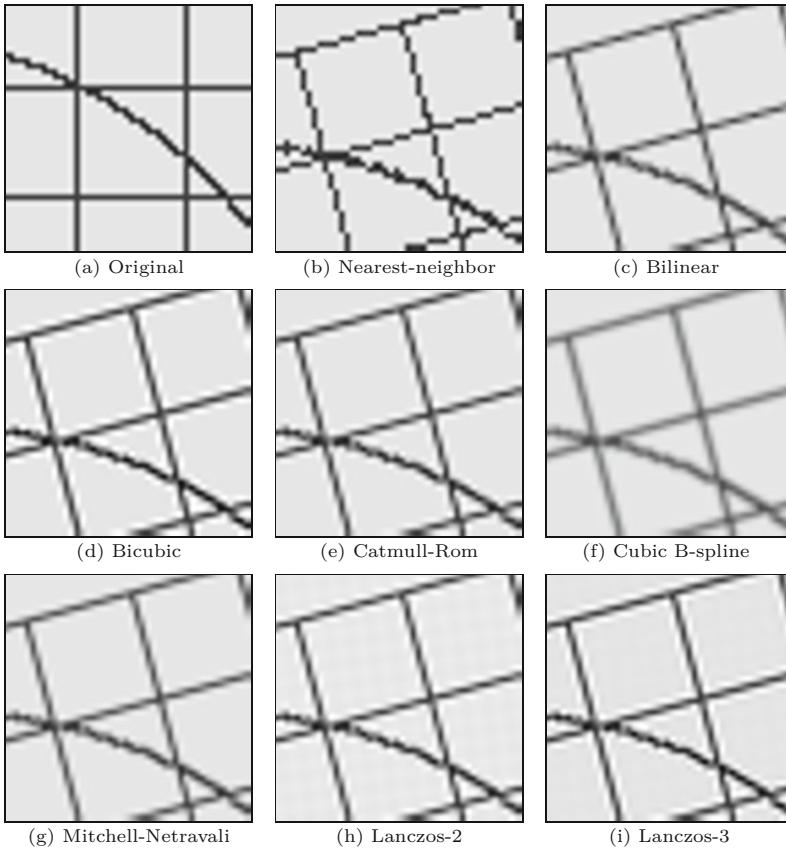
Results from the 2D *Lanczos* interpolation (Fig. 22.22(h)) using the 2-tap kernel W_{L2} cannot be much better than from the bicubic interpolation, which can be adjusted to give similar results without causing any ringing in flat regions, as seen in Fig. 22.14. The 3-tap Lanczos kernel W_{L3} (Fig. 22.22(i)) on the other hand should produce slightly sharper edges at the cost of increased overshoot (see also Exercise 22.3).

In summary, for high-quality applications one should consider the *Catmull-Rom* (Eqns. (22.17) and (22.38)) or the *Mitchell-Netravali* (Eqns. (22.19) and (22.39)) methods, which offer good reconstruction at the same computational cost as the bicubic interpolation.

22.6 Aliasing

As we described in the main part of this chapter, the usual approach for implementing geometric image transformations can be summarized by the following three steps (Fig. 22.24):

1. Each discrete image point (u', v') of the *target* image is projected by the inverse geometric transformation T^{-1} to the continuous coordinate (x, y) in the source image.
2. The continuous image function $\tilde{I}(x, y)$ is reconstructed from the discrete source image $I(u, v)$ by interpolation (using one of the methods described earlier).
3. The interpolated function is sampled at position (x, y) , and the sample value $\tilde{I}(x, y)$ is transferred to the target pixel $I'(u', v')$.



22.6 ALIASING

Fig. 22.22

Image interpolation methods compared (line art).

22.6.1 Sampling the Interpolated Image

One problem not considered so far concerns the process of sampling the reconstructed, continuous image function in the aforementioned step 3. The problem occurs when the geometric transformation T causes parts of the image to be *contracted*. In this case, the distance between adjacent sample points on the source image is locally *increased* by the corresponding inverse transformation T^{-1} . Now, widening the sampling distance reduces the spatial sampling rate and thus the maximum permissible frequencies in the reconstructed image function $\tilde{I}(x, y)$. Eventually this leads to a violation of the sampling criterion and causes visible aliasing in the transformed image. The problem does not occur when the image is enlarged by the geometric transformation because in this case the sampling interval on the source image is shortened (corresponding to a higher sampling frequency) and no aliasing can occur.

Note that this effect is largely unrelated to the interpolation method, as demonstrated by the examples in Fig. 22.25. The effect is most noticeable under nearest-neighbor interpolation in Fig. 22.25(b), where the thin lines are simply not “hit” by the widened sampling raster and thus disappear in some places. Important image information is thereby lost. The bilinear and bicubic interpolation methods in Fig. 22.25(c, d) have wider interpolation kernels but still

Fig. 22.23

Image interpolation methods compared (text image).

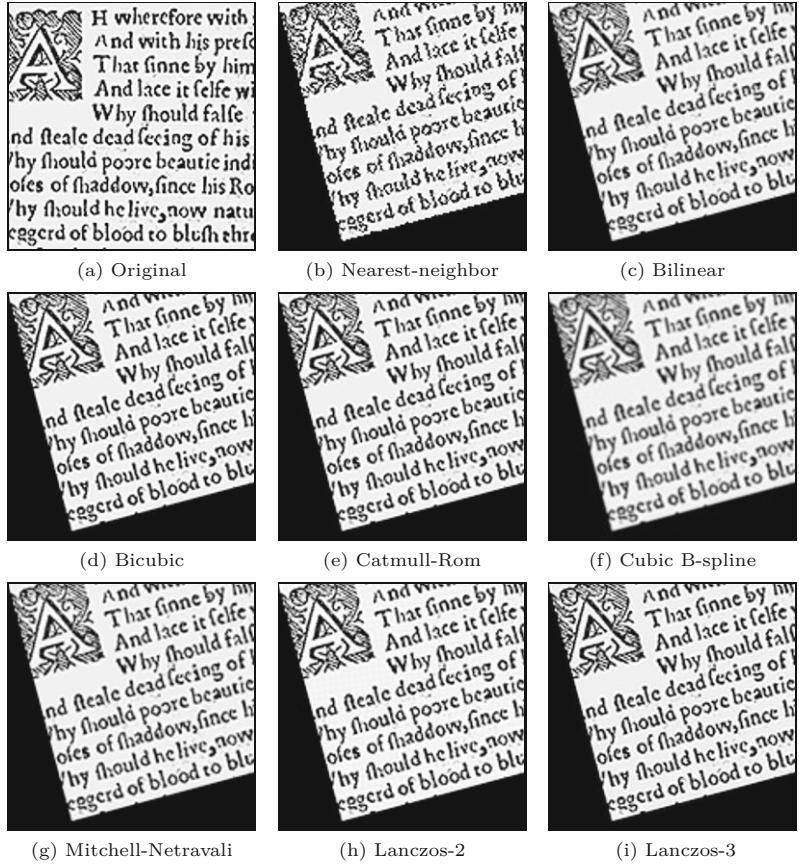
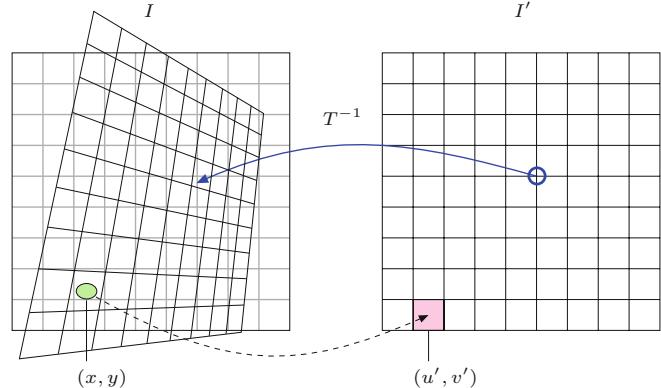


Fig. 22.24

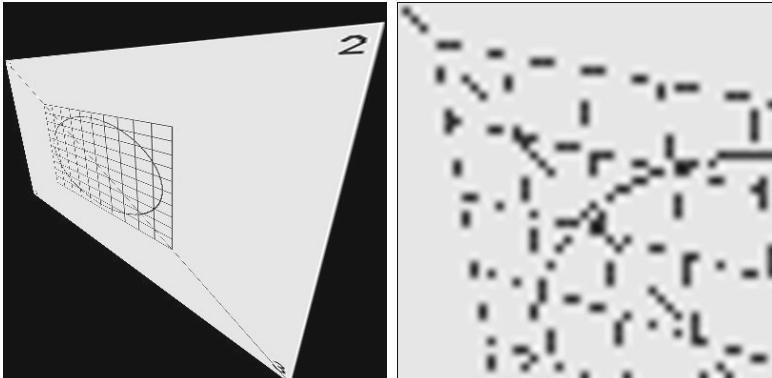
Sampling errors in geometric operations. If the geometric transformation T leads to a local contraction of the image (which corresponds to a local enlargement by T^{-1}), the distance between adjacent sample points in I is increased. This reduces the local sampling frequency and thus the maximum signal frequency allowed in the source image, which eventually leads to aliasing.



cannot avoid the aliasing effect. The problem of course gets worse with increasing reduction factors.

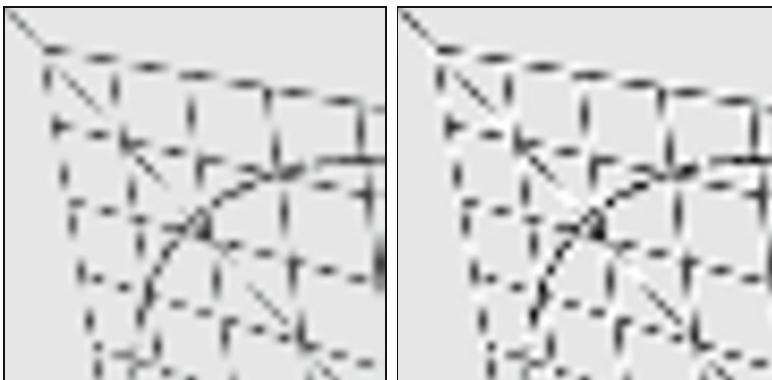
22.6.2 Low-Pass Filtering

One solution to the aliasing problem is to make sure that the interpolated image function is properly frequency-limited before it gets



(a)

(b)



(c)

(d)

22.6 ALIASING

Fig. 22.25

Aliasing caused by local image contraction. Aliasing is caused by a violation of the sampling criterion and is largely unaffected by the interpolation method used: complete transformed image (a), detail using nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

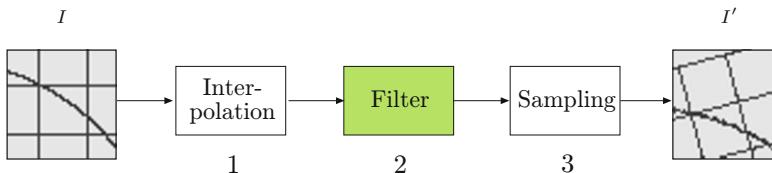


Fig. 22.26

Low-pass filtering to avoid aliasing in geometric operations. After interpolation (step 1), the reconstructed image function is subjected to low-pass filtering (step 2) before being resampled (step 3).

resampled. This can be accomplished with a suitable low-pass filter, as illustrated in Fig. 22.26.

The cutoff frequency of the low-pass filter is determined by the amount of local scale change, which may—depending upon the type of transformation—be different in various parts of the image. In the simplest case the amount of scale change is the same throughout the image (e.g., under global scaling or affine transformations, where the same filter can be used everywhere in the image). In general, however, the low-pass filter is *space-variant* or *nonhomogeneous*, and the local filter parameters are determined by the transformation T and the current image position. If convolution filters are used for both interpolation and low-pass filtering, they could be combined into a common, space-variant reconstruction filter.

Unfortunately, space-variant filtering is computationally expensive and thus is often avoided, even in professional applications (e.g., Adobe Photoshop). The technique is nevertheless used in certain ap-

plications, such as high-quality texture mapping in computer graphics [75, 105, 256]. Integral images, as described in Chapter 3, Sec. 3.8, can be used to implement efficient space-variant smoothing filters.

22.7 Java Implementation

Implementations of most interpolation methods described in this chapter are openly available as part of the `imagingbook` library.⁶ The following interpolators are available as subclasses of the abstract class `PixelInterpolator`:

```
BicubicInterpolator,  
BilinearInterpolator,  
LanczosInterpolator,  
NearestNeighborInterpolator,  
SplineInterpolator.
```

For illustration, the complete implementation of the class `BicubicInterpolator` is shown in Prog. 22.1.

`PixelInterpolator` (class)

This class provides the functionality for interpolating images with scalar pixel values. It defines the following methods:

```
static PixelInterpolator create (InterpolationMethod  
im)
```

Factory method which creates and returns a new interpolator. Admissible values for the parameter `im` and associated interpolator types (subclasses of `ScalarInterpolator`) are listed in Table 22.1.

```
float getInterpolatedValue (ImageAccessor.Scalar ia,  
double x, double y)
```

Returns the interpolated pixel value at the continuous position `x`, `y` of the scalar-valued image (referenced by the image accessor `ia`).

Table 22.1

Admissible values for `InterpolationMethod` and associated interpolator types returned by the static `create(im)` method of `PixelInterpolator`.

InterpolationMethod im	Interpolator Type
NearestNeighbor	<code>NearestNeighborInterpolator()</code>
Bilinear	<code>BilinearInterpolator()</code>
Bicubic	<code>BicubicInterpolator(1.00)</code>
BicubicSmooth	<code>BicubicInterpolator(0.25)</code>
BicubicSharp	<code>BicubicInterpolator(1.75)</code>
CatmullRom	<code>SplineInterpolator(0.5, 0.0)</code>
CubicBSpline	<code>SplineInterpolator(0.0, 1.0)</code>
MitchellNetravali	<code>SplineInterpolator(1.0/3, 1.0/3)</code>
Lanzcos2	<code>LanczosInterpolator(2)</code>
Lanzcos3	<code>LanczosInterpolator(3)</code>
Lanzcos4	<code>LanczosInterpolator(4)</code>

⁶ Package `imagingbook.lib.interpolation`.

```

1 package imagingbook.lib.interpolation;
2
3 import imagingbook.lib.image.ImageAccessor;
4 import java.awt.geom.Point2D;
5
6 public class BicubicInterpolator
7     extends PixelInterpolator {
8
9     private final double a; // sharpness value
10
11    public BicubicInterpolator() {
12        this(0.5);
13    }
14
15    public BicubicInterpolator(double a) {
16        this.a = a;
17    }
18
19    public float getInterpolatedValue(
20        ImageAccessor.Scalar ia, double x, double y) {
21        final int u0 = (int) Math.floor(x);
22        final int v0 = (int) Math.floor(y);
23        double q = 0;
24        for (int j = 0; j <= 3; j++) {
25            int v = v0 - 1 + j;
26            double p = 0;
27            for (int i = 0; i <= 3; i++) {
28                int u = u0 - 1 + i;
29                float pixval = ia.getVal(u, v);
30                p = p + pixval * w_cub(x - u, a);
31            }
32            q = q + p * w_cub(y - v, a);
33        }
34        return (float) q;
35    }
36
37    private final double w_cub(double x, double a) {
38        if (x < 0)
39            x = -x;
40        double z = 0;
41        if (x < 1)
42            z = (-a + 2) * x * x * x + (a - 3) * x * x + 1;
43        else if (x < 2)
44            z = -a * x * x * x + 5 * a * x * x
45            - 8 * a * x + 4 * a;
46        return z;
47    }

```

22.7 JAVA IMPLEMENTATION

Prog. 22.1

Java implementation of bicubic interpolation (class `BicubicInterpolator`), as defined in Alg. 22.1. The class provides two constructors: a default constructor (line 11) with sharpness value $a = 0.5$ and a general constructor for arbitrary a (line 14). The actual pixel interpolation is performed by method `getInterpolatedValue()` in line 18, which implements Alg. 22.1. `w_cub()` in line 36 is the 1D cubic interpolation function (see Eqn. (22.11)).

The class `PixelInterpolator` is primarily used by the methods in class `ImageAccessor`.⁷ See Prog. 22.2 for a basic usage example.

⁷ The `ImageAccessor` class (in package `imagingbook.lib.image`) provides unified access to all types of images available in ImageJ and also supports pixel interpolation.

22 PIXEL INTERPOLATION

Prog. 22.2

Image interpolation example using class `ImageAccessor`. This ImageJ plugin translates the input image by some (non-integer) distance `dx`, `dy`. It uses target-to-source mapping and pixel interpolation of type `BicubicSharp` (see Table 22.1).

The required `ImageAccessor` (interpolator) object for the source image is created in line 31, another for the target image in line 34. This is followed by an iteration over all pixels of the target image. The source image is interpolated (line 41) at the calculated positions (`x`, `y`) and the resulting `float[]` value is inserted into the target image with `setPix()` in line 42. Note that this plugin is generic, that is, it works for all image types.

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import imagingbook.lib.image.ImageAccessor;
5 import imagingbook.lib.image.OutOfBoundsStrategy;
6 import static imagingbook.lib.image.OutOfBoundsStrategy.*;
7 import imagingbook.lib.interpolation.InterpolationMethod;
8 import static imagingbook.lib.interpolation.
     InterpolationMethod.*;
9
10 public class Interpolator_Demo implements PlugInFilter {
11
12     static double dx = 0.5; // translation
13     static double dy = -3.5;
14
15     static OutOfBoundsStrategy OBS = NearestBorder;
16     static InterpolationMethod IPM = BicubicSharp;
17
18     public int setup(String arg, ImagePlus imp) {
19         return DOES_ALL + NO_CHANGES;
20     }
21
22     public void run(ImageProcessor source) {
23         final int w = source.getWidth();
24         final int h = source.getHeight();
25
26         // create the target image (same type as source):
27         ImageProcessor target = source.createProcessor(w, h);
28
29         // create an ImageAccessor for the source image:
30         ImageAccessor sA =
31             ImageAccessor.create(source, OBS, IPM);
32
33         // create an ImageAccessor for the target image:
34         ImageAccessor tA = ImageAccessor.create(target);
35
36         // iterate over all pixels of the target image:
37         for (int u = 0; u < w; u++) {
38             for (int v = 0; v < h; v++) {
39                 double x = u + dx; // continuous source position (x,y)
40                 double y = v + dy;
41                 float[] val = sA.getPix(x, y);
42                 tA.setPix(u, v, val); // update the target pixel
43             }
44         }
45
46         // display the target image:
47         (new ImagePlus("Target", target)).show();
48     }
49 }
```

Exercise 22.1. The 1D interpolation function by Mitchell and Nárayi $w_{mn}(x)$ is defined as a general spline function $w_{cs}(x, a, b)$ (Eqn. (22.19)). Show that this function can be expressed as the weighted sum of a Catmull-Rom function $w_{crm}(x)$ (Eqn. (22.17)) and a cubic B-spline $w_{cbs}(x)$ (Eqn. (22.18)) in the form

$$\begin{aligned} w_{mn}(x) &= w_{cs}\left(x, \frac{1}{3}, \frac{1}{3}\right) \\ &= \frac{1}{3} \cdot [2 \cdot w_{cs}(x, 0.5, 0) + w_{cs}(x, 0, 1)] \\ &= \frac{1}{3} \cdot [2 \cdot w_{crm}(x) + w_{cbs}(x)]. \end{aligned} \quad (22.45)$$

Exercise 22.2. Implement an “ideal” (low-pass) pixel interpolator based on the Sinc function (see Eqn. (22.5)). Assume that the image function is periodic along both coordinate axes. Determine (by truncating the Sinc function at $\pm N$) the minimum number of samples to include and if the result improves by including additional samples. Use the class `BicubicInterpolator` (Prog. 22.1) as a template for your implementation.

Exercise 22.3. Implement the 2D *Lanczos* interpolation with a W_{L3} kernel, as defined in Eqn. (22.42), as a Java class analogous to the class `BicubicInterpolator` (Prog. 22.1). Compare the results to the bicubic interpolation.

Exercise 22.4. The 1D Lanczos interpolation kernel of order $n = 4$ is (analogous to Eqn. (22.25)) defined as

$$w_{L4} = \begin{cases} 4 \cdot \frac{\sin(\pi \frac{x}{4}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 \leq |x| < 4, \\ 0 & \text{for } |x| \geq 4. \end{cases} \quad (22.46)$$

Extend the 2D kernel in Eqn. (22.42) to w_{L4} and implement this interpolator as a Java class analogous to `BicubicInterpolator` (Prog. 22.1). How many image pixels does the calculation include at each position? See if there is any noticeable improvement over the bicubic and the Lanczos-3 interpolation (Exercise 22.3).

Image Matching and Registration

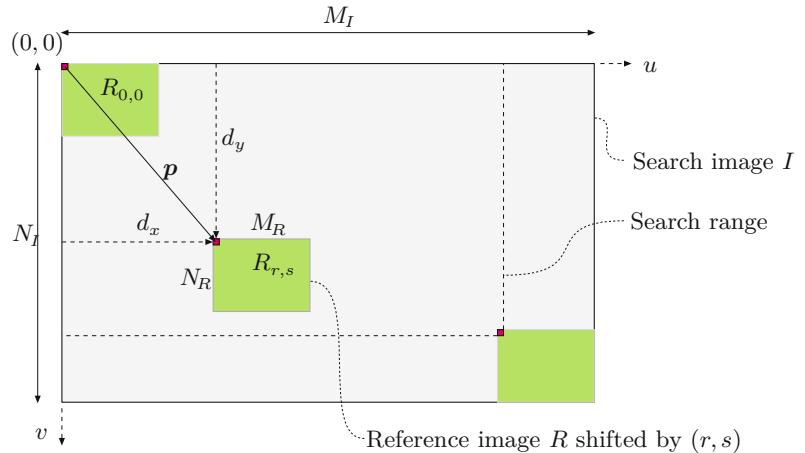
When we compare two images, we are faced with the following basic question: when are two images the same or similar, and how can this similarity be measured? Of course one could trivially define two images I_1 , I_2 as being identical when all pixel values are the same (i.e., the difference $I_1 - I_2$ is zero). Although this kind of definition may be useful in specific applications, such as for detecting changes in successive images under constant lighting and camera conditions, simple pixel differencing is usually too inflexible to be of much practical use. Noise, quantization errors, small changes in lighting, and minute shifts or rotations can all create large numerical pixel differences for pairs of images that would still be perceived as perfectly identical by a human viewer. Obviously, human perception incorporates a much wider concept of similarity and uses cues such as structure and content to recognize similarity between images, even when a direct comparison between individual pixels would not indicate any match. The problem of comparing images at a structural or semantic level is a difficult problem and an interesting research field, for example, in the context of image-based searches on the Internet or database retrieval.

This chapter deals with the much simpler problem of comparing images at the pixel level; in particular, localizing a given subimage—often called a “template”—within some larger image. This task is frequently required, for example, to find matching patches in stereo images, to localize a particular pattern in a scene, or to track a certain pattern through an image sequence. The principal idea behind “template matching” is simple: move the given pattern (template) over the search image, measure the difference against the corresponding subimage at each position, and record those positions where the highest similarity is obtained. But this is not as simple as it may initially sound. After all, what is a suitable distance measure, what total difference is acceptable for a match, and what happens when brightness or contrast changes?

We already touched on this problem of invariance under geometric transformations when we discussed the shape properties of seg-

Fig. 23.1

Geometry of template matching. The reference image R is shifted across the search image I by an offset (r, s) using the origins of the two images as the reference points. The dimensions of the search image ($M_I \times N_I$) and the reference image ($M_R \times N_R$) determine the maximal search region for this comparison.



mented regions in Chapter 10, Sec. 10.4.2. However, geometric invariance is not our main concern in the remaining part of this chapter, where we describe only the most basic template-matching techniques: correlation-based methods for intensity images and “chamfer-matching” for binary images.

23.1 Template Matching in Intensity Images

First we look at the problem of localizing a given *reference image* (template) R within a larger intensity (grayscale) image I , which we call the *search image*. The task is to find those positions where the contents of the reference image R and the corresponding subimage of I are either the same or most similar. If we denote by

$$R_{r,s}(u, v) = R(u - r, v - s) \quad (23.1)$$

the reference image R shifted by the distance (r, s) in the horizontal and vertical directions, respectively, then the matching problem (illustrated in Fig. 23.1) can be summarized as follows:

- Given are the search image I and the reference image R . Find the offset $(r, s) \in \mathbb{Z}^2$ such that the similarity between the shifted reference image $R_{r,s}$ and the corresponding subimage of I is a maximum.

To successfully solve this task, several issues need to be addressed, such as determining a minimum similarity value for accepting a match and developing a good search strategy for finding the optimal displacement. First, and most important, a suitable measure of similarity between subimages must be found that is reasonably tolerant against intensity and contrast variations.

23.1.1 Distance between Image Patterns

To quantify the amount of agreement, we compute a “distance” $d(r, s)$ between the shifted reference image R and the corresponding subimage of I for each offset position (r, s) (Fig. 23.2). Several distance

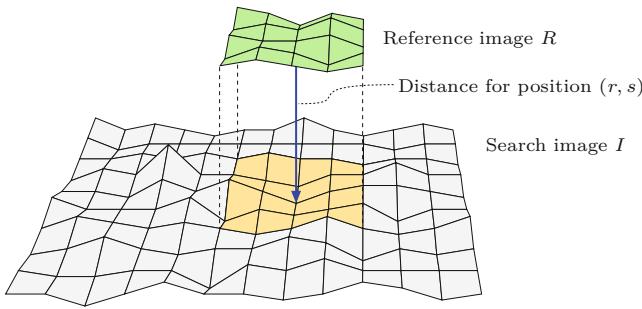


Fig. 23.2
Measuring the distance between 2D image functions. The reference image R is positioned at offset (r, s) on top of the search image I .

measures have been proposed for 2D intensity images, including the following three basic definitions:¹

Sum of absolute differences:

$$d_A(r, s) = \sum_{(i,j) \in R} |I(r+i, s+j) - R(i, j)|. \quad (23.2)$$

Maximum difference:

$$d_M(r, s) = \max_{(i,j) \in R} |I(r+i, s+j) - R(i, j)|. \quad (23.3)$$

Sum of squared differences:

$$d_E(r, s) = \left[\sum_{(i,j) \in R} (I(r+i, s+j) - R(i, j))^2 \right]^{1/2}. \quad (23.4)$$

Note that the expression in Eqn. (23.4) is nothing else but the *Euclidean distance* between two N -dimensional vectors of pixels values. Similarly, the sum of differences in Eqn. (23.2) is equivalent to the L_1 distance, and the maximum difference in Eqn. (23.3) equals the L_∞ distance norm.²

Distance and correlation

Because of its formal properties, the N -dimensional distance d_E (Eqn. (23.4)) is of special importance and well-known in statistics and optimization. To find the best-matching position between the reference image R and the search image I , it is sufficient to *minimize the square* of d_E (which is always positive), which can be expanded to

$$\begin{aligned} d_E^2(r, s) &= \sum_{(i,j) \in R} (I(r+i, s+j) - R(i, j))^2 \\ &= \underbrace{\sum_{(i,j) \in R} I^2(r+i, s+j)}_{A(r, s)} + \underbrace{\sum_{(i,j) \in R} R^2(i, j)}_{B} - 2 \cdot \underbrace{\sum_{(i,j) \in R} I(r+i, s+j) \cdot R(i, j)}_{C(r, s)}. \end{aligned} \quad (23.5)$$

¹ We use the short notation $(i, j) \in R$ to specify the set of all possible template coordinates, that is, $\{(i, j) \mid 0 \leq i < M_R, 0 \leq j < N_R\}$.

² See also Sec. B.1.2 in the Appendix.

Notice that the term B in Eqn. (23.5) is the sum of the squared pixel values in the reference image R , a constant value (independent of r, s) that can thus be ignored. The term $A(r, s)$ is the sum of the squared values within the subimage of I at the current offset (r, s) . $C(r, s)$ is the so-called *linear cross correlation* (\circledast) between I and R , which is defined in the general case as

$$(I \circledast R)(r, s) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(r+i, s+j) \cdot R(i, j), \quad (23.6)$$

which—since R and I are assumed to have zero values outside their boundaries—is, furthermore, equivalent to

$$\sum_{i=0}^{M_R-1} \sum_{j=0}^{N_R-1} I(r+i, s+j) \cdot R(i, j) = \sum_{(i,j) \in R} I(r+i, s+j) \cdot R(i, j) \quad (23.7)$$

and thus the same as $C(r, s)$ in Eqn. (23.5). As we can see in Eqn. (23.6), correlation is in principle the same operation as linear *convolution* (see Ch. 5, Eqn. (5.16)), with the only difference being that the convolution kernel ($R(i, j)$ in this case) is implicitly mirrored.

If we assume for a minute that $A(r, s)$ —the “signal energy”—in Eqn. (23.5) is constant throughout the image I , then $A(r, s)$ can also be ignored and the position of maximum cross correlation $C(r, s)$ coincides with the best match between R and I . In this case, the minimum of $d_E^2(r, s)$ (Eqn. (23.5)) can be found by computing the maximum value of the correlation $I \circledast R$ only. This could be interesting for practical reasons if we consider that the linear convolution (and thus the correlation) with large kernels can be computed very efficiently in the frequency domain (see also Ch. 19, Sec. 19.5).

Normalized cross correlation

Unfortunately, the assumption made earlier that $A(r, s)$ is constant does not hold for most images, and thus the result of the cross correlation strongly varies with intensity changes in the image I . The *normalized cross correlation* $C_N(r, s)$ compensates for this dependency by taking into account the energy in the reference image and the current subimage:

$$C_N(r, s) = \frac{C(r, s)}{\sqrt{A(r, s) \cdot B}} = \frac{C(r, s)}{\sqrt{A(r, s) \cdot \sqrt{B}}} \quad (23.8)$$

$$= \frac{\sum_{(i,j) \in R} I(r+i, s+j) \cdot R(i, j)}{\left[\sum_{(i,j) \in R} I^2(r+i, s+j) \right]^{1/2} \cdot \left[\sum_{(i,j) \in R} R^2(i, j) \right]^{1/2}}. \quad (23.9)$$

If the values in the search and reference images are all positive (which is usually the case), then the result of $C_N(r, s)$ is always in the range $[0, 1]$, independent of the remaining contents in I and R . In this case, the result $C_N(r, s) = 1$ indicates a maximum match between R and the current subimage of I at the offset (r, s) , while $C_N(r, s) = 0$ indicates no match at all.

0 signals no agreement. Thus the normalized correlation has the additional advantage of delivering a standardized match value that can be used directly (using a suitable threshold between 0 and 1) to decide about the acceptance or rejection of a match position.

In contrast to the “global” cross correlation in Eqn. (23.6), the expression in Eqn. (23.8) is a “local” distance measure. However, it, too, has the problem of measuring the *absolute* distance between the template and the subimage. If, for example, the overall intensity of the image I is altered, then even the result of the normalized cross correlation $C_N(r, s)$ may also change dramatically.

Correlation coefficient

One solution to this problem is to compare not the original function values but the differences with respect to the average value of R and the average of the current subimage of I . This modification turns Eqn. (23.8) into

$$C_L(r, s) = \frac{\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}_{r,s}) \cdot (R(i, j) - \bar{R})}{[\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}_{r,s})^2]^{1/2} \cdot [\underbrace{\sum_{(i,j) \in R} (R(i, j) - \bar{R})^2}_{S_R^2 = K \cdot \sigma_R^2}]^{1/2}}, \quad (23.10)$$

with the average values $\bar{I}_{r,s}$ and \bar{R} defined as

$$\bar{I}_{r,s} = \frac{1}{K} \cdot \sum_{(i,j) \in R} I(r+i, s+j) \quad \text{and} \quad \bar{R} = \frac{1}{K} \cdot \sum_{(i,j) \in R} R(i, j), \quad (23.11)$$

respectively, ($K = |R|$ being the size of the reference image R). In statistics, the expression in Eqn. (23.10) is known as the *correlation coefficient*. However, different from the usual application as a global measure in statistics, $C_L(r, s)$ describes a *local*, piecewise correlation between the template R and the current subimage (at offset r, s) of I . The resulting values of $C_L(r, s)$ are in the range $[-1, 1]$ regardless of the contents in R and I . Again a value of 1 indicates maximum agreement between the compared image patterns, while -1 corresponds to a maximum mismatch. The term

$$S_R^2 = K \cdot \sigma_R^2 = \sum_{(i,j) \in R} (R(i, j) - \bar{R})^2 \quad (23.12)$$

in the denominator of Eqn. (23.10) is K times the *variance* (σ_R^2) of the values in the template R , which is constant and thus needs to be computed only once. Due to the fact that $\sigma_R^2 = \frac{1}{K} \sum R^2(i, j) - \bar{R}^2$, the expression in Eqn. (23.12) can be reformulated as

$$S_R^2 = \sum_{(i,j) \in R} R^2(i, j) - K \cdot \bar{R}^2 \quad (23.13)$$

$$= \sum_{(i,j) \in R} R^2(i, j) - \frac{1}{K} \cdot [\sum_{(i,j) \in R} R(i, j)]^2. \quad (23.14)$$

By inserting the results from Eqns. (23.11) and (23.14) we can rewrite Eqn. (23.10) as

$$C_L(r, s) = \frac{\sum_{(i,j) \in R} (I(r+i, s+j) \cdot R(i, j)) - K \cdot \bar{I}_{r,s} \cdot \bar{R}}{\left[\sum_{(i,j) \in R} I^2(r+i, s+j) - K \cdot \bar{I}_{r,s}^2 \right]^{1/2} \cdot S_R}, \quad (23.15)$$

and thereby obtain an efficient way to compute the local correlation coefficient. Since \bar{R} and $S_R = (S_R^2)^{1/2}$ must be calculated only once and the local average of the current subimage $\bar{I}_{r,s}$ is not immediately required for summing up the differences, the whole expression in Eqn. (23.15) can be computed in one common iteration, as shown in Alg. 23.1.

Note that in the calculation of $C_L(r, s)$ in Eqn. (23.15), the denominator becomes zero if any of the two factors is zero. This may happen, for example, if the search image I is locally “flat” and thus has zero variance or if the reference image R is constant. The quantity 1 is added to the denominator in Alg. 23.1 (line 23) to avoid divisions by zero in such cases, which otherwise has no significant effect on the result.

A direct Java implementation of this procedure is shown in Progs. 23.1 and 23.2 in Sec. 23.1.3 (class `CorrCoeffMatcher`).

Examples and discussion

Figure 23.3 compares the performance of the described distance functions in a typical example. The original image (Fig. 23.3(a)) shows a repetitive flower pattern produced under uneven lighting and differences in local brightness. One instance of the repetitive pattern was extracted as the reference image (Fig. 23.3(b)).

- The *sum of absolute differences* (Eqn. (23.2)) in Fig. 23.3(c) shows a distinct peak value at the original template position, as does the *Euclidean distance* (Eqn. (23.4)) in Fig. 23.3(e). Both measures work satisfactorily in this regard but are strongly affected by global intensity changes, as demonstrated in Figs. 23.4 and 23.5.
- The *maximum difference* (Eqn. (23.3)) in Fig. 23.3(d) proves completely useless as a distance measure since it responds more strongly to the lighting changes than to pattern similarity. As expected, the behavior of the *global cross correlation* in Fig. 23.3(f) is also unsatisfactory. Although the result exhibits a *local* maximum at the true template position (hardly visible in the printed image), it is completely dominated by the high-intensity responses in the brighter parts of the image.
- The result from the *normalized cross correlation* in Fig. 23.3(g) appears naturally very similar to the Euclidean distance (Fig. 23.3(e)), because in principle it is the same measure. As expected, the *correlation coefficient* (Eqn. (23.10)) in Fig. 23.3(h) yields the best results. Distinct peaks of similar intensity are produced for all six instances of the template pattern, and the result is unaffected by changing lighting conditions. In this case, the

1: **CorrelationCoefficient** (I, R)

Input: $I(u, v)$, search image; $R(i, j)$, reference image.
Returns a map $C(r, s)$ containing the values of the correlation coefficient between I and R positioned at (r, s) .

STEP 1—INITIALIZE:

- 2: $(M_I, N_I) \leftarrow \text{Size}(I)$
- 3: $(M_R, N_R) \leftarrow \text{Size}(R)$
- 4: $K \leftarrow M_R \cdot N_R$
- 5: $\Sigma_R \leftarrow 0, \Sigma_{R2} \leftarrow 0$
- 6: **for** $i \leftarrow 0, \dots, (M_R - 1)$ **do**
- 7: **for** $j \leftarrow 0, \dots, (N_R - 1)$ **do**
- 8: $\Sigma_R \leftarrow \Sigma_R + R(i, j)$
- 9: $\Sigma_{R2} \leftarrow \Sigma_{R2} + R^2(i, j)$
- 10: $\bar{R} \leftarrow \Sigma_R / K$ ▷ Eq. 23.11
- 11: $S_R \leftarrow (\Sigma_{R2} - K \cdot \bar{R}^2)^{1/2}$ ▷ Eq. 23.14

STEP 2—COMPUTE THE CORRELATION MAP:

- 12: Create map $C: (M_I - M_R + 1) \times (N_I - N_R + 1) \mapsto \mathbb{R}$
- 13: **for** $r \leftarrow 0, \dots, M_I - M_R$ **do** ▷ place R at position (r, s)
- 14: **for** $s \leftarrow 0, \dots, N_I - N_R$ **do**
 Compute the correlation coefficient for position (r, s) :
- 15: $\Sigma_I \leftarrow 0, \Sigma_{I2} \leftarrow 0, \Sigma_{IR} \leftarrow 0$
- 16: **for** $i \leftarrow 0, \dots, M_R - 1$ **do**
- 17: **for** $j \leftarrow 0, \dots, N_R - 1$ **do**
- 18: $a_I \leftarrow I(r + i, s + j)$
- 19: $a_R \leftarrow R(i, j)$
- 20: $\Sigma_I \leftarrow \Sigma_I + a_I$
- 21: $\Sigma_{I2} \leftarrow \Sigma_{I2} + a_I^2$
- 22: $\Sigma_{IR} \leftarrow \Sigma_{IR} + a_I \cdot a_R$
- 23: $C(r, s) \leftarrow \frac{\Sigma_{IR} - \Sigma_I \cdot \bar{R}}{1 + \sqrt{\Sigma_{I2} - \Sigma_I^2 / K} \cdot S_R}$
- 24: **return** C ▷ $C(r, s) \in [-1, 1]$

23.1 TEMPLATE MATCHING IN INTENSITY IMAGES

Alg. 23.1

Calculation of the correlation coefficient. Given is the search image I and the reference image (template) R . In Step 1, the template's average \bar{R} and variance term S_R are computed once. In Step 2, the match function is computed for every template position (r, s) as prescribed by Eqn. (23.15). The result is a map of correlation values $C(r, s) \in [-1, 1]$ that is returned. In line 23 (cf. Eqn. (23.15)) the quantity 1 is added to the denominator to avoid division by zero in the case of zero variance.

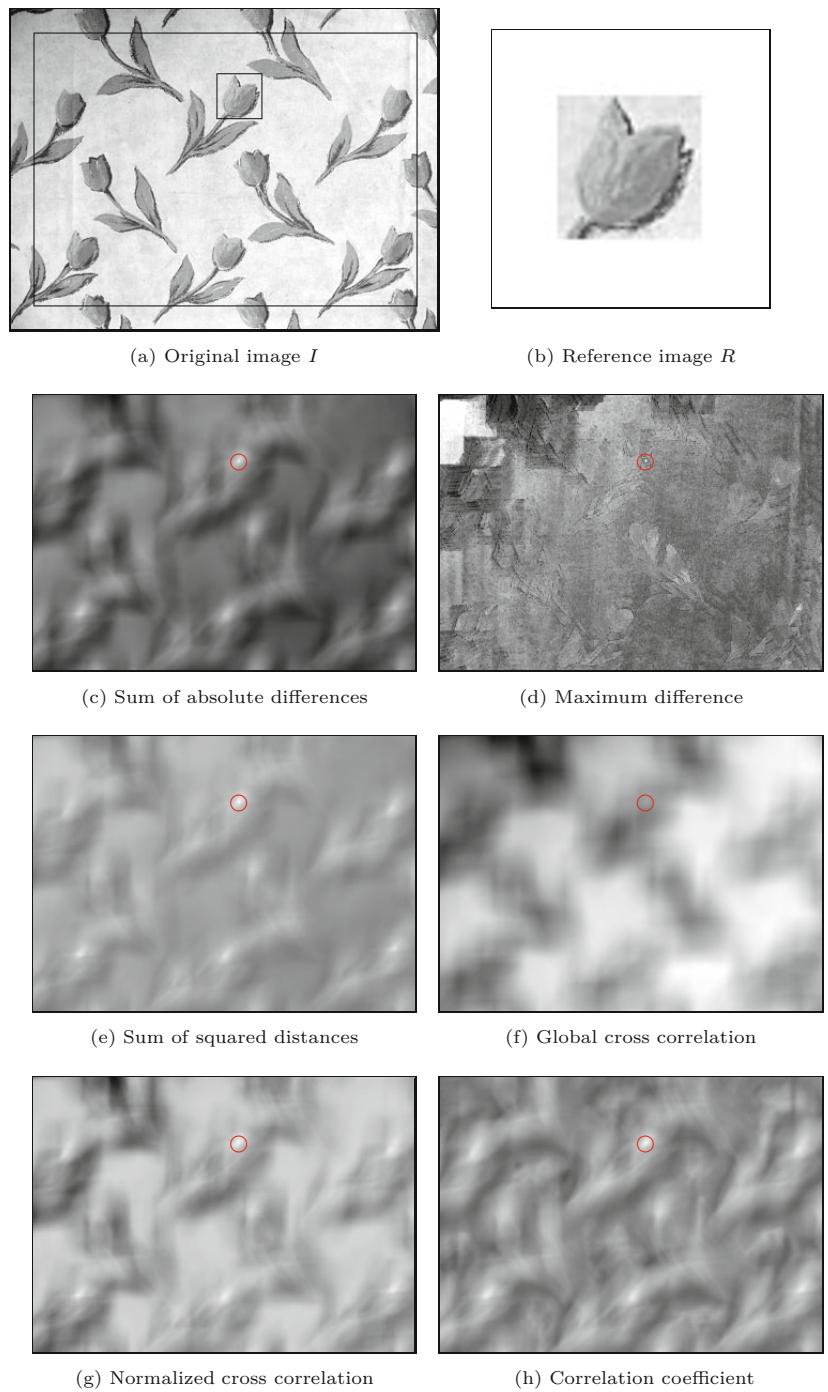
values range from -1.0 (black) to $+1.0$ (white), and zero values are shown as gray.

Figure 23.4 compares the results of the *Euclidean distance* against the *correlation coefficient* under globally changing intensity. For this purpose, the intensity of the reference image R is raised by 50 units such that the template is different from any subpattern in the original image. As can be seen clearly, the initially distinct peaks disappear under the Euclidean distance (Fig. 23.4(c)), while the correlation coefficient (Fig. 23.4(d)) naturally remains unaffected by this change.

In summary, the correlation coefficient can be recommended as a reliable measure for template matching in intensity images under realistic lighting conditions. This method proves relatively robust against global changes of brightness or contrast and tolerates small deviations from the reference pattern. Since the resulting values are in the fixed range of $[-1, 1]$, a simple threshold operation can be used to localize the best match points (Fig. 23.6).

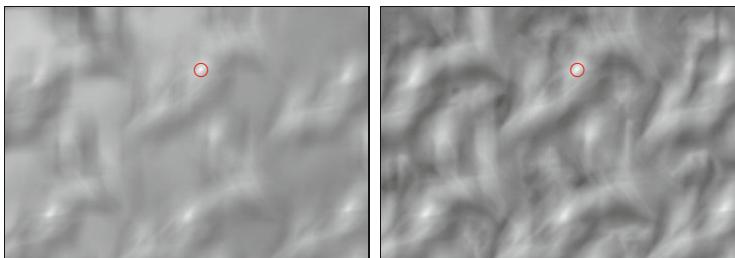
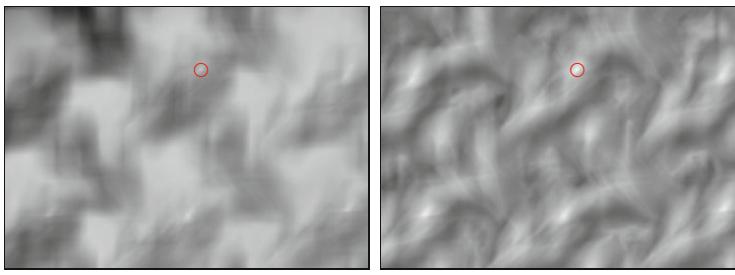
Fig. 23.3

Comparison of various distance functions. From the original image I (a), the marked section is used as the reference image R , shown enlarged in (b). In the resulting difference images (c-h), brightness corresponds to the amount of agreement (white equals minimum distance). The position of the true reference point is marked by a red circle.



Shape of the template

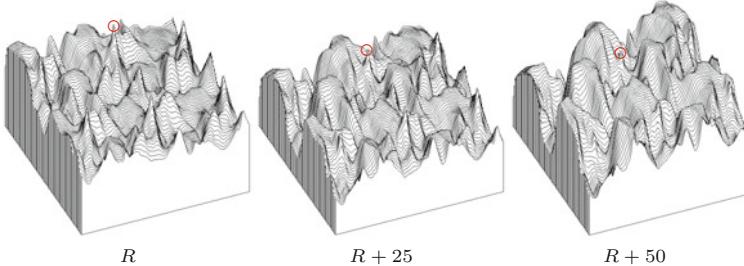
The shape of the reference image does not need to be rectangular as in the previous examples, although it is convenient for the processing. In some applications, circular, elliptical, or custom-shaped templates may be more applicable than a rectangle. In such a case, the template

Original reference image R (a) Euclidean distance $d_E(r, s)$ (b) Correlation coefficient $C_L(r, s)$ Modified reference image $R' = R + 50$ (c) Euclidean distance $d_E(r, s)$ (d) Correlation coefficient $C_L(r, s)$

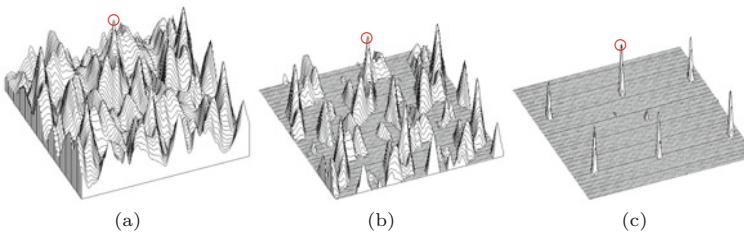
23.1 TEMPLATE MATCHING IN INTENSITY IMAGES

Fig. 23.4

Effects of changing global brightness. Original reference image R : the results from both the Euclidean distance (a) and the correlation coefficient (b) show distinct peaks at the positions of maximum agreement. Modified reference image $R' = R + 50$: the peak values disappear in the Euclidean distance (c), while the correlation coefficient (d) remains unaffected.

**Fig. 23.5**

Euclidean distance under global intensity changes. Distance function for the original template R (left), with the template intensity increased by 25 units (center) and 50 units (right). Notice that the local peaks disappear as the template intensity (and thus the total distance between the image and the template) is increased.

**Fig. 23.6**

Detection of match points by simple thresholding: correlation coefficient (a), positive values only (b), and values greater than 0.5 (c). The remaining peaks indicate the positions of the six similar (but not identical) tulip patterns in the original image (Fig. 23.3(a)).

may still be stored in a rectangular array, but the relevant pixels must somehow be marked (e.g., using a binary mask).

Even more general is the option to assign individual continuous weights to the template elements such that, for example, the center of a template can be given higher significance in the match than the peripheral regions. Implementing such a “windowed matching” technique should be straightforward and require only minor modifications to the standard approach.

23.1.2 Matching Under Rotation and Scaling

Correlation-based matching methods applied in the way described in this section cannot handle significant rotation or scale differences between the search image and the template. One obvious way to overcome rotation is to match using multiple rotated versions of the template, of course at the price of additional computation time. Similarly, one could try to match using several scaled versions of the template to achieve scale independence to some extent. Although this could be combined by using a set of rotated *and* scaled template patterns, the combinatorially growing number of required matching steps could soon become prohibitive for a practical implementation.

An interesting technique is matching in *logarithmic-polar* space, where rotation and scaling map to translations and can thus be handled with correlation-type methods [267]. However, this requires an initial “anchor point”, which again needs to be detected in a rotation and scale invariant way [152, 209, 238]. Another alternative is the popular Lucas-Kanade technique for elastic local matching, which is described at detail in Chapter 24. In principle, given an approximate starting solution, this method cannot only handle rotation and scaling, but arbitrary image transformations or distortions.

23.1.3 Java Implementation

Implementations of most methods described in this chapter are openly available as part of the `imagingbook` library.³ As an example, the code listed in Progs. 23.1 and 23.2 demonstrates the use of the `CorrCoeffMatcher` class for template matching based on the local correlation coefficient (Eqn. (23.10)). The application assumes that the search image (`I`) and the reference image (`R`) are already available as objects of type `FloatProcessor`. They are used to create a new instance of class `CorrCoeffMatcher`, as shown in the following code segment:

```
FloatProcessor I = ...      // search image
FloatProcessor R = ...      // reference image
CorrCoeffMatcher matcher = new CorrCoeffMatcher(I);
float[][] C = matcher.getMatch(R);
```

The correlation coefficient is computed by the method `getMatch()` and returned as a 2D `float`-array (`C`).

23.2 Matching Binary Images

As became evident in the previous section, the comparison of intensity images based on correlation may not be an optimal solution but is sufficiently reliable and efficient under certain restrictions. If we compare binary images in the same way, by counting the number of identical pixels in the search image and the template, the total difference will only be small when most pixels are in exact agreement.

³ Package `imagingbook.pub.matching`.

```

1 package imagingbook.pub.matching;
2
3 import ij.process.FloatProcessor;
4
5 class CorrCoeffMatcher {
6
7     private final FloatProcessor I; // search image
8     private final int MI, NI;      // width/height of search image
9
10    private FloatProcessor R;      // reference image
11    private int MR, NR;           // width/height of reference image
12    private int K;
13    private double meanR;         // mean value of reference ( $\bar{R}$ )
14    private double varR;          // square root of reference variance
15                                ( $\sigma_R$ )
16
17    public CorrCoeffMatcher(FloatProcessor I) { // constructor
18        this.I = I;
19        this.MI = this.I.getWidth();
20        this.NI = this.I.getHeight();
21    }
22
23    public float[][] getMatch(FloatProcessor R) {
24        this.R = R;
25        this.MR = R.getWidth();
26        this.NR = R.getHeight();
27        this.K = MR * NR;
28
29        // calculate the mean ( $\bar{R}$ ) and variance term ( $S_R$ ) of the template:
30        double sumR = 0;           //  $\Sigma_R = \sum R(i,j)$ 
31        double sumR2 = 0;          //  $\Sigma_{R^2} = \sum R^2(i,j)$ 
32        for (int j = 0; j < NR; j++) {
33            for (int i = 0; i < MR; i++) {
34                float aR = R.getf(i,j);
35                sumR += aR;
36                sumR2 += aR * aR;
37            }
38        }
39        this.meanR = sumR / K;     //  $\bar{R} = [\sum R(i,j)]/K$ 
40        this.varR =               //  $S_R = [\sum R^2(i,j) - K \cdot \bar{R}^2]^{1/2}$ 
41        Math.sqrt(sumR2 - K * meanR * meanR);
42
43        float[][] C = new float[MI - MR + 1][NI - NR + 1];
44        for (int r = 0; r <= MI - MR; r++) {
45            for (int s = 0; s <= NI - NR; s++) {
46                float d = (float) getMatchValue(r, s);
47                C[r][s] = d;
48            }
49        }
50        return C;
51    }
52
53    // continued...

```

23.2 MATCHING BINARY IMAGES

Prog. 23.1

Implementation of class `CorrCoeffMatcher` (part 1/2). The constructor method (lines 16–20) calculates the mean $\bar{R} = \text{meanR}$ (Eqn. (23.11)) and the variance $S_R = \text{varR}$ (Eqn. (23.14)) of the reference image R . The method `getMatch(R)` (lines 22–51) determines the match values between the search image I and the reference image R for all positions (r, s) .

Prog. 23.2

Implementation of class `CorrCoeffMatcher` (part 2/2). The local match value $C(r, s)$ (see Eqn. (23.15)) at the individual position (r, s) is calculated by method `getMatchValue(r, s)` (lines 54–72).

```

54  private double getMatchValue(int r, int s) {
55      double sumI = 0;      //  $\Sigma_I = \sum I(r+i, s+j)$ 
56      double sumI2 = 0;     //  $\Sigma_{I2} = \sum (I(r+i, s+j))^2$ 
57      double sumIR = 0;     //  $\Sigma_{IR} = \sum I(r+i, s+j) \cdot R(i, j)$ 
58
59      for (int j = 0; j < NR; j++) {
60          for (int i = 0; i < MR; i++) {
61              float aI = I.getf(r + i, s + j);
62              float aR = R.getf(i, j);
63              sumI += aI;
64              sumI2 += aI * aI;
65              sumIR += aI * aR;
66          }
67      }
68
69      double meanI = sumI / K;    //  $\bar{I}_{r,s} = \Sigma_I / K$ 
70      return (sumIR - K * meanI * meanR) /
71             (1 + Math.sqrt(sumI2 - K * meanI * meanI) * varR);
72  }
73
74 } // end of class CorrCoeffMatcher

```

Since there is no continuous transition between pixel values, the distribution produced by a simple distance function will generally be ill-behaved (i.e., highly discontinuous with many local extrema; see Fig. 23.7).

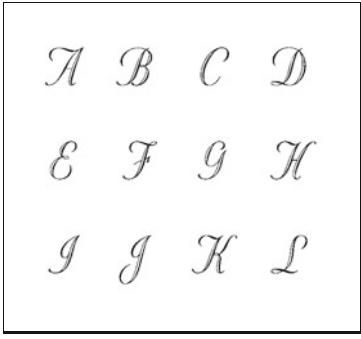
23.2.1 Direct Comparison of Binary Images

The problem with directly comparing binary images is that even the smallest deviations between image patterns, such as those caused by a small shift, rotation, or distortion, can create very high distance values. Shifting a thin line drawing by only a single pixel, for example, may be sufficient to switch from full agreement to no agreement at all (i.e., from zero difference to maximum difference). Thus a simple distance function gives no indication how far away and in which direction to search for a better match position.

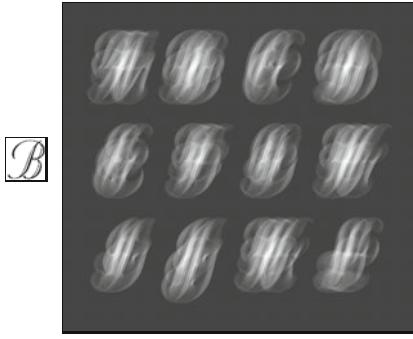
An interesting question is how matching of binary images can be made more tolerant against small differences of the compared patterns. Thus the goal is not only to detect the single image position, where most foreground pixels in the two images match up, but also (if possible) to obtain a measure indicating how far (in terms of geometry) we are away from this position.

23.2.2 The Distance Transform

A first step in this direction is to record the distance to the closest foreground pixel for every position (u, v) in the search image I . This gives us the minimum distance (though not the direction) for shifting a particular pixel onto a foreground pixel. Starting from a binary image $I(u, v) = I(\mathbf{u})$, we denote



(a)



(b)

23.2 MATCHING BINARY IMAGES

Fig. 23.7

Direct comparison of binary images. Given are a binary search image (a) and a binary reference image (b). The local similarity value for any template position corresponds to the relative number of matching (black) foreground pixels. High similarity values are shown as bright spots in the result (c). While the maximum similarity is naturally found at the correct position (at the center of the glyph B) the match function behaves wildly, with many local maxima.

$$FG(I) = \{\mathbf{u} \mid I(\mathbf{u}) = 1\}, \quad (23.16)$$

$$BG(I) = \{\mathbf{u} \mid I(\mathbf{u}) = 0\}, \quad (23.17)$$

as the set of coordinates of the foreground and background pixels, respectively. The so-called distance transform of I , $D(\mathbf{u}) \in \mathbb{R}$, is defined as

$$D(\mathbf{u}) := \min_{\mathbf{u}' \in FG(I)} \text{dist}(\mathbf{u}, \mathbf{u}'), \quad (23.18)$$

for all $\mathbf{u} = (u, v)$, where $u = 0, \dots, M-1$, $v = 0, \dots, N-1$ (for image size $M \times N$). The value D at a given position \mathbf{u} thus equals the distance between \mathbf{u} and the nearest foreground pixel in I . If $I(\mathbf{u})$ is a foreground pixel itself (i.e., $x \in FG$), then the distance $D(\mathbf{u}) = 0$ since no shift is necessary for moving this pixel onto a foreground pixel.

The function $\text{dist}(\mathbf{u}, \mathbf{u}')$ in Eqn. (23.18) measures the geometric distance between the two coordinate points $\mathbf{u} = (u, v)$ and $\mathbf{u}' = (u', v')$. Examples of suitable distance functions are the Euclidean distance (L_2 norm)

$$d_E(\mathbf{u}, \mathbf{u}') = \|\mathbf{u} - \mathbf{u}'\| = \sqrt{(u - u')^2 + (v - v')^2} \in \mathbb{R}^+ \quad (23.19)$$

and the *Manhattan distance*⁴ (L_1 norm)

$$d_M(\mathbf{u}, \mathbf{u}') = |u - u'| + |v - v'| \in \mathbb{N}_0. \quad (23.20)$$

Figure 23.8 shows a simple example of a distance transform using the Manhattan distance $d_M()$.

The direct calculation of the distance transform (following the definition in Eqn. (23.18)) is computationally expensive, because the closest foreground pixel must be found for each pixel position \mathbf{p} (unless $I(\mathbf{p})$ is a foreground pixel itself).⁵

Chamfer algorithm

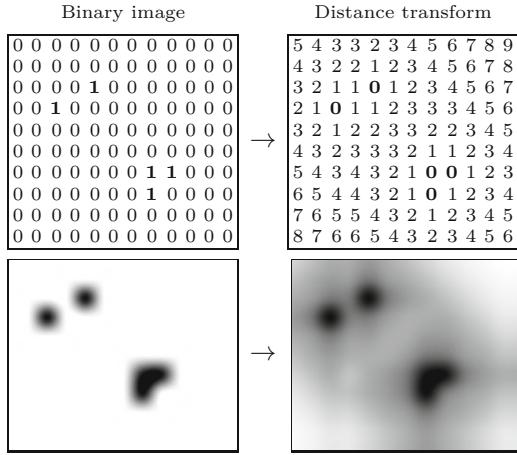
The so-called *chamfer* algorithm [30] is an efficient method for computing the distance transform. Similar to the sequential region labeling algorithm (see Ch. 10, Alg. 10.2), the chamfer algorithm traverses

⁴ Also called “city block distance”.

⁵ A simple (brute force) algorithm for the distance transform would perform a full scan over the entire image for each processed pixel, resulting in $\mathcal{O}(N^2 \cdot N^2) = \mathcal{O}(N^4)$ steps for an image of size $N \times N$.

Fig. 23.8

Example of a distance transform of a binary image using the Manhattan distance $d_M()$. Foreground pixels in the binary image have value 1 (shown inverted).



the image twice by propagating the computed values across the image like a wave. The first traversal starts at the upper left corner of the image and propagates the distance values downward in a diagonal direction. The second traversal proceeds in the opposite direction from the bottom to the top. For each traversal, a “distance mask” is used for the propagation of the distance values; that is,

$$M^L = \begin{bmatrix} m_2 & m_1 & m_2 \\ m_1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & m_1 \\ m_2 & m_1 & m_2 \end{bmatrix} \quad (23.21)$$

for the first and second traversals, respectively. The values in M^L and M^R describe the geometric distance between the current pixel (marked \times) and the relevant neighboring pixels. They depend upon the distance function $\text{dist}(\mathbf{x}, \mathbf{x}')$ used. Algorithm 23.2 outlines the chamfer method for computing the distance transform $D(u, v)$ for a binary image $I(u, v)$ using the above distance masks.

For the Manhattan distance, the chamfer algorithm computes the distance transform (Eqn. (23.20)) *exactly* using the masks

$$M_M^L = \begin{bmatrix} 2 & 1 & 2 \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M_M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ 2 & 1 & 2 \end{bmatrix}. \quad (23.22)$$

Similarly for the Euclidean distance (Eqn. (23.19)) can be calculated with the masks

$$M_E^L = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M_E^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix}. \quad (23.23)$$

Note that the result obtained with these masks is only an *approximation* of the Euclidean distance to the nearest foreground pixel, which is nevertheless more accurate than the estimate produced by the Manhattan distance. As demonstrated by the examples in Fig. 23.9, the distances obtained with the Euclidean masks are exact along the coordinate axes and the diagonals but are overestimated (i.e., too

```

1: DistanceTransform( $I, norm$ )
Input:  $I$ , a, binary image;  $norm \in \{L_1, L_2\}$ , distance function.
Returns the distance transform of  $I$ .
STEP 1: INITIALIZE
2:  $(m_1, m_2) \leftarrow \begin{cases} (1, 2) & \text{for } norm = L_1 \\ (1, \sqrt{2}) & \text{for } norm = L_2 \end{cases}$ 
3:  $(M, N) \leftarrow \text{Size}(I)$ 
4: Create map  $D: M \times N \mapsto \mathbb{R}$ 
5: for all  $(u, v) \in M \times N$  do
6:  $D(u, v) \leftarrow \begin{cases} 0 & \text{for } I(u, v) > 0 \\ \infty & \text{otherwise} \end{cases}$ 
STEP 2: L→R PASS
7: for  $v \leftarrow 0, \dots, N-1$  do ▷ top → bottom
8:   for  $u \leftarrow 0, \dots, M-1$  do ▷ left → right
9:     if  $D(u, v) > 0$  then
10:       $d_1, d_2, d_3, d_4 \leftarrow \infty$ 
11:      if  $u > 0$  then
12:         $d_1 \leftarrow m_1 + D(u-1, v)$ 
13:        if  $v > 0$  then
14:           $d_2 \leftarrow m_2 + D(u-1, v-1)$ 
15:        if  $v > 0$  then
16:           $d_3 \leftarrow m_1 + D(u, v-1)$ 
17:        if  $u < M-1$  then
18:           $d_4 \leftarrow m_2 + D(u+1, v-1)$ 
19:       $D(u, v) \leftarrow \min(D(u, v), d_1, d_2, d_3, d_4)$ 
STEP 3: R→L PASS
20: for  $v \leftarrow N-1, \dots, 0$  do ▷ bottom → top
21:   for  $u \leftarrow M-1, \dots, 0$  do ▷ right → left
22:     if  $D(u, v) > 0$  then
23:        $d_1, d_2, d_3, d_4 \leftarrow \infty$ 
24:       if  $u < M-1$  then
25:          $d_1 \leftarrow m_1 + D(u+1, v)$ 
26:         if  $v < N-1$  then
27:            $d_2 \leftarrow m_2 + D(u+1, v+1)$ 
28:         if  $v < N-1$  then
29:            $d_3 \leftarrow m_1 + D(u, v+1)$ 
30:         if  $u > 0$  then
31:            $d_4 \leftarrow m_2 + D(u-1, v+1)$ 
32:        $D(u, v) \leftarrow \min(D(u, v), d_1, d_2, d_3, d_4)$ 
33: return  $D$ 

```

23.2 MATCHING BINARY IMAGES

Alg. 23.2

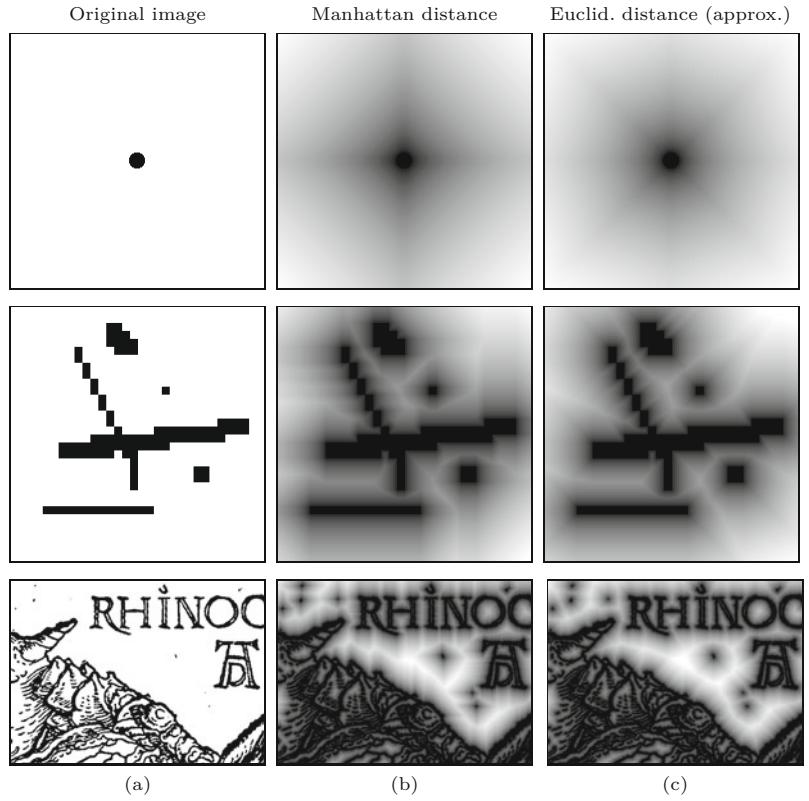
Chamfer algorithm for computing the distance transform. From the binary image I , the distance transform D (Eqn. (23.18)) is computed using a pair of distance masks (Eqn. (23.21)) for the first and second passes. Notice that the image borders require special treatment.

high) for all other directions. A more precise approximation can be obtained with distance masks of greater size (e.g., 5×5 pixels; see Exercise 23.3), which include the exact distances to pixels in a larger neighborhood [30]. Furthermore, floating point-operations can be avoided by using distance masks with scaled integer values, such as the masks

$$M_{E'}^L = \begin{bmatrix} 4 & 3 & 4 \\ 3 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M_{E'}^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 3 \\ 4 & 3 & 4 \end{bmatrix} \quad (23.24)$$

Fig. 23.9

Distance transform with the chamfer algorithm: original image with black foreground pixels (a), and results of distance transforms using the Manhattan distance (b) and the Euclidean distance (c). The brightness (scaled to maximum contrast) corresponds to the estimated distance to the nearest foreground pixel.



for the Euclidean distance. Compared with the original masks (Eqn. (23.23)), the resulting distance values are scaled by about the factor 3.

23.2.3 Chamfer Matching

The chamfer algorithm offers an efficient way to approximate the distance transform for a binary image of arbitrary size. The next step is to use the distance transform for matching binary images. *Chamfer matching* (first described in [19]) uses the distance transform to localize the points of maximum agreement between a binary search image I and a binary reference image (template) R . Instead of counting the overlapping foreground pixels as in the direct approach (see Sec. 23.2.1), chamfer matching uses the accumulated values of the distance transform as the match score Q . At each position (r, s) of the template R , the distance values corresponding to all foreground pixels in R are accumulated, that is,

$$Q(r, s) = \frac{1}{|FG(R)|} \cdot \sum_{\substack{(i,j) \in \\ FG(R)}} D(r + i, s + j), \quad (23.25)$$

where $K = |FG(R)|$ denotes the number of foreground pixels in the template R .

The complete procedure for computing the match score Q is summarized in Alg. 23.3. If at some position each foreground pixel in the

1: ChamferMatch (I, R)

Input: I , binary search image; R , binary reference image.

Returns a 2D map of match scores.

STEP 1 – INITIALIZE:

```

2:  $(M_I, N_I) \leftarrow \text{Size}(I)$ 
3:  $(M_R, N_R) \leftarrow \text{Size}(R)$ 
4:  $D \leftarrow \text{DistanceTransform}(I)$  ▷ Alg. 23.2
5: Create map  $Q: (M_I - M_R + 1) \times (N_I - N_R + 1) \mapsto \mathbb{R}$ 
```

STEP 2 – COMPUTE MATCH FUNCTION:

```

6: for  $r \leftarrow 0, \dots, M_I - M_R$  do ▷ place  $R$  at  $(r, s)$ 
7:   for  $s \leftarrow 0, \dots, N_I - N_R$  do
8:     Get match score for  $R$  placed at  $(r, s)$ 
9:      $q \leftarrow 0$ 
10:     $n \leftarrow 0$  ▷ number of foreground pixels in  $R$ 
11:    for  $i \leftarrow 0, \dots, M_R - 1$  do
12:      for  $j \leftarrow 0, \dots, N_R - 1$  do
13:        if  $R(i, j) > 0$  then ▷ foreground pixel in  $R$ 
14:           $q \leftarrow q + D(r + i, s + j)$ 
15:           $n \leftarrow n + 1$ 
16:     $Q(r, s) \leftarrow q/n$ 
```

16: **return** Q

23.2 MATCHING BINARY IMAGES
Alg. 23.3

Chamfer matching (calculation of the match function). Given is a binary search image I and a binary reference image (template) R . In step 1, the distance transform D is computed for the image I using the chamfer algorithm (Alg. 23.2). In step 2, the sum of distance values is accumulated for all foreground pixels in template R for each template position (r, s) . The resulting scores are stored in the 2D match map Q , which is returned.

template R coincides with a foreground pixel in the image I , the sum of the distance values is zero, which indicates a perfect match. The more foreground pixels of the template fall onto distance values greater than zero, the larger is the resulting score value Q (sum of distances). The best match is found at the global minimum of Q , that is,

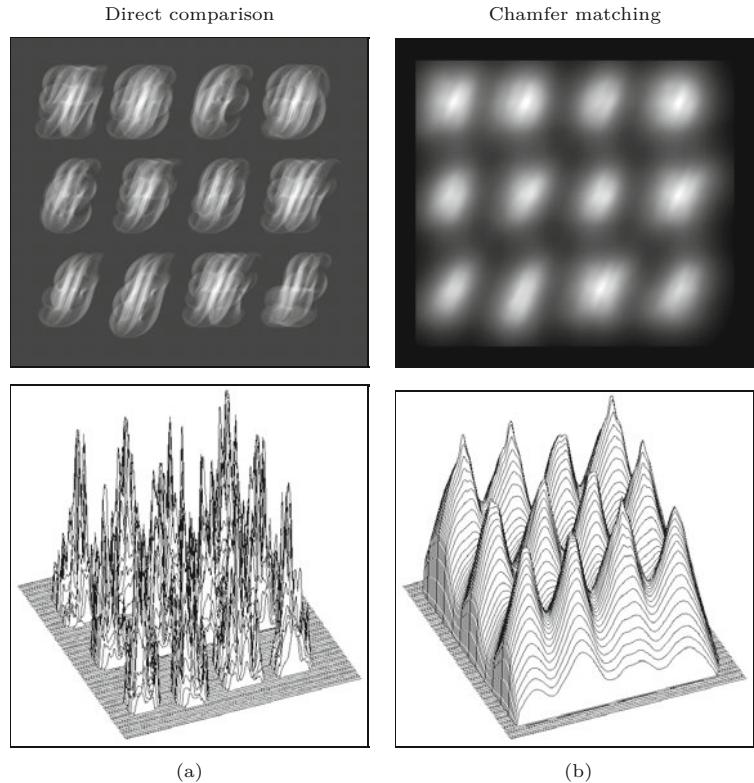
$$\mathbf{x}_{\text{opt}} = (r_{\text{opt}}, s_{\text{opt}}) = \underset{(r, s)}{\operatorname{argmin}}(Q(r, s)). \quad (23.26)$$

The example in Fig. 23.10 demonstrates the difference between direct pixel comparison and chamfer matching using the binary image shown in Fig. 23.7. Obviously the match score produced by the chamfer method is considerably smoother and exhibits only a few distinct local maxima. This is of great advantage because it facilitates the detection of optimal match points using simple local search methods. Figure 23.11 shows another example with circles and squares. The circles have different diameters and the medium-sized circle is used as the template. As this example illustrates, chamfer matching is tolerant against small-scale changes between the search image and the template and even in this case yields a smooth score function with distinct peaks.

While chamfer matching is not a “silver bullet”, it is efficient and works sufficiently well if the applications and conditions are suitable. It is most suited for matching line or edge images where the percentage of foreground pixels is small, such as for registering aerial images or aligning wide-baseline stereo images. The method tolerates deviations between the image and the template to a small extent but is of course not generally invariant under scaling, rotation, and deformation. The quality of the results deteriorates quickly when images contain random noise (“clutter”) or large foreground regions, because

Fig. 23.10

Direct pixel comparison vs. chamfer matching (see original images in Fig. 23.7). Unlike the results of the direct pixel comparison (a), the chamfer match score Q (b) is much smoother. It shows distinct peak values in places of high agreement that are easy to track down with local search methods. The match score Q (Eqn. (23.25)) in (b) is shown inverted for easy comparison.



the method is based on minimizing the distances to foreground pixels. One way to reduce the probability of false matches is not to use a *linear* summation (as in Eqn. (23.25)) but add up the *squared* distances, that is,

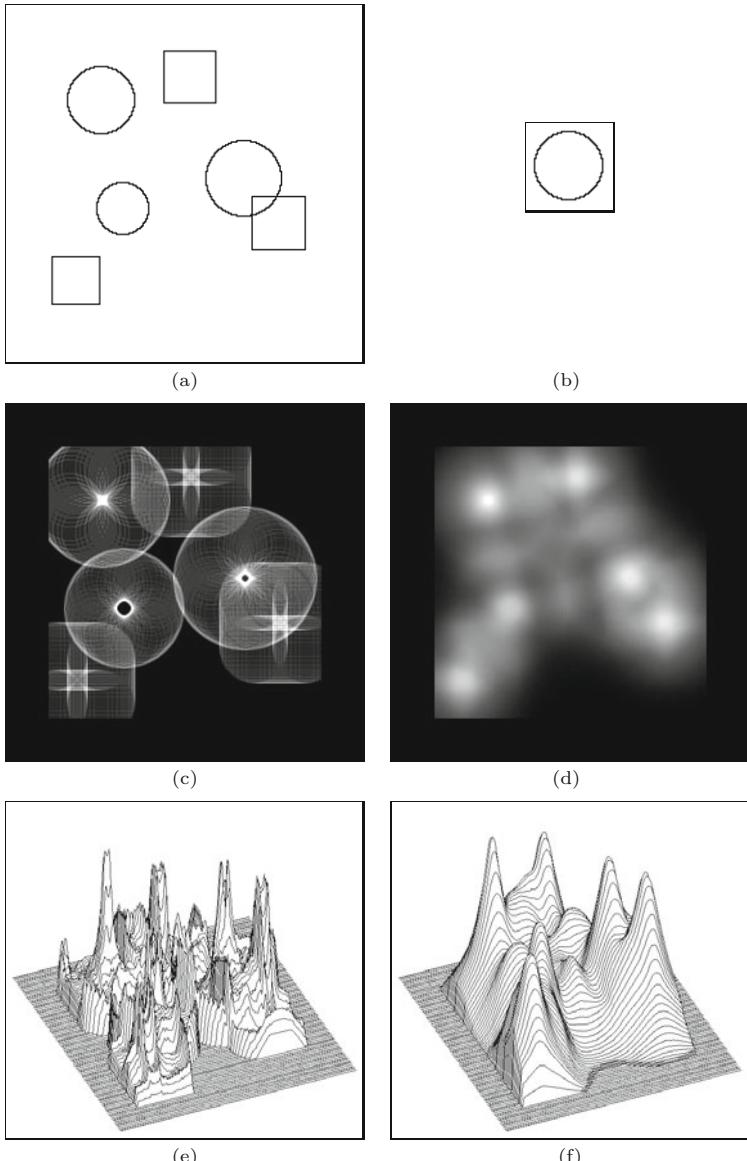
$$Q_{rms}(r, s) = \left[\frac{1}{K} \cdot \sum_{\substack{(i,j) \in \\ FG(R)}} (D(r + i, s + i))^2 \right]^{1/2} \quad (23.27)$$

("root mean square" of the distances) as the match score between the template R and the current subimage, as suggested in [30]. Also, hierarchical variants of the chamfer method have been proposed to reduce the search effort as well as to increase robustness [31].

23.2.4 Java Implementation

The calculation of the distance transform, as described in Alg. 23.2, is implemented by the class `DistanceTransform`.⁶ Program 23.3 shows the complete code for the class `ChamferMatcher` for comparing binary images with the distance transform, which is a direct implementation of Alg. 23.3. Additional examples (ImageJ plugins) can be found in the on-line code repository.

⁶ Package `imagingbook.pub.matching`.



23.3 EXERCISES

Fig. 23.11

Chamfer matching under varying scales. Binary search image with three circles of different diameters and three identical squares (a). The medium-sized circle at the top is used as the template (b). The result from a direct pixel comparison (c, e) and the result from chamfer matching (d, f). Again the chamfer match produces a much smoother score, which is most notable in the 3D plots shown in the bottom row (e, f). Notice that the three circles and the squares produce high match scores with similar absolute values (f).

23.3 Exercises

Exercise 23.1. Implement the chamfer-matching method (Alg. 23.2) for binary images using the Euclidean distance and the Manhattan distance.

Exercise 23.2. Implement the *exact* Euclidean distance transform using a “brute-force” search for each closest foreground pixel (this may take a while to compute). Compare your results with the approximation obtained with the chamfer method (Alg. 23.2), and compute the maximum deviation (as percentage of the real distance).

Exercise 23.3. Modify the chamfer algorithm for computing the distance transform (Alg. 23.2) by replacing the 3×3 pixel Euclidean distance masks (Eqn. (23.23)) with the following masks of size 5×5 :

$$M^L = \begin{bmatrix} \cdot & 2.236 & \cdot & 2.236 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 1.000 & \textcolor{red}{\times} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}, \quad (23.28)$$

$$M^R = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \textcolor{red}{\times} & 1.000 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 2.236 & \cdot & 2.236 & \cdot \end{bmatrix}. \quad (23.29)$$

Compare the results with those obtained with the standard masks. Why are no additional mask elements required along the coordinate axes and the diagonals?

Exercise 23.4. Implement the chamfer-matching technique using (a) the linear summation of distances (Eqn. (23.25)) and (b) the summation of squared distances (Eqn. (23.27)) for computing the match score. Select suitable test images to find out if version (b) is really more robust in terms of reducing the number of false matches.

Exercise 23.5. Adapt the template-matching method described in Sec. 23.1 for the comparison of RGB color images.

```

1 package imagingbook.pub.matching;
2 import ij.process.ByteProcessor;
3 import imagingbook.pub.matching.DistanceTransform.Norm;
4
5 public class ChamferMatcher {
6     private final ByteProcessor I;
7     private final int MI, NI;
8     private final float[][] D;           // distance transform of I
9
10    public ChamferMatcher(ByteProcessor I) {
11        this(I, Norm.L2);
12    }
13
14    public ChamferMatcher(ByteProcessor I, Norm norm) {
15        this.I = I;
16        this.MI = this.I.getWidth();
17        this.NI = this.I.getHeight();
18        this.D = (new DistanceTransform(I, norm)).
19            getDistanceMap();
20
21    public float[][] getMatch(ByteProcessor R) {
22        final int MR = R.getWidth();
23        final int NR = R.getHeight();
24        final int[][] Ra = R.getIntArray();
25        float[][] Q = new float[MI - MR + 1][NI - NR + 1];
26        for (int r = 0; r <= MI - MR; r++) {
27            for (int s = 0; s <= NI - NR; s++) {
28                float q = getMatchValue(Ra, r, s);
29                Q[r][s] = q;
30            }
31        }
32        return Q;
33    }
34
35    private float getMatchValue(int[][] R, int r, int s) {
36        float q = 0.0f;
37        for (int i = 0; i < R.length; i++) {
38            for (int j = 0; j < R[i].length; j++) {
39                if (R[i][j] > 0) { // foreground pixel in reference image
40                    q = q + D[r + i][s + j];
41                }
42            }
43        }
44        return q;
45    }
46 }

```

23.3 EXERCISES

Prog. 23.3

Java implementation of Alg. 23.3 (class `ChamferMatcher`). The distance transform of the binary search image I is calculated in the constructor method by an instance of class `DistanceTransform` and stored as a 2D `float` array (line 18). The method `getMatch(R)` in lines 21–45 computes the 2D match function Q (again as a `float` array) for the reference image R .

Non-Rigid Image Matching

The correlation-based registration methods described in Chapter 23 are *rigid* in the sense that they provide for *translation* as the only form of geometric transformation and positioning is limited to whole pixel units. In this chapter we look at methods that are capable of registering a reference image under (almost) arbitrary geometric transformations, such as changes in rotation, scale, and affine distortion, and also to *sub-pixel* accuracy.

At the core of this chapter is a detailed description of the classic Lucas-Kanade algorithm [154] and its efficient implementation. Unlike the methods presented earlier, the algorithms described here typically do not perform a global search over the entire image to find the best match, but start from an initial estimate of the geometric transformation to home in on the optimum position and distortion in an iterative fashion. This is not difficult, for example, in tracking applications, where the approximate location of a particular image patch can be predicted from the observed motion in previous frames. Of course, the global matching methods described in Chapter 23 can be used to find a coarse starting solution.

24.1 The Lucas-Kanade Technique

The basic idea of the Lucas-Kanade technique is best illustrated in the 1D case (see Fig. 24.1(a)).

24.1.1 Registration in 1D

Given two 1D, real-valued functions $f(x)$, $g(x)$, the registration problem is to find the disparity t in the (horizontal) x -direction under the assumption that g is a shifted version of f , that is,

$$g(x) = f(x - t). \quad (24.1)$$

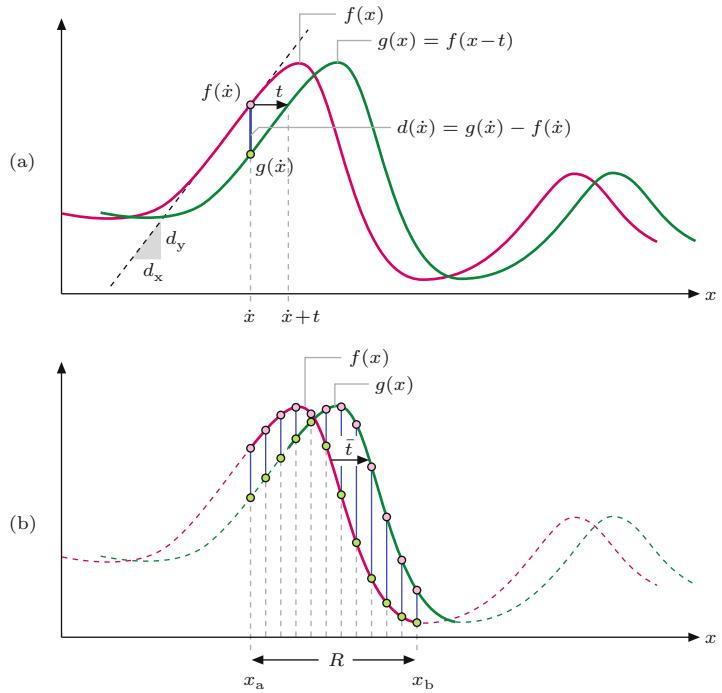
If the function f is linear in a (sufficiently large) neighborhood of some point x with slope $f'(x)$, then

Fig. 24.1

Registering two 1D functions (figure adapted from [154]).

The 1D function $g(x)$ is assumed to be a shifted version of $f(x)$. In (a), f is approximately linear at position \dot{x} , with slope $f'(\dot{x}) = d_y/d_x$.

Under this condition, the horizontal displacement t can be estimated from the difference of the local function values $f(\dot{x})$ and $g(\dot{x})$ as $t \approx (f(\dot{x}) - g(\dot{x}))/f'(\dot{x})$. In (b), the overall displacement \bar{t} is calculated by averaging the individual displacement estimates from multiple samples in the region $R = [x_a, x_b]$.



$$f(x - t) \approx f(x) - t \cdot f'(x) \quad (24.2)$$

and therefore

$$g(x) \approx f(x) - t \cdot f'(x). \quad (24.3)$$

Thus, given the function values $f(x)$, $g(x)$ and the first derivative $f'(x)$ at some point x , the displacement t can be estimated (from Eqn. (24.2)) as

$$t \approx \frac{f(x) - g(x)}{f'(x)}. \quad (24.4)$$

Note that this can be viewed as a first-order Taylor expansion¹ of the function f . Obviously, the estimate of the shift t in Eqn. (24.4) depends only on a single pair of function samples at position x and fails at points where f is either not linear or flat, that is, where the first derivative f' vanishes. To obtain a more robust displacement estimate it appears natural to extend the calculation over a range R of sample values, thereby aligning a complete section of the two functions f and g (see Fig. 24.1(b)). This problem can be formulated as finding the displacement t that minimizes the L_2 distance between the two functions f and g over a range R , that is, finding t such that

$$\mathcal{E}(t) = \sum_{x \in R} [f(x-t) - g(x)]^2 = \sum_{x \in R} [f(x) - t \cdot f'(x) - g(x)]^2 \quad (24.5)$$

¹ See also Sec. C.3.2 in the Appendix.

is a minimum. This can be accomplished by calculating the first derivative of the aforementioned expression (with respect to t) and setting it equal to zero, which gives

$$\frac{\partial \mathcal{E}}{\partial t} = 2 \cdot \sum_{x \in R} f'(x) \cdot [f(x) - f'(x) \cdot t - g(x)] = 0. \quad (24.6)$$

By solving this equation the optimal shift is found as

$$t_{\text{opt}} = \left[\sum_{x \in R} [f'(x)]^2 \right]^{-1} \cdot \sum_{x \in R} f'(x) \cdot [f(x) - g(x)]. \quad (24.7)$$

Note that this local estimation works even if the function f is flat at some positions in R , unless $f'(x)$ is zero everywhere R . However, since the estimate is based only on linear (i.e., first-order) prediction, the estimate is generally not accurate. For this purpose, the following iterative optimization scheme is proposed in [154], which is really the basis of the Lucas-Kanade algorithm. With $t^{(0)} = t_{\text{start}}$ as the initial estimate of the displacement (which may be zero), t is successively updated as

$$t^{(k)} = t^{(k-1)} + \left[\sum_{x \in R} [f'(x)]^2 \right]^{-1} \cdot \sum_{x \in R} f'(x) \cdot [f(x) - g(x)], \quad (24.8)$$

for $k = 1, 2, \dots$, until either $t^{(k)}$ converges or a maximum number of steps is reached.

24.1.2 Extension to Multi-Dimensional Functions

As shown in [154], the formulation given in Sec. 24.1.1 can be easily generalized to align multi-dimensional, scalar-valued functions, including 2D images. In general, the involved functions $F(\mathbf{x})$ and $G(\mathbf{x})$ are now defined over \mathbb{R}^m , and thus all coordinates $\mathbf{x} = (x_1, \dots, x_m)$ and spatial shifts $\mathbf{t} = (t_1, \dots, t_m)$ are m -dimensional column vectors. The task is, analogous to Eqn. (24.5), to find the vector \mathbf{t} that minimizes the error quantity

$$\mathcal{E}(\mathbf{t}) = \sum_{\mathbf{x} \in R} [F(\mathbf{x} - \mathbf{t}) - G(\mathbf{x})]^2, \quad (24.9)$$

where R denotes an m -dimensional region. The linear approximation in Eqn. (24.2) becomes

$$F(\mathbf{x} - \mathbf{t}) \approx F(\mathbf{x}) - \nabla_F(\mathbf{x}) \cdot \mathbf{t}, \quad (24.10)$$

where the row vector $\nabla_F(\mathbf{x}) = \left(\frac{\partial F}{\partial x_1}(\mathbf{x}), \dots, \frac{\partial F}{\partial x_m}(\mathbf{x}) \right)$ is the m -dimensional *gradient* of the function F , evaluated at position \mathbf{x} . Minimizing $\mathcal{E}(\mathbf{t})$ over \mathbf{t} is again accomplished by solving $\frac{\partial \mathcal{E}}{\partial \mathbf{t}} = 0$, that is (analogous to Eqn. (24.6)),

$$2 \cdot \sum_{\mathbf{x} \in R} \nabla_F(\mathbf{x}) \cdot [F(\mathbf{x}) - \nabla_F(\mathbf{x}) \cdot \mathbf{t} - G(\mathbf{x})] = 0. \quad (24.11)$$

The solution to Eqn. (24.11) is

**24 NON-RIGID IMAGE
MATCHING**

$$\mathbf{t}_{\text{opt}} = \left[\sum_{\mathbf{x} \in R} \nabla_F^\top(\mathbf{x}) \cdot \nabla_F(\mathbf{x}) \right]^{-1} \cdot \left[\sum_{\mathbf{x} \in R} \nabla_F^\top(\mathbf{x}) \cdot [F(\mathbf{x}) - G(\mathbf{x})] \right] \quad (24.12)$$

$$= \mathbf{H}_F^{-1} \cdot \left[\sum_{\mathbf{x} \in R} \nabla_F^\top(\mathbf{x}) \cdot [F(\mathbf{x}) - G(\mathbf{x})] \right], \quad (24.13)$$

where \mathbf{H}_F is an estimate of the $m \times m$ Hessian matrix² for the function F over the region R . Note the similarity of Eqn. (24.13) to the 1D version in Eqn. (24.7).

24.2 The Lucas-Kanade Algorithm

Based on the ideas outlined in Sec. 24.1, the Lucas-Kanade algorithm [154] is not only capable of registering 2D images by finding the optimal translation, but works for a range of geometric transformations T_p that can be parameterized by a n -dimensional vector p . Among others, this includes affine and projective transformations (see Ch. 21) as the most important cases.

The same mathematical notation is used as in Chapter 23, that is, I denotes the *search image* and R is the (typically smaller) *reference image*. The placement and possible distortion of the matching image patch is described by a *geometric transformation* T_p (cf. Ch. 21), where p denotes a vector of transformation parameters. The goal of the Lucas-Kanade registration algorithm is to minimize the expression

$$\mathcal{E}(p) = \sum_{\mathbf{x} \in R} [I(T_p(\mathbf{x})) - R(\mathbf{x})]^2 \quad (24.14)$$

with respect to the geometric transformation parameters p , where I is the (search) image, R is the reference image (template), and $T_p(\mathbf{x})$ is a geometric transformation or warp function with parameters p . For example, simple 2D translation is described by the transformation

$$T_p(\mathbf{x}) = \mathbf{x} + p = \begin{pmatrix} x + t_x \\ y + t_y \end{pmatrix}, \quad (24.15)$$

where $\mathbf{x} = (x, y)^\top$ and $p = (t_x, t_y)^\top$. The task of the alignment process is to find the parameters that describe how to warp the search image I , such that the match between I and R is optimal over the support region R . [Figure 24.2](#) illustrates the corresponding geometry.

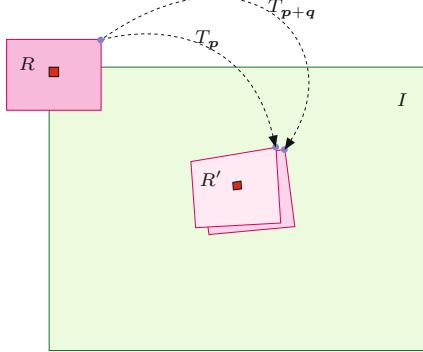
In each iteration, the Lucas-Kanade algorithm starts with an estimate of the transformation parameters p and attempts to find the parameter increment q that locally minimizes the expression

$$\mathcal{E}(q) = \sum_{\mathbf{x} \in R} [I(T_{p+q}(\mathbf{x})) - R(\mathbf{x})]^2. \quad (24.16)$$

After calculating the optimal parameter change q_{opt} , the parameter vector p is updated in the form

$$p \leftarrow p + q_{\text{opt}} \quad (24.17)$$

² See Sec. C.2.6 in the Appendix for details.


Fig. 24.2

Geometric relations in the (forward) Lucas-Kanade registration algorithm. I denotes the search image and R is the reference image. The mapping T_p warps the reference image R from the original position (centered at the origin) to R' , with \mathbf{p} being the initial parameter estimate. Matching is performed between the search image I and the warped reference image R' . T_{p+q} is the improved warp; the optimal parameter change \mathbf{q} is estimated in each iteration.

until the process converges. Typically, the update loop is terminated when the magnitude of the change vector \mathbf{q}_{opt} drops below a predefined threshold.

The expression to be minimized in Eqn. (24.16) depends on the image content and is generally nonlinear with respect to \mathbf{q} . A locally linear approximation of this function is obtained by the first-order Taylor expansion on I , that is,³

$$I(T_{p+q}(\mathbf{x})) \approx I(T_p(\mathbf{x})) + \underbrace{\nabla_I(T_p(\mathbf{x}))}_{1 \times 2} \cdot \underbrace{\mathbf{J}_{T_p}(\mathbf{x})}_{2 \times n} \cdot \underbrace{\mathbf{q}}_{n \times 1}, \quad (24.18)$$

$\in \mathbb{R}$

where the 2D (column) vector

$$\nabla_I(\mathbf{x}) = (I_x(\mathbf{x}), I_y(\mathbf{x})) \quad (24.19)$$

is the *gradient* of the image I at some position \mathbf{x} and $\mathbf{J}_{T_p}(\mathbf{x})$ denotes the *Jacobian* matrix⁴ of the warp function T_p , also evaluated at position \mathbf{x} . In general, the Jacobian of a 2D warp function

$$T_p(\mathbf{x}) = \begin{pmatrix} T_{x,p}(\mathbf{x}) \\ T_{y,p}(\mathbf{x}) \end{pmatrix} \quad (24.20)$$

with n parameters $\mathbf{p} = (p_0, p_1, \dots, p_{n-1})^\top$ is a $2 \times n$ matrix function

$$\mathbf{J}_{T_p}(\mathbf{x}) = \begin{pmatrix} \frac{\partial T_{x,p}}{\partial p_0}(\mathbf{x}) & \frac{\partial T_{x,p}}{\partial p_1}(\mathbf{x}) & \dots & \frac{\partial T_{x,p}}{\partial p_{n-1}}(\mathbf{x}) \\ \frac{\partial T_{y,p}}{\partial p_0}(\mathbf{x}) & \frac{\partial T_{y,p}}{\partial p_1}(\mathbf{x}) & \dots & \frac{\partial T_{y,p}}{\partial p_{n-1}}(\mathbf{x}) \end{pmatrix}. \quad (24.21)$$

With the linear approximation in Eqn. (24.18), the original minimization problem in Eqn. (24.14) can now be written as

³ In some of the following equations, we distinguish carefully between row and column vectors and the dimensions of vectors and matrices are explicitly displayed (in underbraces) to avoid possible confusion.

⁴ The Jacobian \mathbf{J} of a function f is a matrix containing the first partial derivatives of f , that is, it is a matrix of functions (see also Sec. C.2.1 in the Appendix).

$$\mathcal{E}(\mathbf{q}) \approx \sum_{\mathbf{u} \in R} [I(T_p(\mathbf{u})) + \nabla_I(T_p(\mathbf{u})) \cdot \mathbf{J}_{T_p}(\mathbf{u}) \cdot \mathbf{q} - R(\mathbf{u})]^2 \quad (24.22)$$

$$= \sum_{\mathbf{u} \in R} [I(\hat{\mathbf{u}}) + \nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u}) \cdot \mathbf{q} - R(\mathbf{u})]^2, \quad (24.23)$$

with $\hat{\mathbf{x}} = T_p(\mathbf{x})$. Finding the parameters \mathbf{q} that give the smallest difference $\mathcal{E}(\mathbf{q})$ is a linear least-squares minimization problem, which can be solved by taking the first partial derivative with respect to \mathbf{q} , that is,

$$\underbrace{\frac{\partial d}{\partial \mathbf{q}}}_{n \times 1} \approx \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u})]^\top}_{n \times 1} \cdot \underbrace{[I(\hat{\mathbf{u}}) + \nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u}) \cdot \mathbf{q} - R(\mathbf{u})]}_{\in \mathbb{R}}^2, \quad (24.24)$$

and setting it equal to zero.⁵ Solving the resulting equation for the unknown \mathbf{q} yields the parameter change minimizing Eqn. (24.24) as

$$\mathbf{q}_{\text{opt}} = \bar{\mathbf{H}}^{-1} \cdot \delta_p, \quad (24.25)$$

where $\bar{\mathbf{H}}$ is an estimate of the Hessian matrix (see Eqns. (24.29)–(24.30)),

$$\delta_p = \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u})]^\top}_{\mathbf{s}(\mathbf{u}) \in \mathbb{R}^n} \cdot \underbrace{[R(\mathbf{u}) - I(\hat{\mathbf{u}})]}_{D(\mathbf{u}) \in \mathbb{R}} = \sum_{\mathbf{u} \in R} \mathbf{s}^\top(\mathbf{u}) \cdot D(\mathbf{u}) \quad (24.26)$$

is a n -dimensional column vector, and

$$D(\mathbf{u}) = R(\mathbf{u}) - I(\hat{\mathbf{u}}) \quad (24.27)$$

is the resulting (scalar-valued) error image. $\mathbf{s}(\mathbf{u}) = (s_0(\mathbf{u}), \dots, s_{n-1}(\mathbf{u}))$ is a n -dimensional row vector, with each element corresponding to one of the parameters in p . The 2D *scalar* fields formed by the individual components of the vector field $\mathbf{s}(\mathbf{u})$,

$$s_0, \dots, s_{n-1}: M_R \times N_R \mapsto \mathbb{R}, \quad (24.28)$$

are called *steepest descent images* for the current transformation parameters p .⁶ These images are of the same size as the reference image R . Finally, the $n \times n$ matrix

$$\bar{\mathbf{H}} = \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u})]^\top}_{n \times 1} \cdot \underbrace{[\nabla_I(\hat{\mathbf{u}}) \cdot \mathbf{J}_{T_p}(\mathbf{u})]}_{1 \times n} \quad (24.29)$$

$$= \sum_{\mathbf{u} \in R} \mathbf{s}^\top(\mathbf{u}) \cdot \mathbf{s}(\mathbf{u}) \approx \begin{pmatrix} \frac{\partial^2 D}{\partial p_0^2}(p) & \cdots & \frac{\partial^2 D}{\partial p_0 \partial p_{n-1}}(p) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 D}{\partial p_{n-1} \partial p_0}(p) & \cdots & \frac{\partial^2 D}{\partial p_{n-1}^2}(p) \end{pmatrix} \quad (24.30)$$

⁵ Note that in Eqn. (24.24) the left factor inside the summation is a n -dimensional column vector, while the right factor is a scalar.

⁶ The value $s_k(\mathbf{u})$ indicates the optimal change of parameter p_k for the individual pixel position \mathbf{u} to achieve a steepest-descent optimization of Eqn. (24.23) (see [13, Sec. 4.3]).

in Eqn. (24.25) is an estimate of the Hessian matrix⁷ for the given transformation parameters \mathbf{p} , calculated over all coordinates \mathbf{x} of the reference image R (Eqn. (24.29)).

The inverse of this matrix is used to calculate the optimal parameter change \mathbf{q}_{opt} in Eqn. (24.25). A better alternative to this formulation is to solve

$$\bar{\mathbf{H}} \cdot \mathbf{q}_{\text{opt}} = \boldsymbol{\delta}_{\mathbf{p}}, \quad (24.31)$$

for \mathbf{q}_{opt} as the unknown, without explicitly calculating $\mathbf{H}_{\mathbf{p}}^{-1}$. This is a system of linear equations in the standard form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, which is numerically more stable and efficient to solve than Eqn. (24.25).⁸

24.2.1 Summary of the Algorithm

In order not to get lost after this (quite mathematical) presentation, let us recap the key steps of the Lucas-Kanade method in a more compact form. In summary, given a search image I , a reference image R , a geometric transformation $T_{\mathbf{p}}$, an initial parameter estimate \mathbf{p}_{init} , and the convergence limit ϵ , the Lucas-Kanade algorithm performs the following steps:

A. Initialize:

1. Calculate the gradient $\nabla_I(\mathbf{u})$ of the search image I for all image positions $\mathbf{u} \in I$.
2. Initialize the transformation parameters: $\mathbf{p} \leftarrow \mathbf{p}_{\text{init}}$.

B. Repeat:

3. Calculate the warped gradient image $\nabla'_I(\mathbf{u}) = \nabla_I(T_{\mathbf{p}}(\mathbf{u}))$, for each position $\mathbf{u} \in R$ (by interpolation of ∇_I).
4. Calculate the $(2 \times n)$ Jacobian matrix $\mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u}) = \frac{\partial T_{\mathbf{p}}}{\partial \mathbf{p}}(\mathbf{u})$ of the warp function $T_{\mathbf{p}}(\mathbf{x})$, for each position $\mathbf{u} \in R$ and the current parameter vector \mathbf{p} (see Eqn. (24.21)).
5. Compute the n -dim. row vectors $\mathbf{s}_{\mathbf{u}} = \nabla'_I(\mathbf{u}) \cdot \mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u})$, for each position $\mathbf{u} \in R$ (see Eqn. (24.26)).
6. Compute the cumulative $n \times n$ Hessian matrix as $\bar{\mathbf{H}} = \sum_{\mathbf{u} \in R} \mathbf{s}_{\mathbf{u}}^T \cdot \mathbf{s}_{\mathbf{u}}$ (see Eqn. (24.29)).
7. Calculate the error image $D(\mathbf{x}) = R(\mathbf{u}) - I(T_{\mathbf{p}}(\mathbf{u}))$, for each position $\mathbf{u} \in R$ (by interpolation of I , see Eqn. (24.26)).
8. Compute the column vector $\boldsymbol{\delta}_{\mathbf{p}} = \sum_{\mathbf{u} \in R} \mathbf{s}_{\mathbf{u}}^T \cdot D(\mathbf{u})$ (see Eqn. (24.26)).
9. Calculate the optimal parameter change $\mathbf{q}_{\text{opt}} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\mathbf{p}}$ (see Eqn. (24.25)).
10. Update the transformation parameter: $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{q}_{\text{opt}}$ (see Eqn. (24.17)).

Until $\|\mathbf{q}_{\text{opt}}\| < \epsilon$.

⁷ The Hessian matrix of a n -variable, real-valued function f is composed of f 's second-order partial derivatives (see also Sec. C.2.6 in the Appendix). The Hessian matrix \mathbf{H} is always symmetric.

⁸ Moreover, Eqn. (24.31) may be solvable even if the matrix $\bar{\mathbf{H}}$ is almost singular and thus numerically not invertible [160, p. 164].

24 NON-RIGID IMAGE MATCHING

Alg. 24.1

Lucas-Kanade (“forward-additive”) registration algorithm. The origin of the reference image R is placed at its center. The gradient of the image is calculated only once (line 6), but interpolated in every iteration (line 15). Also, the $n \times n$ Hessian matrix $\bar{\mathbf{H}}$ is calculated and inverted in every iteration. The Jacobian of the warp function T is also evaluated repeatedly (line 16), though this is not an expensive calculation, at least for affine warps (lines 32–33). Procedure $\text{Interpolate}(I, \mathbf{x}')$ returns the interpolated value of the image I at the continuous position $\mathbf{x}' \in \mathbb{R}^2$ (see Ch. 22 for details and possible implementations).

$\text{Interpolate}(I, \mathbf{x}')$ returns the interpolated value of the image I at the continuous position $\mathbf{x}' \in \mathbb{R}^2$ (see Ch. 22 for details and possible implementations).

```

1: LucasKanadeForward( $I, R, T, \mathbf{p}_{\text{init}}, \epsilon, i_{\text{max}}$ )
   Input:  $I$ , the search image;  $R$ , the reference image;  $T$ , a 2D warp
         function that maps any point  $\mathbf{x} \in \mathbb{R}^2$  to some point  $\mathbf{x}' = T_p(\mathbf{x})$ ,
         with transformation parameters  $\mathbf{p} = (p_0, \dots, p_{n-1})$ ;  $\mathbf{p}_{\text{init}}$ , initial
         estimate of the warp parameters;  $\epsilon$ , the error limit;  $i_{\text{max}}$ , the
         maximum number of iterations.
   Returns the modified warp parameter vector  $\mathbf{p}$  for the best fit
   between  $I$  and  $R$ , or nil if no match could be found.

2:  $(M_R, N_R) \leftarrow \text{Size}(R)$                                  $\triangleright$  size of the reference image  $R$ 
3:  $\mathbf{x}_c \leftarrow 0.5 \cdot (M_R - 1, N_R - 1)$                    $\triangleright$  center of  $R$ 
4:  $\mathbf{p} \leftarrow \mathbf{p}_{\text{init}}$                                      $\triangleright$  initial transformation parameters
5:  $n \leftarrow \text{Length}(\mathbf{p})$                                  $\triangleright$  parameter count
6:  $(I_x, I_y) \leftarrow \text{Gradient}(I)$                            $\triangleright$  calculate the gradient  $\nabla I$ 
7:  $i \leftarrow 0$                                                $\triangleright$  iteration counter

8: do                                                        $\triangleright$  main loop
9:    $i \leftarrow i + 1$ 
10:   $\bar{\mathbf{H}} \leftarrow \mathbf{0}_{n,n}$                                  $\triangleright \bar{\mathbf{H}} \in \mathbb{R}^{n \times n}$ , initialized to zero
11:   $\delta_p \leftarrow \mathbf{0}_n$                                      $\triangleright \delta_p \in \mathbb{R}^n$ , initialized to zero
12:  for all positions  $\mathbf{u} \in (M_R \times N_R)$  do
13:     $\mathbf{x} \leftarrow \mathbf{u} - \mathbf{x}_c$                                  $\triangleright$  position w.r.t. the center of  $R$ 
14:     $\mathbf{x}' \leftarrow T_p(\mathbf{x})$                                  $\triangleright$  warp  $\mathbf{x}$  to  $\mathbf{x}'$  by transf.  $T_p$ 
   Estimate the gradient of  $I$  at the warped position  $\mathbf{x}'$ :
15:     $\nabla \leftarrow (\text{Interpolate}(I_x, \mathbf{x}'), \text{Interpolate}(I_y, \mathbf{x}'))$      $\triangleright$  2D row
        vector
16:     $\mathbf{J} \leftarrow \text{Jacobian}(T_p, \mathbf{x})$                        $\triangleright$  Jacobian of  $T_p$  at pos.  $\mathbf{x}$ 
17:     $\mathbf{s} \leftarrow (\nabla \cdot \mathbf{J})^\top$                              $\triangleright \mathbf{s}$  is a column vector of length  $n$ 
18:     $\mathbf{H} \leftarrow \mathbf{s} \cdot \mathbf{s}^\top$                             $\triangleright$  outer product,  $\mathbf{H}$  is of size  $n \times n$ 
19:     $\bar{\mathbf{H}} \leftarrow \bar{\mathbf{H}} + \mathbf{H}$                           $\triangleright$  cumulate the Hessian (Eq. 24.30)
20:     $d \leftarrow R(\mathbf{u}) - \text{Interpolate}(I, \mathbf{x}')$            $\triangleright$  pixel difference  $d \in \mathbb{R}$ 
21:     $\delta_p \leftarrow \delta_p + \mathbf{s} \cdot d$ 
22:     $\mathbf{q}_{\text{opt}} \leftarrow \bar{\mathbf{H}}^{-1} \cdot \delta_p$   $\triangleright$  Eq. 24.17, or solve  $\bar{\mathbf{H}} \cdot \mathbf{q}_{\text{opt}} = \delta_p$  (Eq. 24.31)
23:     $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{q}_{\text{opt}}$ 

24:  while ( $\|\mathbf{q}_{\text{opt}}\| > \epsilon$ )  $\wedge$  ( $i < i_{\text{max}}$ )       $\triangleright$  repeat until convergence
25:  if  $i < i_{\text{max}}$  then
26:    return  $\mathbf{p}$ 
27:  else
28:    return nil

29: Gradient( $I$ )
   Returns the gradient of  $I$  as a pair of maps.
30:  $H_x = \frac{1}{8} \cdot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad H_y = \frac{1}{8} \cdot \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ 
31: return ( $I * H_x, I * H_y$ )

32: Jacobian( $T_p, \mathbf{x}$ )
   Returns the  $2 \times n$  Jacobian matrix of the 2D warp function
    $T_p(\mathbf{x}) = (T_{x,p}(\mathbf{x}), T_{y,p}(\mathbf{x}))$  with parameters  $\mathbf{p} = (p_0, \dots, p_{n-1})$ 
   for the spatial position  $\mathbf{x} \in \mathbb{R}^2$ .
33: return  $\begin{pmatrix} \frac{\partial T_{x,p}}{\partial p_0}(\mathbf{x}) & \frac{\partial T_{x,p}}{\partial p_1}(\mathbf{x}) & \dots & \frac{\partial T_{x,p}}{\partial p_{n-1}}(\mathbf{x}) \\ \frac{\partial T_{y,p}}{\partial p_0}(\mathbf{x}) & \frac{\partial T_{y,p}}{\partial p_1}(\mathbf{x}) & \dots & \frac{\partial T_{y,p}}{\partial p_{n-1}}(\mathbf{x}) \end{pmatrix}$      $\triangleright$  see Eq. 24.21

```

The complete specification of the Lucas-Kanade algorithm (referred to as the “forward-additive” algorithm in [13]) is given in Alg. 24.1. In addition to the two images I and R , the procedure requires the assumed type of the geometric transformation T , the estimated initial transformation parameters \mathbf{p}_{init} , a convergence limit ϵ and the maximum number of iterations i_{max} . The optimal parameter vector \mathbf{p} is returned or nil if the optimization did not converge. For better numerical stability, the origin of the reference image R is placed at its center \mathbf{x}_c (see line 3), as is also illustrated in Fig. 24.2. The algorithm shows (unlike the just given summary) that it is sufficient to calculate the Jacobian \mathbf{J} (see line 16) and the Hessian matrix $\tilde{\mathbf{H}}$ (see line 18) only for the current position (\mathbf{u}) in the reference image, which implies relatively modest storage requirements. Additional instructions for calculating the Jacobian and Hessian matrices for specific linear transformations T are described in Sec. 24.4. In the case that $\tilde{\mathbf{H}}$ cannot be inverted (because it is singular) in line 22, the algorithm could either stop (and return nil) or continue with a small random perturbation of the transformation parameters \mathbf{p} .

This so-called forward-additive algorithm performs reliably if the assumed type of geometric transformation is correct and the initial parameter estimate is sufficiently close to the actual parameters. However, it is computationally demanding since it requires repeated warping of the gradient image and the Jacobian \mathbf{J}_{T_p} as well as the Hessian matrix \mathbf{H} must be re-calculated in each iteration. Very similar results at greatly improved performance are obtained with the “inverse compositional algorithm” described in Sec. 24.3.

24.3 Inverse Compositional Algorithm

This algorithm, described in [14], exchanges the roles of the search image I and the reference image R . As illustrated in Fig. 24.3, the reference image R remains anchored at the original position, while the geometric transformations are applied to (parts of) the search image I . In particular, the transformation T_p now describes the mapping from the warped image I' back to the original image I . The advantage of this algorithm is that it avoids re-evaluating the Jacobian and Hessian matrices in every iteration while exhibiting convergence properties similar to the Lucas-Kanade (forward-additive) algorithm described in Sec. 24.2.

In this algorithm, the expression to be minimized in each iteration is (cf. Eqn. (24.16))

$$\mathcal{E}(\mathbf{q}) = \sum_{\mathbf{u} \in R} [R(T_q(\mathbf{u})) - I(T_p(\mathbf{u}))]^2, \quad (24.32)$$

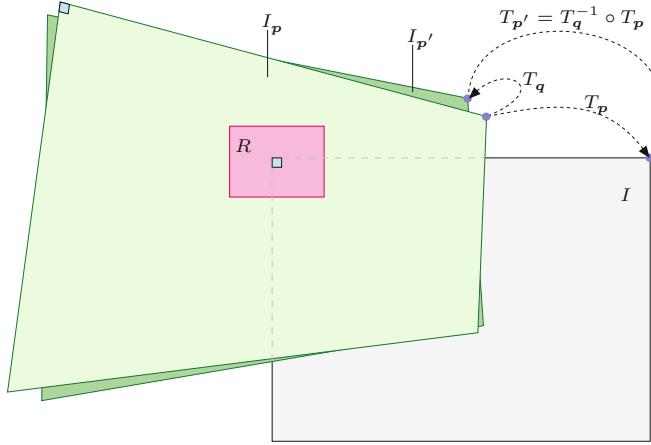
with respect to the parameter change \mathbf{q} , producing an optimal change vector \mathbf{q}_{opt} . Subsequently, the geometric transformation is updated not by simply *adding* \mathbf{q}_{opt} to the current parameter estimate \mathbf{p} (as in Eqn. (24.17)), but by *concatenating* the corresponding warps in the form

$$T_{p'}(\mathbf{x}) = (T_{q_{\text{opt}}}^{-1} \circ T_p)(\mathbf{x}) = T_p(T_{q_{\text{opt}}}^{-1}(\mathbf{x})) \quad (24.33)$$

Fig. 24.3

Geometry of the *inverse compositional* registration algorithm. I denotes the search image and R is the reference image. The geometric transformation T_p warps the image I_p back to the original search image I , with p being the initial parameter estimate.

Matching is performed between the (unwarped) reference image R and the warped search image I_p . Note that the reference image R always remains anchored at the origin. In each iteration, the incremental warp T_q (with parameter vector q) is estimated, mapping the image I_p to image $I_{p'}$. The resulting composite warp $T_{p'}$ (mapping $I_{p'}$ back to I) with parameters p' is obtained by concatenating the transformations T_q^{-1} and T_p .



where \circ denotes the concatenation (successive application) of transformations. In the special (but frequent) case of linear geometric transformations, the concatenation is simply accomplished by multiplying the corresponding transformation matrices M_p , $M_{q_{\text{opt}}}$, that is,

$$M_{p'} = M_p \cdot M_{q_{\text{opt}}}^{-1} \quad (24.34)$$

(see also Sec. 24.4.4). Also note that the “incremental” transformation $T_{q_{\text{opt}}}$ is *inverted* before it is concatenated with the current warp T_p , to calculate the parameters of the resulting composite warp $T_{p'}$. Thus the geometric transformation T must be invertible, but this is again no problem with linear (affine or projective) warps.

In summary, given a search image I , a reference image R , a geometric transformation T_p , an initial parameter estimate p_{init} and the convergence limit ϵ , the “inverse compositional algorithm” performs the following steps:

A. Initialize:

1. Calculate the gradient $\nabla_R(x)$ of the reference image R for all $x \in R$.
2. Calculate the Jacobian $J(x) = \frac{\partial T_p}{\partial p}(x)$ of the warp function $T_p(x)$ for all $x \in R$, with $p = \mathbf{0}$.
3. Compute $s_x = \nabla_R(x) \cdot J(x)$ for all $x \in R$.
4. Calculate the Hessian matrix as $H = \sum_R s_x^T \cdot s_x$ and pre-calculate its inverse H^{-1} .
5. Initialize the transformation parameters: $p \leftarrow p_{\text{init}}$.

B. Repeat:

6. Warp the search image I to I' , such that $I'(x) = I(T_p(x))$, for all $x \in R$.
7. Compute the (column) vector $\delta_p = \sum_R s_x \cdot [I'(x) - R(x)]$.
8. Estimate the optimal parameter change $q_{\text{opt}} = H^{-1} \cdot \delta_p$.
9. Find the warp parameters p' , such that $T_{p'} = T_{q_{\text{opt}}}^{-1} \circ T_p$.
10. Update the warp parameter $p \leftarrow p'$.

Until $\|q_{\text{opt}}\| < \epsilon$.

1: **LucasKanadeInverse**($I, R, T, p_{\text{init}}, \epsilon, i_{\text{max}}$)

Input: I , the search image; R , the reference image; T , a 2D warp function that maps any point $\mathbf{x} \in \mathbb{R}^2$ to $\mathbf{x}' = T_p(\mathbf{x})$ using parameters $p = (p_0, \dots, p_{n-1})$; p_{init} , initial estimate of the warp parameters; ϵ , the error limit (typ. $\epsilon = 10^{-3}$); i_{max} , the maximum number of iterations.

Returns the updated warp parameter vector p for the best fit between I and R , or nil if no match could be found.

```

2:  $(M_R, N_R) \leftarrow \text{Size}(R)$                                  $\triangleright$  size of the reference image  $R$ 
3:  $\mathbf{x}_c \leftarrow 0.5 \cdot (M_R - 1, N_R - 1)$                        $\triangleright$  center of  $R$ 
   Initialize:
4:  $n \leftarrow \text{Length}(p)$                                           $\triangleright$  parameter count  $n$ 
5: Create map  $S$ :  $(M_R \times N_R) \mapsto \mathbb{R}^n$   $\triangleright n$  “steepest-descent images”
6:  $(R_x, R_y) \leftarrow \text{Gradient}(R)$                                 $\triangleright (R_x(\mathbf{u}), R_y(\mathbf{u}))^\top = \nabla_R(\mathbf{u})$ 
7:  $\bar{\mathbf{H}} \leftarrow \mathbf{0}_{n,n}$                                       $\triangleright$  initialize  $n \times n$  Hessian matrix to zero
8: for all positions  $\mathbf{u} \in (M_R \times N_R)$  do
9:    $\mathbf{x} \leftarrow \mathbf{u} - \mathbf{x}_c$                                           $\triangleright$  centered position
10:   $\nabla_R \leftarrow (R_x(\mathbf{u}), R_y(\mathbf{u}))$                             $\triangleright$  2-dimensional row vector
11:   $\mathbf{J} \leftarrow \text{Jacobian}(T_0(\mathbf{x}))$                           $\triangleright$  Jacob. of  $T$  at pos.  $\mathbf{x}$  with  $p = \mathbf{0}$ 
12:   $\mathbf{s} \leftarrow (\nabla_R \cdot \mathbf{J})^\top$                                  $\triangleright$   $\mathbf{s}$  is a column vector of length  $n$ 
13:   $S(\mathbf{u}) \leftarrow \mathbf{s}$                                           $\triangleright$  keep  $\mathbf{s}$  for later use
14:   $\mathbf{H} \leftarrow \mathbf{s} \cdot \mathbf{s}^\top$                                  $\triangleright$  outer product,  $\mathbf{H}$  is of size  $n \times n$ 
15:   $\bar{\mathbf{H}} \leftarrow \bar{\mathbf{H}} + \mathbf{H}$                                  $\triangleright$  cumulate the Hessian (Eq. 24.30)
16:   $\bar{\mathbf{H}}^{-1} \leftarrow \text{Inverse}(\bar{\mathbf{H}})$ 
17:  if  $\bar{\mathbf{H}}^{-1} = \text{nil}$  then                                 $\triangleright$   $\bar{\mathbf{H}}$  could not be inverted
18:    return nil                                               $\triangleright$  stop
19:   $p \leftarrow p_{\text{init}}$                                           $\triangleright$  initial parameter estimate
20:   $i \leftarrow 0$                                              $\triangleright$  iteration counter

   Main loop:
21:  do
22:     $i \leftarrow i + 1$ 
23:     $\delta_p \leftarrow \mathbf{0}_n$                                           $\triangleright \delta_p \in \mathbb{R}^n$ , initialized to zero
24:    for all positions  $\mathbf{u} \in (M_R \times N_R)$  do
25:       $\mathbf{x} \leftarrow \mathbf{u} - \mathbf{x}_c$                                           $\triangleright$  centered position
26:       $\mathbf{x}' \leftarrow T_p(\mathbf{x})$                                         $\triangleright$  warp  $I$  to  $I'$ 
27:       $d \leftarrow \text{Interpolate}(I, \mathbf{x}') - R(\mathbf{u})$            $\triangleright$  pixel difference  $d \in \mathbb{R}$ 
28:       $\mathbf{s} \leftarrow S(\mathbf{u})$                                           $\triangleright$  get pre-calculated  $\mathbf{s}$ 
29:       $\delta_p \leftarrow \delta_p + \mathbf{s} \cdot d$ 
30:       $q_{\text{opt}} \leftarrow \mathbf{H}^{-1} \cdot \delta_p$                        $\triangleright \mathbf{H}^{-1}$  is pre-calculated in line 16
31:       $p' \leftarrow \text{determine, such that } T_{p'}(\mathbf{x}) = T_p(T_{q_{\text{opt}}}^{-1}(\mathbf{x}))$ 
32:       $p \leftarrow p'$ 
33:      while ( $\|q_{\text{opt}}\| > \epsilon$ )  $\wedge (i < i_{\text{max}})$            $\triangleright$  repeat until convergence
34:    return  $\begin{cases} p & \text{for } i < i_{\text{max}} \\ \text{nil} & \text{otherwise} \end{cases}$ 
```

24.3 INVERSE COMPOSITIONAL ALGORITHM

Alg. 24.2

Inverse compositional registration algorithm. The gradient vectors $\nabla_R(u, v)$ of the reference image R are calculated only once (line 6) using procedure `Gradient()`, as defined in Alg. 24.1. The Jacobian matrix \mathbf{J} of the warp function T_p is also evaluated only once (line 11) for $p = \mathbf{0}$ (i.e., the identity mapping) over all positions of the reference image R . Similarly, the Hessian matrix \mathbf{H} and its inverse \mathbf{H}^{-1} are calculated only once (lines 15, 16). \mathbf{H}^{-1} is used to calculate the optimal parameter change vector q_{opt} in line 30 of the main loop. Procedure `Interpolate()` in line 27 is the same as in Alg. 24.1. This algorithm is typically about 5–10 times faster than the original Lucas-Kanade (forward) algorithm (see Alg. 24.1), with similar convergence properties.

One can see clearly that in this variant several steps are performed only once at initialization and do not appear inside the main loop. A detailed and concise listing of the inverse compositional algorithm is given in Alg. 24.2 and concrete setups for various linear transforma-

tions are described in Sec. 24.4. Since the Jacobian matrix (for the null parameter vector $\mathbf{p} = \mathbf{0}$) and the Hessian matrix are calculated only once during initialization, this algorithm executes significantly faster than the original Lucas-Kanade (forward-additive) algorithm, while offering similar convergence properties.

24.4 Parameter Setups for Various Linear Transformations

The use of linear transformatons for the geometric mapping T is very common. In the following, we describe detailed setups required for the Lucas-Kanade algorithm for various geometric transformations, such as pure translation as well as affine and projective transformations. This should help to reduce the chance of confusion about the content and structure of the involved vectors and matrices. For additional details and concrete implementations of these transformations readers should consult the associated Java source code in the `imagingbook`⁹ library.

24.4.1 Pure Translation

In the case of pure 2D translation, we have $n = 2$ parameters t_x, t_y and the geometric transformation is (see Eqn. (24.15))

$$\dot{\mathbf{x}} = T_{\mathbf{p}}(\mathbf{x}) = \mathbf{x} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}, \quad (24.35)$$

with the parameter vector $\mathbf{p} = (p_0, p_1)^T = (t_x, t_y)^T$ and $\mathbf{x} = (x, y)^T$. Thus the two component functions of the transformation (cf. Eqn. (24.18)) are

$$\begin{aligned} T_{x,\mathbf{p}}(\mathbf{x}) &= x + t_x, \\ T_{y,\mathbf{p}}(\mathbf{x}) &= y + t_y, \end{aligned} \quad (24.36)$$

with the 2×2 Jacobian matrix

$$\mathbf{J}_{T_{\mathbf{p}}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial T_{x,\mathbf{p}}}{\partial t_x}(\mathbf{x}) & \frac{\partial T_{x,\mathbf{p}}}{\partial t_y}(\mathbf{x}) \\ \frac{\partial T_{y,\mathbf{p}}}{\partial t_x}(\mathbf{x}) & \frac{\partial T_{y,\mathbf{p}}}{\partial t_y}(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \quad (24.37)$$

Note that in this case $\mathbf{J}_{T_{\mathbf{p}}}(\mathbf{x})$ is constant,¹⁰ that is, independent of the position \mathbf{x} and the parameters \mathbf{p} . The 2D column vector $\delta_{\mathbf{p}}$ (Eqn. (24.26)) is calculated as

$$\delta_{\mathbf{p}} = \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(T_{\mathbf{p}}(\mathbf{u})) \cdot \mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u})]}_{\dot{\mathbf{u}} \in \mathbb{R}^2}^T \cdot \underbrace{[R(\mathbf{u}) - I(T_{\mathbf{p}}(\mathbf{u}))]}_{D(\mathbf{u}) \in \mathbb{R}} \quad (24.38)$$

$$= \sum_{\mathbf{u} \in R} \underbrace{[(I_x(\dot{\mathbf{u}}), I_y(\dot{\mathbf{u}})) \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}]}_{s(\mathbf{u}) = (s_0(\mathbf{u}), s_1(\mathbf{u}))}^T \cdot D(\mathbf{u}) = \sum_{\mathbf{u} \in R} \begin{pmatrix} I_x(\dot{\mathbf{u}}) \\ I_y(\dot{\mathbf{u}}) \end{pmatrix} \cdot D(\mathbf{u}) \quad (24.39)$$

$$= \begin{pmatrix} \sum_{\mathbf{u}} I_x(\dot{\mathbf{u}}) \cdot D(\mathbf{u}) \\ \sum_{\mathbf{u}} I_y(\dot{\mathbf{u}}) \cdot D(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \sum_{\mathbf{u}} s_0(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum_{\mathbf{u}} s_1(\mathbf{u}) \cdot D(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \delta_1 \end{pmatrix}, \quad (24.40)$$

⁹ Package `imagingbook.pub.geometry.mappings`.

¹⁰ \mathbf{I}_2 denotes the 2×2 identity matrix.

where I_x, I_y denote the (estimated) first derivatives of the search image I in x and y -direction, respectively.¹¹ Thus in this case the *steepest descent images* (Eqn. (24.28)) $s_0(\mathbf{x}) = I_x(\dot{\mathbf{x}})$ and $s_1(\mathbf{x}) = I_y(\dot{\mathbf{x}})$ are simply the components of the interpolated gradient of I in the region of the shifted reference image. The associated Hessian matrix (Eqn. (24.29)) is calculated as

$$\bar{\mathbf{H}} = \sum_{\mathbf{u} \in R} [\nabla_I(T_p(\mathbf{u})) \cdot \mathbf{J}_{T_p}(\mathbf{u})]^\top \cdot [\nabla_I(T_p(\mathbf{u})) \cdot \mathbf{J}_{T_p}(\mathbf{u})] \quad (24.41)$$

$$= \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(\dot{\mathbf{u}}) \cdot (\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix})]}_{s(\mathbf{u})}^\top \cdot \underbrace{[\nabla_I(\dot{\mathbf{u}}) \cdot (\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix})]}_{s(\mathbf{u})} = \sum_{\mathbf{u} \in R} s^\top(\mathbf{u}) \cdot s(\mathbf{u}) \quad (24.42)$$

$$= \sum_{\mathbf{u} \in R} \nabla_I^\top(\dot{\mathbf{u}}) \cdot \nabla_I(\dot{\mathbf{u}}) = \sum_{\mathbf{u} \in R} \begin{pmatrix} I_x(\dot{\mathbf{u}}) \\ I_y(\dot{\mathbf{u}}) \end{pmatrix} \cdot (I_x(\dot{\mathbf{u}}), I_y(\dot{\mathbf{u}})) \quad (24.43)$$

$$= \sum_{\mathbf{u} \in R} \begin{pmatrix} I_x^2(\dot{\mathbf{u}}) & I_x(\dot{\mathbf{u}}) \cdot I_y(\dot{\mathbf{u}}) \\ I_x(\dot{\mathbf{u}}) \cdot I_y(\dot{\mathbf{u}}) & I_y^2(\dot{\mathbf{u}}) \end{pmatrix} \quad (24.44)$$

$$= \begin{pmatrix} \sum I_x^2(\dot{\mathbf{u}}) & \sum I_x(\dot{\mathbf{u}}) \cdot I_y(\dot{\mathbf{u}}) \\ \sum I_x(\dot{\mathbf{u}}) \cdot I_y(\dot{\mathbf{u}}) & \sum I_y^2(\dot{\mathbf{u}}) \end{pmatrix} = \begin{pmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{pmatrix}, \quad (24.45)$$

again with $\dot{\mathbf{u}} = T_p(\mathbf{u})$. Since $\bar{\mathbf{H}}$ is symmetric ($H_{01} = H_{10}$) and only of size 2×2 , its *inverse* can be easily obtained in closed form:

$$\bar{\mathbf{H}}^{-1} = \frac{1}{H_{00} \cdot H_{11} - H_{01} \cdot H_{10}} \cdot \begin{pmatrix} H_{11} & -H_{01} \\ -H_{10} & H_{00} \end{pmatrix} \quad (24.46)$$

$$= \frac{1}{H_{00} \cdot H_{11} - H_{01}^2} \cdot \begin{pmatrix} H_{11} & -H_{01} \\ -H_{01} & H_{00} \end{pmatrix}. \quad (24.47)$$

The resulting optimal parameter increment (see Eqn. (24.25)) is

$$\mathbf{q}_{\text{opt}} = \begin{pmatrix} t'_x \\ t'_y \end{pmatrix} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_p = \bar{\mathbf{H}}^{-1} \cdot \begin{pmatrix} \delta_0 \\ \delta_1 \end{pmatrix} \quad (24.48)$$

$$= \frac{1}{H_{11} \cdot H_{22} - H_{12}^2} \cdot \begin{pmatrix} H_{11} \cdot \delta_0 - H_{01} \cdot \delta_1 \\ H_{00} \cdot \delta_1 - H_{01} \cdot \delta_0 \end{pmatrix}, \quad (24.49)$$

with δ_0, δ_1 as defined in Eqn. (24.40). Alternatively the same result could be obtained by solving Eqn. (24.31) for \mathbf{q}_{opt} .

24.4.2 Affine Transformation

An affine transformation in 2D can be expressed (for example) with homogeneous coordinates¹² in the form

$$T_p(\mathbf{x}) = \begin{pmatrix} 1 + a & b & t_x \\ c & 1 + d & t_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (24.50)$$

with $n = 6$ parameters $\mathbf{p} = (p_0, \dots, p_5)^\top = (a, b, c, d, t_x, t_y)^\top$. This parameterization of the affine transformation implies that the *null*

¹¹ See Sec. C.3.1 in the Appendix for how to estimate gradients of discrete images.

¹² See also Chapter 21, Secs. 21.1.2 and 21.1.3.

parameter vector ($\mathbf{p} = \mathbf{0}$) corresponds to the *identity* transformation. The component functions of this transformation thus are

$$\begin{aligned} T_{x,\mathbf{p}}(\mathbf{x}) &= (1+a) \cdot x + b \cdot y + t_x, \\ T_{y,\mathbf{p}}(\mathbf{x}) &= c \cdot x + (1+d) \cdot y + t_y, \end{aligned} \quad (24.51)$$

and the associated Jacobian matrix at some position $\mathbf{x} = (x, y)$ is

$$\mathbf{J}_{T_{\mathbf{p}}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial T_{x,\mathbf{p}}}{\partial a} & \frac{\partial T_{x,\mathbf{p}}}{\partial b} & \frac{\partial T_{x,\mathbf{p}}}{\partial c} & \frac{\partial T_{x,\mathbf{p}}}{\partial d} & \frac{\partial T_{x,\mathbf{p}}}{\partial t_x} & \frac{\partial T_{x,\mathbf{p}}}{\partial t_y} \\ \frac{\partial T_{y,\mathbf{p}}}{\partial a} & \frac{\partial T_{y,\mathbf{p}}}{\partial b} & \frac{\partial T_{y,\mathbf{p}}}{\partial c} & \frac{\partial T_{y,\mathbf{p}}}{\partial d} & \frac{\partial T_{y,\mathbf{p}}}{\partial t_x} & \frac{\partial T_{y,\mathbf{p}}}{\partial t_y} \end{pmatrix}(\mathbf{x}) \quad (24.52)$$

$$= \begin{pmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{pmatrix}. \quad (24.53)$$

Note that in this case the Jacobian only depends on the position $\mathbf{x} = (x, y)$, not on the transformation parameters \mathbf{p} . It can thus be pre-calculated once for all positions \mathbf{x} of the reference image R . The 6-dimensional column vector $\delta_{\mathbf{p}}$ (Eqn. (24.26)) is obtained as

$$\delta_{\mathbf{p}} = \sum_{\mathbf{u} \in R} \underbrace{[\nabla_I(T_{\mathbf{p}}(\mathbf{u})) \cdot \mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u})]}_{s(\mathbf{u})}^T \cdot \underbrace{[R(\mathbf{u}) - I(T_{\mathbf{p}}(\mathbf{u}))]}_{D(\mathbf{u})} \quad (24.54)$$

$$= \sum_{\mathbf{u} \in R} \left[(I_x(\dot{\mathbf{u}}), I_y(\dot{\mathbf{u}})) \cdot \begin{pmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{pmatrix} \right]^T \cdot D(\mathbf{u}) \quad (24.55)$$

$$= \sum_{\mathbf{u} \in R} \begin{pmatrix} I_x(\dot{\mathbf{u}}) \cdot x \\ I_x(\dot{\mathbf{u}}) \cdot y \\ I_y(\dot{\mathbf{u}}) \cdot x \\ I_y(\dot{\mathbf{u}}) \cdot y \\ I_x(\dot{\mathbf{u}}) \\ I_y(\dot{\mathbf{u}}) \end{pmatrix} \cdot D(\mathbf{u}) = \sum_{\mathbf{u} \in R} \begin{pmatrix} s_0(\mathbf{u}) \\ s_1(\mathbf{u}) \\ s_2(\mathbf{u}) \\ s_3(\mathbf{u}) \\ s_4(\mathbf{u}) \\ s_5(\mathbf{u}) \end{pmatrix} \cdot D(\mathbf{u}) \quad (24.56)$$

$$= \sum_{\mathbf{u} \in R} \begin{pmatrix} s_0(\mathbf{u}) \cdot D(\mathbf{u}) \\ s_1(\mathbf{u}) \cdot D(\mathbf{u}) \\ s_2(\mathbf{u}) \cdot D(\mathbf{u}) \\ s_3(\mathbf{u}) \cdot D(\mathbf{u}) \\ s_4(\mathbf{u}) \cdot D(\mathbf{u}) \\ s_5(\mathbf{u}) \cdot D(\mathbf{u}) \end{pmatrix} = \begin{pmatrix} \sum s_0(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum s_1(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum s_2(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum s_3(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum s_4(\mathbf{u}) \cdot D(\mathbf{u}) \\ \sum s_5(\mathbf{u}) \cdot D(\mathbf{u}) \end{pmatrix}, \quad (24.57)$$

again with $\dot{\mathbf{u}} = T_{\mathbf{p}}(\mathbf{u})$. The corresponding Hessian matrix (of size 6×6) is found as

$$\bar{\mathbf{H}} = \sum_{\mathbf{u} \in R} [\nabla_I(T_{\mathbf{p}}(\mathbf{u})) \cdot \mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u})]^T \cdot [\nabla_I(T_{\mathbf{p}}(\mathbf{u})) \cdot \mathbf{J}_{T_{\mathbf{p}}}(\mathbf{u})] \quad (24.58)$$

$$= \sum_{\mathbf{u} \in R} \mathbf{s}^T(\mathbf{u}) \cdot \mathbf{s}(\mathbf{u}) = \sum_{\mathbf{x} \in R} \begin{pmatrix} I_x(\dot{\mathbf{u}}) \cdot x \\ I_x(\dot{\mathbf{u}}) \cdot y \\ I_y(\dot{\mathbf{u}}) \cdot x \\ I_y(\dot{\mathbf{u}}) \cdot y \\ I_x(\dot{\mathbf{u}}) \\ I_y(\dot{\mathbf{u}}) \end{pmatrix}^T \cdot \begin{pmatrix} I_x(\dot{\mathbf{u}}) \cdot x \\ I_x(\dot{\mathbf{u}}) \cdot y \\ I_y(\dot{\mathbf{u}}) \cdot x \\ I_y(\dot{\mathbf{u}}) \cdot y \\ I_x(\dot{\mathbf{u}}) \\ I_y(\dot{\mathbf{u}}) \end{pmatrix} = \quad (24.59)$$

$$\begin{pmatrix} \Sigma I_x^2(\dot{\mathbf{u}}) x^2 & \Sigma I_x^2(\dot{\mathbf{u}}) xy & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x^2 & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) xy & \Sigma I_x^2(\dot{\mathbf{u}}) x & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x \\ \Sigma I_x^2(\dot{\mathbf{u}}) xy & \Sigma I_x^2(\dot{\mathbf{u}}) y^2 & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) xy & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y^2 & \Sigma I_x^2(\dot{\mathbf{u}}) y & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y \\ \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x^2 & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) xy & \Sigma I_y^2(\dot{\mathbf{u}}) x^2 & \Sigma I_y^2(\dot{\mathbf{u}}) xy & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x & \Sigma I_y^2(\dot{\mathbf{u}}) x \\ \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) xy & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y^2 & \Sigma I_y^2(\dot{\mathbf{u}}) xy & \Sigma I_y^2(\dot{\mathbf{u}}) y^2 & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y & \Sigma I_y^2(\dot{\mathbf{u}}) y \\ \Sigma I_x^2(\dot{\mathbf{u}}) x & \Sigma I_x^2(\dot{\mathbf{u}}) y & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y & \Sigma I_x^2(\dot{\mathbf{u}}) & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) \\ \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) x & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) y & \Sigma I_y^2(\dot{\mathbf{u}}) x & \Sigma I_y^2(\dot{\mathbf{u}}) y & \Sigma I_x(\dot{\mathbf{u}}) I_y(\dot{\mathbf{u}}) & \Sigma I_y^2(\dot{\mathbf{u}}) \end{pmatrix}. \quad (24.60)$$

Finally, the optimal parameter increment (see Eqn. (24.25)) is calculated as

$$\mathbf{q}_{\text{opt}} = (a', b', c', d', t'_x, t'_y)^\top = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_p \quad (24.61)$$

or, equivalently, by solving $\mathbf{H} \cdot \mathbf{q}_{\text{opt}} = \boldsymbol{\delta}_p$ (see Eqn. (24.31)). For both approaches, no closed-form solution is possible but numerical methods must be used.

24.4.3 Projective Transformation

A projective transformation¹³ can be expressed (for example) with homogeneous coordinates in the form

$$T_p(\mathbf{x}) = \mathbf{M}_p \cdot \mathbf{x} = \begin{pmatrix} 1+a & b & t_x \\ c & 1+d & t_y \\ e & f & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (24.62)$$

with $n = 8$ parameters $\mathbf{p} = (p_0, \dots, p_7) = (a, b, c, d, e, f, t_x, t_y)$. Again the null parameter vector corresponds to the identity transformation. In this case, the results need to be converted back to non-homogeneous coordinates (see Ch. 21, Sec. 21.1.2), which yields the transformation's effective (nonlinear) component functions

$$T_{x,p}(\mathbf{x}) = \frac{(1+a) \cdot x + b \cdot y + t_x}{e \cdot x + f \cdot y + 1} = \frac{\alpha}{\gamma}, \quad (24.63)$$

$$T_{y,p}(\mathbf{x}) = \frac{c \cdot x + (1+d) \cdot y + t_y}{e \cdot x + f \cdot y + 1} = \frac{\beta}{\gamma}, \quad (24.64)$$

with $\mathbf{x} = (x, y)$ and

$$\alpha = (1+a) \cdot x + b \cdot y + t_x, \quad (24.65)$$

$$\beta = c \cdot x + (1+d) \cdot y + t_y, \quad (24.66)$$

$$\gamma = e \cdot x + f \cdot y + 1. \quad (24.67)$$

In this case, the associated Jacobian matrix for position $\mathbf{x} = (x, y)$,

$$\begin{aligned} \mathbf{J}_{T_p}(\mathbf{x}) &= \begin{pmatrix} \frac{\partial T_{x,p}}{\partial a} & \frac{\partial T_{x,p}}{\partial b} & \frac{\partial T_{x,p}}{\partial c} & \frac{\partial T_{x,p}}{\partial d} & \frac{\partial T_{x,p}}{\partial e} & \frac{\partial T_{x,p}}{\partial f} & \frac{\partial T_{x,p}}{\partial t_x} & \frac{\partial T_{x,p}}{\partial t_y} \\ \frac{\partial T_{y,p}}{\partial a} & \frac{\partial T_{y,p}}{\partial b} & \frac{\partial T_{y,p}}{\partial c} & \frac{\partial T_{y,p}}{\partial d} & \frac{\partial T_{y,p}}{\partial e} & \frac{\partial T_{y,p}}{\partial f} & \frac{\partial T_{y,p}}{\partial t_x} & \frac{\partial T_{y,p}}{\partial t_y} \end{pmatrix}(\mathbf{x}) \\ &= \frac{1}{\gamma} \cdot \begin{pmatrix} x & y & 0 & 0 & -\frac{x \cdot \alpha}{\gamma} & -\frac{y \cdot \alpha}{\gamma} & 1 & 0 \\ 0 & 0 & x & y & -\frac{x \cdot \beta}{\gamma} & -\frac{y \cdot \beta}{\gamma} & 0 & 1 \end{pmatrix}, \end{aligned} \quad (24.68)$$

depends on both the position \mathbf{x} as well as the transformation parameters \mathbf{p} . The setup for the resulting Hessian matrix \mathbf{H} is analogous to Eqns. (24.58)–(24.61).

24.4.4 Concatenating Linear Transformations

The “inverse compositional” algorithm described in Sec. 24.3 requires the concatenation of geometric transformations (see Eqn. (24.33)). In

¹³ See also Chapter 21, Sec. 21.1.4.

particular, if T_p, T_q are *linear* transformations (in homogeneous coordinates, see Eqn. (24.62)), with associated transformation matrices \mathbf{M}_p and \mathbf{M}_q (such that $T_p(\mathbf{x}) = \mathbf{M}_p \cdot \mathbf{x}$ and $T_q(\mathbf{x}) = \mathbf{M}_q \cdot \mathbf{x}$, respectively), the matrix for the concatenated transformation,

$$T_{p'}(\mathbf{x}) = (T_p \circ T_q)(\mathbf{x}) = T_q(T_p(\mathbf{x})) \quad (24.69)$$

is simply the product of the original matrices, that is,

$$\mathbf{M}_{p'} \cdot \mathbf{x} = \mathbf{M}_q \cdot \mathbf{M}_p \cdot \mathbf{x}. \quad (24.70)$$

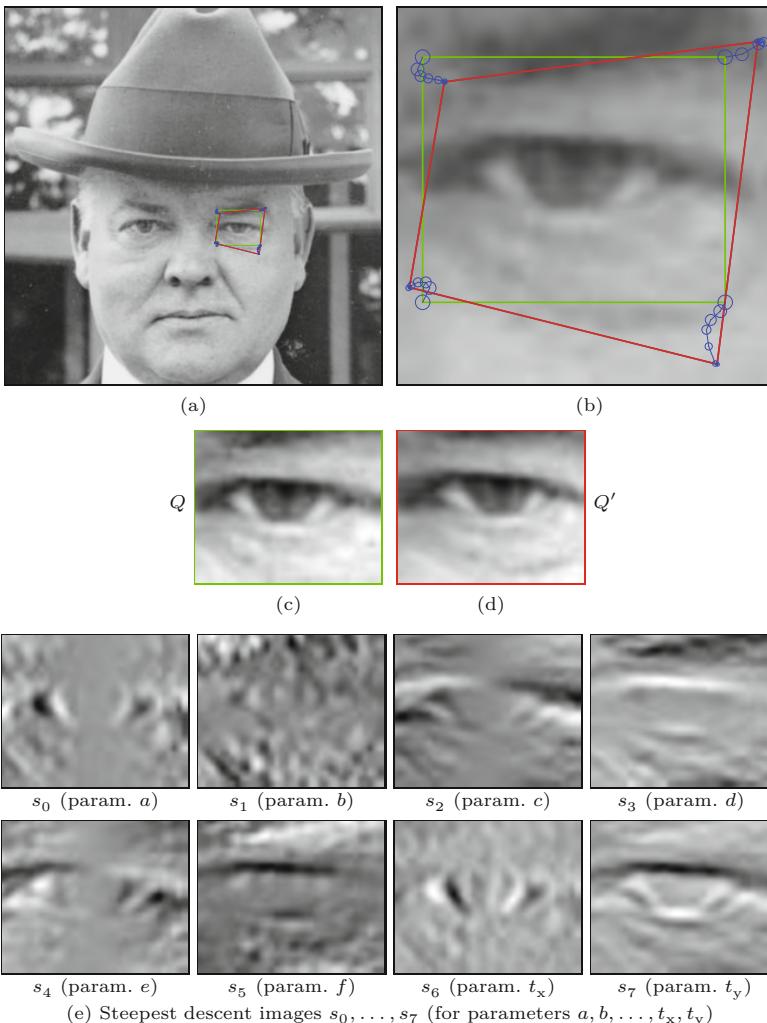
The resulting parameter vector p' for the composite transformation $T_{p'}$ can be simply extracted from the corresponding elements of the matrix $\mathbf{M}_{p'}$ (see Eqn. (24.50) and Eqn. (24.62)), respectively.

24.5 Example

[Figure 24.4](#) shows an example for using the classic Lucas-Kanade (forward-additive) matcher. Initially, a rectangular region Q is selected in the search image I , marked by the green rectangle in [Fig. 24.4\(a,b\)](#), which specifies the approximate position of the reference image. To create the (synthetic) reference image R , all four corners of the rectangle Q were perturbed randomly in x - and y -direction by Gaussian noise (with $\sigma = 2.5$) in x - and y -direction. The resulting quadrilateral Q' (red outline in [Fig. 24.4\(a,b\)](#)) specifies the region in image I where the reference image R was extracted by transformation and interpolation (see [Fig. 24.4\(d\)](#)). The matching process starts from the rectangle Q , which specifies the *initial* warp transformation T_{init} , given by the green rectangle (Q), while the real (but unknown) transformation corresponds to the red quadrilateral (Q'). Each iteration of the matcher updates the warp transformation T . The blue circles in [Fig. 24.4\(b\)](#) mark the corners of the back-projected reference frame under the changing transformation T ; the radius of the circles corresponds to the remaining registration error between the reference image R and the current subimage of I .

[Figure 24.4\(e\)](#) shows the steepest-descent images s_0, \dots, s_7 (see Eqn. (24.28)) for the first iteration. Each of these images is of the same size as R and corresponds to one of the 8 parameters $a, b, c, d, e, f, t_x, t_y$ of the projective warp transformation (see Eqn. (24.62)). The value $s_k(u, v)$ in a particular image s_k corresponds to the optimal change of the transformation parameter k with respect to the associated image position (u, v) . The actual change of parameter k is calculated by averaging over all positions (u, v) of the reference image R .

The example demonstrates the robustness and fast convergence of the classic Lucas-Kanade matcher, which typically requires only 5–20 iterations. In this case, the matcher performed 7 iterations to converge (with convergence limit $\epsilon = 0.00001$). In comparison, the inverse-compositional matcher typically requires more iterations and is less tolerant to deviations of the initial warp transformation,



24.6 JAVA IMPLEMENTATION

Fig. 24.4
 Lucas-Kanade (forward-additive) matcher with projective warp transformation. Original image I (a); the initial warp transformation T_{init} is visualized by the green rectangle Q , which corresponds to the subimage shown in (c). The actual reference image R (d) has been extracted from the red quadrilateral Q' (by transformation and interpolation). The blue circles mark the corners of the back-projected reference image under the changing transformation T_p . The radius of each circle corresponds to the registration error between the transformed reference image R and the currently overlapping part of the search image I . The *steepest-descent images* s_0, \dots, s_7 (one for each of the 8 parameters $a, b, c, d, e, f, t_x, t_y$ of the projective transformation) for the first iteration are shown in (e). These images are of the same size as the reference image R .

that is, has a smaller convergence range than the additive-forward algorithm.¹⁴

24.6 Java Implementation

The algorithms described in this chapter have been implemented in Java, with the source code available as part of the `imagingbook`¹⁵ library on the book's accompanying website. As usual, most Java variables and methods in the online code have been named similarly to the identifiers used in the text for easier understanding.

¹⁴ In fact, the inverse-compositional algorithm does not converge with this particular example.

¹⁵ Package `imagingbook.pub.lucaskanade`.

LucasKanadeMatcher (class)

This is the (abstract) super-class of the concrete matchers (`ForwardAdditiveMatcher`, `InverseCompositionalMatcher`) described further. It defines a static inner class `Parameters`¹⁶ with public parameter fields such as

```
tolerance (=  $\epsilon$ , default 0.00001),  
maxIterations (=  $i_{\max}$ , default 100).
```

In addition, class `LucasKanadeMatcher` itself provides the following public methods:

`LinearMapping getMatch (ProjectiveMapping T)`

Performs a complete match on the given image pair I , R (required by the sub-class constructors), with T used as the initial geometric transformation. The transformation object T may be of any subtype of `ProjectiveMapping`,¹⁷ including `Translation` and `AffineMapping`. The method returns a new transformation object for the optimal match, or `null` if the matcher did not converge.

`ProjectiveMapping iterateOnce (ProjectiveMapping T)`

This method performs a single matching iteration with the current warp transformation T . It is typically invoked repeatedly after an initial call to `initializeMatch()`. The updated warp transformation is returned, or `null` if the iteration was unsuccessful (e.g., if the Hessian matrix could not be inverted).

`boolean hasConverged ()`

Returns `true` if (and only if) the minimization criteria (specified by the `tolerance` parameter) have been reached. This method is typically used to terminate the optimization loop after calling `iterateOnce()`.

`Point2D[] getReferencePoints ()`

Returns the four corner points of the bounding rectangle of the reference image R , centered at the origin. All warp transformations (including T_{init} and T_p) refer to these coordinates. Note that the returned point coordinates are generally non-integer values; for example, for a reference image size 11×8 , the reference corner points are $A = (-5, -3.5)$, $B = (5, -3.5)$, $C = (5, 3.5)$, and $D = (-5, 3.5)$ (see Fig. 24.5).

`ProjectiveMapping getReferenceMappingTo (Point2D[] Q)`

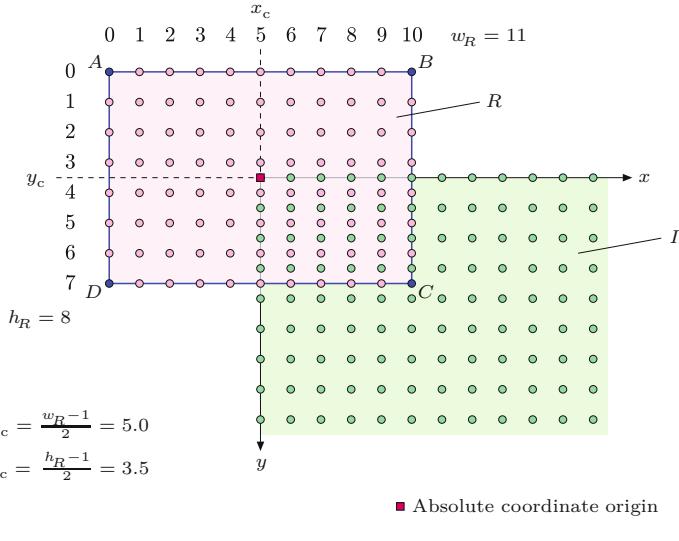
Calculates the (linear) geometric transformation between the reference image R (centered at the origin) and the quadrilateral specified by the point sequence Q . The type of the returned mapping depends on the number of points in Q (max. 4).

`double getRmsError ()`

Returns the RMS error between images I and R for the most recent iteration (usually called after `iterateOnce()`).

¹⁶ See the usage example in Prog. 24.1.

¹⁷ Class `ProjectiveMapping` is described in Chapter 21, Sec. 21.1.4.



24.6 JAVA IMPLEMENTATION

Fig. 24.5

Reference coordinates. The center of the reference image R is aligned with the origin of the search image I (red square), which is taken as the absolute origin. Image samples (indicated by round dots) are assumed to be located at integer positions. In this example, the reference image R is of size $w_R = 11$ and $h_R = 8$, thus the center coordinates are $x_c = 5.0$ and $y_c = 3.5$. In the x/y coordinate frame of I (i.e., absolute coordinates), the four corners of R 's bounding rectangle are $A = (-5, -3.5)$, $B = (5, -3.5)$, $C = (5, 3.5)$ and $D = (-5, 3.5)$. All warp transformations refer to these reference points (cf. Figs. 24.2 and 24.3).

LucasKanadeForwardMatcher (class)

This sub-class of `LucasKanadeMatcher` implements the Lucas-Kanade (“forward-additive”) algorithm, as outlined in Alg. 24.1. It provides the aforementioned methods for `LucasKanadeMatcher` and two constructors:

```
LucasKanadeForwardMatcher (FloatProcessor I,
                           FloatProcessor R)
```

Here I is the search image, R is the (smaller) reference image. It creates a new instance of `LucasKanadeForwardMatcher` using default parameter values.

```
LucasKanadeForwardMatcher (FloatProcessor I,
                           FloatProcessor R, Parameters params)
```

Creates a new instance of type `LucasKanadeForwardMatcher` using the specific settings in `params`.

LucasKanadeInverseMatcher (class)

This sub-class of `LucasKanadeMatcher` implements the “inverse compositional” algorithm, as described in Alg. 24.2. It provides the same methods and constructors as class `LucasKanadeForwardMatcher`:

```
LucasKanadeInverseMatcher (FloatProcessor I,
                           FloatProcessor R).
```

```
LucasKanadeInverseMatcher (FloatProcessor I,
                           FloatProcessor R, Parameters params).
```

24.6.1 Application Example

The code example in Prog. 24.1 demonstrates the use of the Lucas-Kanade API. The ImageJ plugin is applied to the search image I (the current image) and requires a rectangular ROI to be selected, which is taken as the initial guess for the match region. The reference image is created synthetically by extracting a warped sub-image

24 NON-RIGID IMAGE MATCHING

Prog. 24.1

Lucas-Kanade code example (ImageJ plugin). This plugin is applied to the search image (I) and assumes that a rectangular ROI is selected whose bounding rectangle and corner points (Q) are obtained in lines 22–27. The search image I is copied from the current image (as a `FloatProcessor` object) in line 19. The size of the reference image R (created in line 24) is defined by the ROI rectangle, whose corner points Q also determine the initial parameters of the geometric transformation T_{init} (line 27 and 37, respectively). The synthetic reference image R (with the same size as the ROI) is extracted from the search image by warping from a quadrilateral (QQ'), which is obtained by randomly perturbing the corner points of the selected ROI (lines 28–29). A new matcher object is created in lines 32–33, in this case of type `LucasKanadeForwardMatcher` (alternatively, `LucasKanadeInverseMatcher` could have been used). The actual match operation is performed in lines 40–44. It consists of a simple `do-while` loop which is terminated if either, the transformation T becomes invalid (`null`), the matcher has converged or the maximum number of iterations has been reached. Alternatively, lines 40–44 could have been replaced by the statement $T = \text{matcher.getMatch}(T_{init})$.

If the matcher has converged, the final transformation T_p maps to the best-matching sub-image of I .

```
1 import ...
2
3 public class LucasKanade_Demo implements PlugInFilter {
4
5     static int maxIterations = 100;
6
7     public int setup(String args, ImagePlus img) {
8         return DOES_8G + ROI_REQUIRED;
9     }
10
11    public void run(ImageProcessor ip) {
12        Roi roi = img.getRoi();
13        if (roi != null && roi.getType() != Roi.RECTANGLE) {
14            IJ.error("Rectangular selection required!");
15            return;
16        }
17
18        // Step 1: create the search image  $I$ :
19        FloatProcessor I = ip.convertToFloatProcessor();
20
21        // Step 2: create the (empty) reference image  $R$ :
22        Rectangle roiR = roi.getBounds();
23        FloatProcessor R =
24            new FloatProcessor(roiR.width, roiR.height);
25
26        // Step 3: perturb the rectangle  $Q$  to  $Q'$  to extract reference image  $R$ :
27        Point2D[] Q = getCornerPoints(roiR); // =  $Q$ 
28        Point2D[] QQ = perturbGaussian(Q); // =  $Q'$ 
29        (new ImageExtractor(I)).extractImage(R, QQ);
30
31        // Step 4: create the Lucas-Kanade matcher (forward or inverse):
32        LucasKanadeMatcher matcher =
33            new LucasKanadeForwardMatcher(I, R);
34
35        // Step 5: calculate the initial mapping  $T_{init}$ :
36        ProjectiveMapping Tinit =
37            matcher.getReferenceMappingTo(Q);
38
39        // Step 6: initialize and run the matching loop:
40        ProjectiveMapping T = Tinit;
41        do {
42            T = matcher.iterateOnce(T);
43        } while (T != null && !matcher.hasConverged() &&
44                  matcher.getIteration() < maxIterations);
45
46        // Step 7: evaluate the result:
47        if (T == null || !matcher.hasConverged()) {
48            IJ.log("no match found!");
49            return;
50        }
51        else {
52            ProjectiveMapping Tfinal = T;
53            ...
54        }
55
56    }
```

of I from a random quadrilateral around the selected ROI.¹⁸ The required geometric transformations (such as `ProjectiveMapping`, `AffineMapping`, `Translation` etc.) are described in Chapter 21, Sec. 21.1.

The example demonstrates how the Lucas-Kanade matcher is initialized and called repeatedly inside the optimization loop using a projective transformation. This usage mode is specifically intended for testing purposes, since it allows to retrieve the state of the matcher after every iteration. The same result could be obtained by replacing the whole loop (lines 40–44 in Prog. 24.1) with the single instruction

```
ProjectiveMapping T = matcher.getMatch(Tinit);
```

Moreover, in line 33, the `LucasKanadeForwardMatcher` could be replaced by an instance of `LucasKanadeInverseMatcher` without any additional changes. For further details, see the complete source code on the book’s website.

24.7 Exercises

Exercise 24.1. Determine the general structure of the Hessian matrix for the projective transformation (see Sec. 24.4.3), analogous to the affine transformation in Eqns. (24.58)–(24.60).

Exercise 24.2. Create comparative statistics of the convergence properties of the classes `ForwardAdditiveMatcher` and `InverseCompositionalMatcher` by evaluating the number of iterations required including the percentage of failures. Use a test scenario with randomly perturbed reference regions as shown in Prog. 24.1.

Exercise 24.3. It is sometimes suggested to refine the warp transformation step-by-step instead of using the full transformation for the whole matching process. For example, one could first match with a pure translation model, then—starting from the result of the first match—switch to an affine transformation model, and eventually apply a full projective transformation. Explore this idea and find out whether this can yield a more robust matching process.

Exercise 24.4. Adapt the 2D Lucas-Kanade method described in Sec. 24.2 for the registration of discrete 1D signals under shifting and scaling. Given is a search signal $I(u)$, for $u = 0, \dots, M_I - 1$, and a reference signal $R(u)$, for $u = 0, \dots, M_R - 1$. It is assumed that I contains a transformed version of R , which is specified by the mapping $T_p(x) = s \cdot x + t$, with the two unknown parameters $p = (s, t)$. A practical application could be the registration of neighboring image lines under perspective distortion.

Exercise 24.5. Use the Lucas-Kanade matcher to design a tracker that follows a given reference patch through a sequence of N images. Hint: In ImageJ, an image sequence (AVI-video or multi-frame TIFF)

¹⁸ The class `ImageExtractor`, used to extract the warped sub-image, is part of the `imagingbook` library (package `imagingbook.lib.image`).

can be imported as an `ImageStack` and simply processed frame-by-frame. Select the original reference patch in the first frame of the image sequence and use its position to calculate the initial warp transformation to find a match in the second image. Subsequently, take the match obtained in the second image as the initial transformation for the third image, etc. Consider two approaches: (a) use the initial patch as the reference image for *all* frames of the sequence or (b) extract a new reference image for each pair of frames.

Scale-Invariant Feature Transform (SIFT)

Many real applications require the localization of reference positions in one or more images, for example, for image alignment, removing distortions, object tracking, 3D reconstruction, etc. We have seen that corner points¹ can be located quite reliably and independent of orientation. However, typical corner detectors only provide the position and strength of each candidate point, they do not provide any information about its characteristic or “identity” that could be used for matching. Another limitation is that most corner detectors only operate at a particular scale or resolution, since they are based on a rigid set of filters.

This chapter describes the *Scale-Invariant Feature Transform* (SIFT) technique for local feature detection, which was originally proposed by D. Lowe [152] and has since become a “workhorse” method in the imaging industry. Its goal is to locate image features that can be identified robustly to facilitate matching in multiple images and image sequences as well as object recognition under different viewing conditions. SIFT employs the concept of “scale space” [151] to capture features at *multiple* scale levels or image resolutions, which not only increases the number of available features but also makes the method highly tolerant to scale changes. This makes it possible, for example, to track features on objects that move towards the camera and thereby change their scale continuously or to stitch together images taken with widely different zoom settings.

Accelerated variants of the SIFT algorithm have been implemented by streamlining the scale space calculation and feature detection or the use of GPU hardware [20, 90, 218].

In principle, SIFT works like a multi-scale corner detector with sub-pixel positioning accuracy and a rotation-invariant feature descriptor attached to each candidate point. This (typically 128-dimensional) feature descriptor summarizes the distribution of the gradient directions in a spatial neighborhood around the corresponding feature point and can thus be used like a “fingerprint”. The main steps involved in the calculation of SIFT features are as follows:

¹ See Chapter 7.

1. Extrema detection in a Laplacian-of-Gaussian (LoG) scale space to locate potential interest points.
2. Key point refinement by fitting a continuous model to determine precise location and scale.
3. Orientation assignment by the dominant orientation of the feature point from the directions of the surrounding image gradients.
4. Formation of the feature descriptor by normalizing the local gradient histogram.

These steps are all described in the remaining parts of this chapter. There are several reasons why we explain the SIFT technique here at such great detail. For one, it is by far the most complex algorithm that we have looked at so far, its individual steps are carefully designed and delicately interdependent, with numerous parameters that need to be considered. A good understanding of the inner workings and limitations is thus important for successful use as well as for analyzing problems if the results are not as expected.

25.1 Interest Points at Multiple Scales

The first step in detecting interest points is to find locations with stable features that can be localized under a wide range of viewing conditions and different scales. In the SIFT approach, interest point detection is based on Laplacian-of-Gaussian (LoG) filters, which respond primarily to distinct bright blobs surrounded by darker regions, or vice versa. Unlike the filters used in popular corner detectors,² LoG filters are *isotropic*, i.e., insensitive to orientation. To locate interest points over multiple scales, a scale space representation of the input image is constructed by recursively smoothing the image with a sequence of small Gaussian filters. The difference between the images in adjacent scale layers is used to approximate the LoG filter at each scale. Interest points are finally selected by finding the local maxima in the 3D LoG scale space.

25.1.1 The LoG Filter

In this section, we first outline LoG filters and the basic construction of a Gaussian scale space, followed by a detailed description of the actual implementation and the parameters used in the SIFT approach.

The LoG is a so-called *center-surround* operator, which most strongly responds to isolated local intensity peaks, edge, and corner-like image structures. The corresponding filter kernel is based on the second derivative of the Gaussian function, as illustrated in Fig. 25.1 for the 1D case. The 1D Gaussian function of width σ is defined as

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{x^2}{2\sigma^2}} \quad (25.1)$$

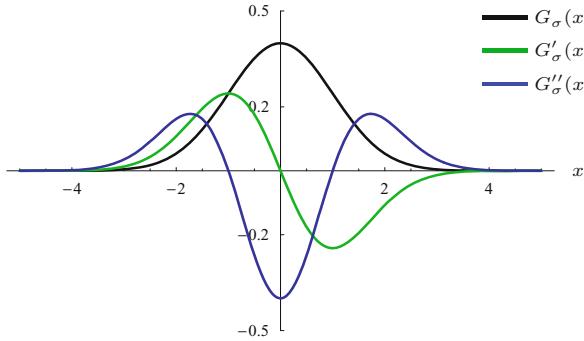
and its *first* derivative is

² See Chapter 7.

25.1 INTEREST POINTS AT MULTIPLE SCALES

Fig. 25.1

1D Gaussian function $G_\sigma(x)$ with $\sigma = 1$ (black), its first derivative $G'_\sigma(x)$ (green) and second derivative $G''_\sigma(x)$ (blue).



$$G'_\sigma(x) = \frac{dG_\sigma}{dx}(x) = -\frac{x}{\sqrt{2\pi} \cdot \sigma^3} \cdot e^{-\frac{x^2}{2\sigma^2}}. \quad (25.2)$$

Analogously, the *second* derivative of the 1D Gaussian is

$$G''_\sigma(x) = \frac{d^2G_\sigma}{dx^2}(x) = \frac{x^2 - \sigma^2}{\sqrt{2\pi} \cdot \sigma^5} \cdot e^{-\frac{x^2}{2\sigma^2}}. \quad (25.3)$$

The *Laplacian* (denoted ∇^2) of a continuous, 2D function $f(x, y)$ is defined as the sum of the second partial derivatives for the x - and y -directions, traditionally written as

$$(\nabla^2 f)(x, y) = \frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y). \quad (25.4)$$

Note that, unlike the *gradient*³ of a 2D function, the result of the Laplacian is not a vector but a *scalar* quantity. Its value is invariant against rotations of the coordinate system, that is, the Laplacian operator has the important property of being *isotropic*.

By applying the *Laplacian* operator to a rotationally symmetric 2D Gaussian,

$$G_\sigma(x, y) = \frac{1}{2\pi \cdot \sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (25.5)$$

with identical widths $\sigma = \sigma_x = \sigma_y$ in the x/y directions (see Fig. 25.2(a)), we obtain the LoG function

$$\begin{aligned} L_\sigma(x, y) &= (\nabla^2 G_\sigma)(x, y) = \frac{\partial^2 G_\sigma}{\partial x^2}(x, y) + \frac{\partial^2 G_\sigma}{\partial y^2}(x, y) \\ &= \frac{(x^2 - \sigma^2)}{2\pi \cdot \sigma^6} \cdot e^{-\frac{x^2+y^2}{2\cdot\sigma^2}} + \frac{(y^2 - \sigma^2)}{2\pi \cdot \sigma^6} \cdot e^{-\frac{x^2+y^2}{2\cdot\sigma^2}} \\ &= \frac{1}{\pi \cdot \sigma^4} \cdot \left(\frac{x^2 + y^2 - 2\sigma^2}{2 \cdot \sigma^2} \right) \cdot e^{-\frac{x^2+y^2}{2\cdot\sigma^2}}, \end{aligned} \quad (25.6)$$

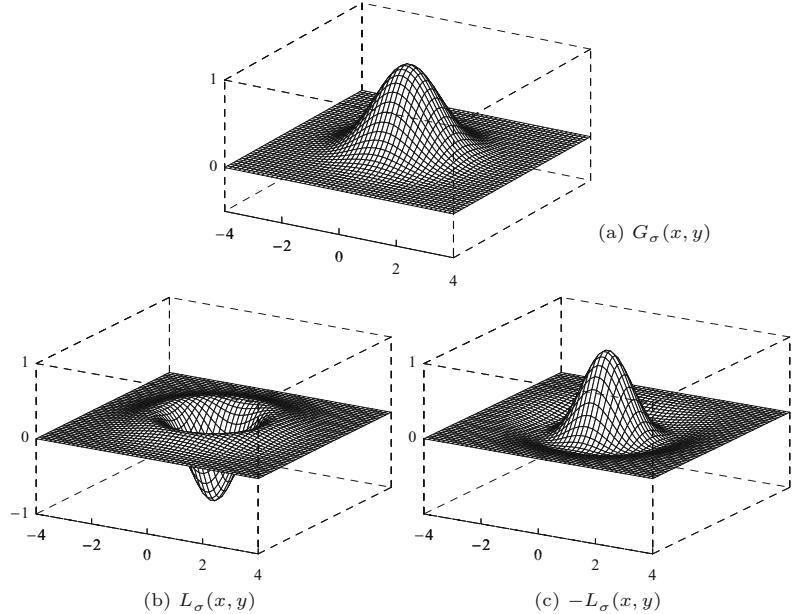
as shown in Fig. 25.2(b). The continuous LoG function in Eqn. (25.6) has the absolute value integral

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} |L_\sigma(x, y)| dx dy = \frac{4}{\sigma^2 e}, \quad (25.7)$$

³ See Chapter 6, Sec. 6.2.1.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.2
2D Gaussian and LoG. Gaussian function $G_\sigma(x, y)$ with $\sigma = 1$ (a); the corresponding LoG function $L_\sigma(x, y)$ in (b), and the inverted function (“Mexican hat” or “Sombrero” kernel) $-L_\sigma(x, y)$ in (c). For illustration, all three functions are normalized to an absolute value of 1 at the origin.



and zero average, that is,

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} L_\sigma(x, y) \, dx \, dy = 0. \quad (25.8)$$

When used as the kernel of a linear filter,⁴ the LoG responds maximally to circular spots that are *darker* than the surrounding background and have a radius of approximately σ .⁵ Blobs that are *brighter* than the surrounding background are enhanced by filtering with the negative LoG kernel, that is, $-L_\sigma$, which is often referred to as the “Mexican hat” or “Sombrero” filter (see Fig. 25.2). Both types of blobs can be detected simultaneously by simply taking the absolute value of the filter response (see Fig. 25.3).

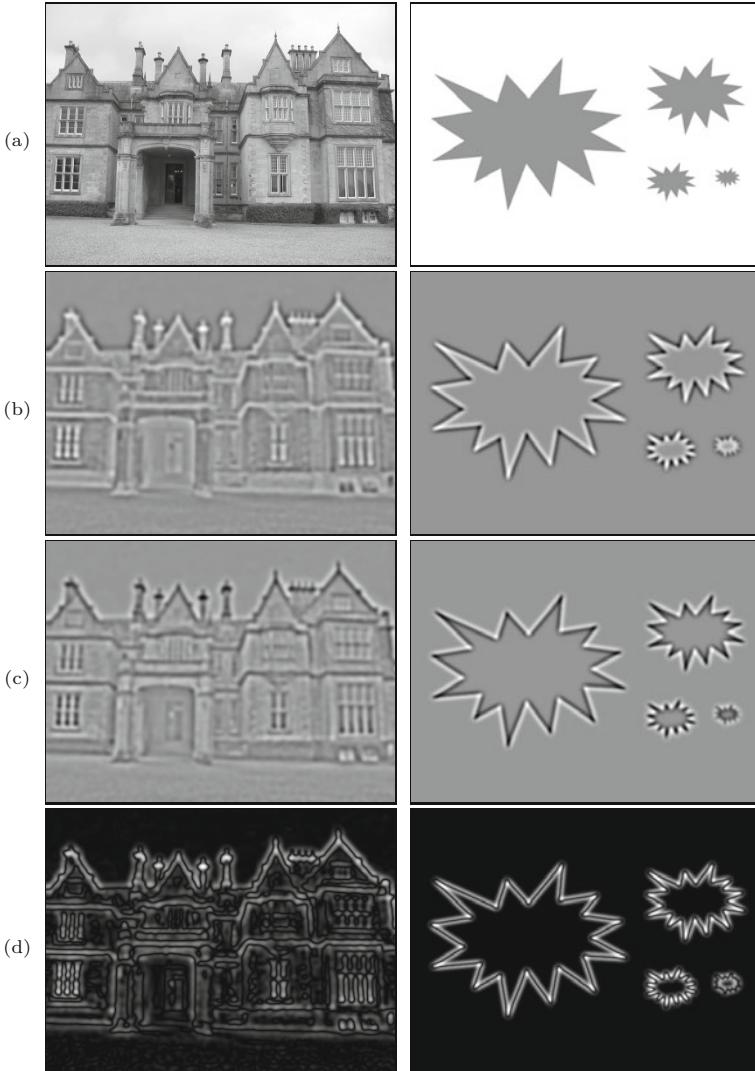
Since the LoG function is based on derivatives, its magnitude strongly depends on the steepness of the Gaussian slope, which is controlled by σ . To obtain responses of comparable magnitude over multiple scales, a *scale normalized* LoG kernel can be defined in the form [151]

$$\hat{L}_\sigma(x, y) = \sigma^2 \cdot (\nabla^2 G_\sigma)(x, y) = \sigma^2 \cdot L_\sigma(x, y) \quad (25.9)$$

$$= \frac{1}{\pi \sigma^2} \cdot \left(\frac{x^2 + y^2 - 2\sigma^2}{2\sigma^2} \right) \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}. \quad (25.10)$$

⁴ To produce a sufficiently accurate discrete LoG filter kernel, the support radius should be set to at least 4σ (kernel diameter $\geq 8\sigma$).

⁵ The LoG is often used as a model for early processes in biological vision systems [161], particularly to describe the center-surround response of receptive fields. In this model, an “on-center” cell is *stimulated* when the center of its receptive field is exposed to light, and is *inhibited* when light falls on its surround. Conversely, an “off-center” cell is stimulated by light falling on its surround. Thus filtering with the original LoG L_σ (Eqn. (25.6)) corresponds to the behavior of off-center cells, while the response to the negative LoG kernel $-L_\sigma$ is that of an on-center cell.



25.1 INTEREST POINTS AT MULTIPLE SCALES

Fig. 25.3

Filtering with the LoG kernel (with $\sigma = 3$). Original images (a). A linear filter with the LoG kernel $L_\sigma(x, y)$ responds strongest to dark spots in a bright surround (b), while the inverted kernel $-L_\sigma(x, y)$ responds strongest to bright spots in a dark surround (c). In (b, c), zero values are shown as medium gray, negative values are dark, positive values are bright. The absolute value of (b) or (c) combines the responses from both dark and bright spots (d).

Note that the integral of this function,

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} |\hat{L}_\sigma(x, y)| dx dy = \frac{4}{e}, \quad (25.11)$$

is constant and thus (unlike Eqn. (25.7)) independent of the scale parameter σ (see Fig. 25.4).

Approximating the LoG by the difference of two Gaussians (DoG)

Although the LoG is “quasi-separable” [113, 243] and can thus be calculated efficiently, the most common method for implementing the LoG filter is to approximate it by the *difference of two Gaussians* (DoG) of widths σ and $\kappa\sigma$, respectively, that is,

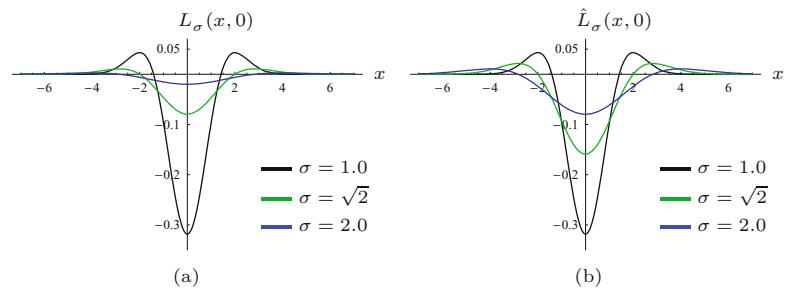
$$L_\sigma(x, y) \approx \lambda \cdot \underbrace{[G_{\kappa\sigma}(x, y) - G_\sigma(x, y)]}_{= D_{\sigma, \kappa}(x, y)} \quad (25.12)$$

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.4

Normalization of the LoG function. Cross section of LoG function $L_\sigma(x, y)$ as defined in Eqn. (25.6) (a); scale-normalized LoG (b) as defined in Eqn. (25.10).

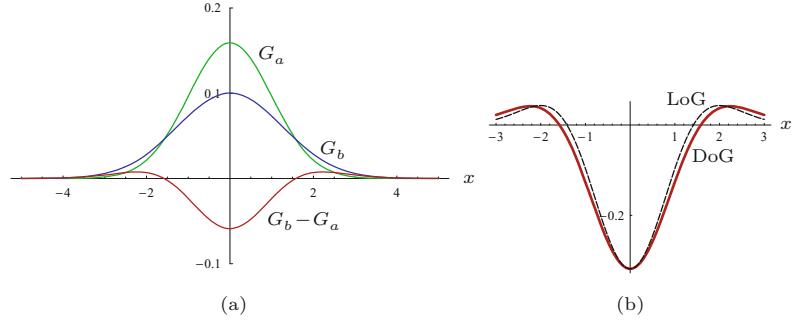
$\sigma = 1.0$ (black), $\sigma = \sqrt{2}$ (green), $\sigma = 2.0$ (blue). All three functions in (b) have the same absolute value integral that is independent of σ (see Eqn. (25.11)).



with the parameter $\kappa > 1$ specifying the relative width of the two Gaussians (defined in Eqn. (25.5)). Properly scaled (by some factor λ , see Eqn. (25.13)), the DoG function $D_{\sigma, \kappa}(x, y)$ approximates the LoG function $L_\sigma(x, y)$ in Eqn. (25.6) with arbitrary precision, as κ approaches 1 ($\kappa = 1$ being excluded, of course). In practice, values of κ in the range $1.1, \dots, 1.3$ yield sufficiently accurate results. As an example, Fig. 25.5 shows the cross-section of the 2D DoG function for $\kappa = 2^{1/3} \approx 1.25992$.⁶

Fig. 25.5

Approximating the LoG by the DoG. The two original Gaussians, $G_a(x)$ with $\sigma_a = 1.0$ and $G_b(x)$ with $\sigma_b = \sigma_a \cdot \kappa = \kappa = 2^{1/3}$, shown by the green and blue curves, respectively (a). The red curve in (a) shows the DoG function $D_{\sigma, \kappa}(x, y) = G_b(x, y) - G_a(x, y)$ for $y = 0$. In (b), the dashed line shows the reference LoG function in comparison to the DoG (red). The DoG is scaled to match the magnitude of the LoG function.



The factor $\lambda \in \mathbb{R}$ in Eqn. (25.12) controls the magnitude of the DoG function; it depends on both the ratio κ and the scale parameter σ . To match the magnitude of the original LoG (Eqn. (25.6)) at the origin, it must be set to

$$\lambda = \frac{2\kappa^2}{\sigma^2 \cdot (\kappa^2 - 1)}. \quad (25.13)$$

Similarly, the *scale-normalized* LoG \hat{L}_σ (Eqn. (25.10)) can be approximated by the DoG function $D_{\sigma, \kappa}$ (Eqn. (25.12)) as

$$\begin{aligned} \hat{L}_\sigma(x, y) &= \sigma^2 L_\sigma(x, y) \\ &\approx \underbrace{\sigma^2 \cdot \lambda}_{\hat{\lambda}} \cdot D_{\sigma, \kappa}(x, y) = \frac{2\kappa^2}{\kappa^2 - 1} \cdot D_{\sigma, \kappa}(x, y), \end{aligned} \quad (25.14)$$

⁶ The factor $\kappa = 2^{1/3}$ originates from splitting the scale interval 2 (i.e., one scale octave) into 3 equal intervals, as described later on. Another factor mentioned frequently in the literature is 1.6, which, however, does not yield a satisfactory approximation. Possibly that value refers to the ratio of the variances σ_2^2/σ_1^2 and not the ratio of the standard deviations σ_2/σ_1 .

$\mathcal{G}(x, y, \sigma)$	continuous Gaussian scale space
$\mathbf{G} = (\mathbf{G}_0, \dots, \mathbf{G}_{K-1})$	discrete Gaussian scale space with K levels
\mathbf{G}_k	single level in a discrete Gaussian scale space
$\mathbf{L} = (\mathbf{L}_0, \dots, \mathbf{L}_{K-1})$	discrete LoG scale space with K levels
\mathbf{L}_k	single level in a LoG scale space
$\mathbf{D} = (\mathbf{D}_0, \dots, \mathbf{D}_{P-1})$	discrete DoG scale space with P octaves
\mathbf{D}_k	single level in a DoG scale space
$\mathbf{G} = (\mathbf{G}_0, \dots, \mathbf{G}_{P-1})$	hierarchical Gaussian scale space with P octaves
$\mathbf{G}_p = (\mathbf{G}_{p,0}, \dots, \mathbf{G}_{p,Q-1})$	octave in a hier. Gaussian scale space with Q levels
$\mathbf{G}_{p,q}$	single level in a hierarchical Gaussian scale space
$\mathbf{D} = (\mathbf{D}_0, \dots, \mathbf{D}_{P-1})$	hierarchical DoG scale space with P octaves
$\mathbf{D}_p = (\mathbf{D}_{p,0}, \dots, \mathbf{D}_{p,Q-1})$	octave in a hierarchical DoG scale space with Q levels
$\mathbf{D}_{p,q}$	single level in a hierarchical DoG scale space
$\mathbf{N}_c(i, j, k)$	$3 \times 3 \times 3$ neighborhood in DoG scale space
$\mathbf{k} = (p, q, u, v)$	discrete key point position in hierarchical scale space ($p, q, u, v \in \mathbb{Z}$)
$\mathbf{k}' = (p, q, x, y)$	continuous (refined) key point position ($x, y \in \mathbb{R}$)

25.1 INTEREST POINTS AT MULTIPLE SCALES

Table 25.1

Scale space-related symbols used in this chapter.

with the factor $\hat{\lambda} = \sigma^2 \cdot \lambda = 2\kappa^2/(\kappa^2 - 1)$ being constant and therefore independent of the scale σ . Thus, as pointed out in [153], with a fixed scale increment κ , the DoG already approximates the scale-normalized LoG up to a constant factor, and thus no additional scaling is required to compare the magnitudes of the DoG responses obtained at different scales.⁷

In the SIFT approach, the DoG is used as an approximation of the (scale-normalized) LoG filter at multiple scales, based on a Gaussian scale space representation of the input image that is described next.⁸

25.1.2 Gaussian Scale Space

The concept of scale space [150] is motivated by the observation that real-world scenes exhibit relevant image features over a large range of sizes and, depending on the particular viewing situation, at various different scales. To relate image structures at different and unknown sizes, it is useful to represent the images simultaneously at different scale levels. The scale space representation of an image adds *scale* as a third coordinate (in addition to the two image coordinates). Thus the scale space is a 3D structure, which can be navigated not only along the x/y positions but also across different scale levels.

Continuous Gaussian scale space

The scale-space representation of an image at a particular scale level is obtained by filtering the image with a kernel that is parameterized to the desired scale. Because of its unique properties [11, 71], the most common type of scale space is based on successive filtering with Gaussian kernels. Conceptually, given a continuous, 2D function $F(x, y)$, its Gaussian scale space representation is a 3D function

⁷ See Sec. E.4 in the Appendix for additional details.

⁸ See Table 25.1 for a summary of the most important scale space-related symbols used in this chapter.

$$\mathcal{G}(x, y, \sigma) = (F * H^{G,\sigma})(x, y), \quad (25.15)$$

where $H^{G,\sigma} \equiv G_\sigma(x, y)$ is a 2D Gaussian kernel (see Eqn. (25.5)) with unit integral, and $*$ denotes the linear convolution over x, y . Note that $\sigma \geq 0$ serves as both the continuous scale parameter and the width of the corresponding Gaussian filter kernel.

A fully continuous Gaussian scale space $\mathcal{G}(x, y, \sigma)$ covers a 3D volume and represents the original function $F(x, y)$ at varying scales σ . For $\sigma = 0$, the Gaussian kernel $H^{G,0}$ has zero width, which makes it equivalent to an impulse or Dirac function $\delta(x, y)$.⁹ This is the neutral element of linear convolution, that is,

$$\mathcal{G}(x, y, 0) = (F * H^{G,0})(x, y) = (F * \delta)(x, y) = F(x, y). \quad (25.16)$$

Thus the base level $\mathcal{G}(x, y, 0)$ of the Gaussian scale space is identical to the input function $F(x, y)$. In general (with $\sigma > 0$), the Gaussian kernel $H^{G,\sigma}$ acts as a low-pass filter with a cutoff frequency proportional to $1/\sigma$ (see Sec. E.3 in the Appendix), the maximum frequency (or bandwidth) of the original “signal” $F(x, y)$ being potentially unlimited.

Discrete Gaussian scale space

This is different for a *discrete* input function $I(u, v)$, whose bandwidth is implicitly limited to half the sampling frequency, as mandated by the sampling theorem to avoid aliasing.¹⁰ Thus, in the discrete case, the lowest level $\mathcal{G}(x, y, 0)$ of the Gaussian scale space is not accessible! To model the implicit bandwidth limitations of the sampling process, the discrete input image $I(u, v)$ is assumed to be pre-filtered (with respect to the underlying continuous signal) with a Gaussian kernel of width $\sigma_s \geq 0.5$ [153], that is,

$$\mathcal{G}(u, v, \sigma_s) \equiv I(u, v). \quad (25.17)$$

Thus the discrete input image $I(u, v)$ is implicitly placed at some initial level σ_s of the Gaussian scale space, and the lower levels with $\sigma < \sigma_s$ are not available.

Any higher level $\sigma_h > \sigma_s$ of the Gaussian scale space can be derived from the original image $I(u, v)$ by filtering with Gaussian kernel $H^{G,\bar{\sigma}}$, that is,

$$\mathcal{G}(u, v, \sigma_h) = (I * H^{G,\bar{\sigma}})(u, v), \quad \text{with } \bar{\sigma} = \sqrt{\sigma_h^2 - \sigma_s^2}. \quad (25.18)$$

This is due to the fact that applying two Gaussian filters of widths σ_1 and σ_2 , one after the other, is equivalent to a single convolution with a Gaussian kernel of width $\sigma_{1,2}$, that is,¹¹

$$(I * H^{G,\sigma_1}) * H^{G,\sigma_2} \equiv I * H^{G,\sigma_{1,2}}, \quad (25.19)$$

⁹ See Chapter 5, Sec. 5.3.4.

¹⁰ See Chapter 18, Sec. 18.2.1.

¹¹ See Sec. E.1 in the Appendix for additional details on combining Gaussian filters.

with $\sigma_{1,2} = (\sigma_1^2 + \sigma_2^2)^{1/2}$. We define the *discrete Gaussian scale space* representation of an image I as a vector of M images, one for each scale level m :

$$\mathbf{G} = (G_0, G_1, \dots, G_{M-1}). \quad (25.20)$$

Associated with each level G_m is its absolute scale $\sigma_m > 0$, and each level G_m represents a blurred version of the original image, that is, $G_m(u, v) \equiv \mathcal{G}(u, v, \sigma_m)$ in the notation introduced in Eqn. (25.15). The scale ratio between adjacent scale levels,

$$\Delta_\sigma = \frac{\sigma_{m+1}}{\sigma_m}, \quad (25.21)$$

is pre-defined and constant. Usually, Δ_σ is specified such that the absolute scale σ_m doubles with a given number of levels Q , called an *octave*. In this case, the resulting scale increment is $\Delta_\sigma = 2^{1/Q}$ with (typically) $Q = 3, \dots, 6$.

In addition, a *base scale* $\sigma_0 > \sigma_s$ is specified for the initial level G_0 , with σ_s denoting the smoothing of the discrete image implied by the sampling process, as discussed already. Based on empirical results, a base scale of $\sigma_0 = 1.6$ is recommended in [153] to achieve reliable interest point detection. Given Q and the base scale σ_0 , the absolute scale at an arbitrary scale space level G_m is

$$\sigma_m = \sigma_0 \cdot \Delta_\sigma^m = \sigma_0 \cdot 2^{m/Q}, \quad (25.22)$$

for $m = 0, \dots, M - 1$.

As follows from Eqn. (25.18), each scale level G_m can be obtained directly from the discrete input image I by a filter operation

$$G_m = I * H^{G, \bar{\sigma}_m}, \quad (25.23)$$

with a Gaussian kernel $H^{G, \bar{\sigma}_m}$ of width

$$\bar{\sigma}_m = \sqrt{\sigma_m^2 - \sigma_s^2} = \sqrt{\sigma_0^2 \cdot 2^{2m/Q} - \sigma_s^2}. \quad (25.24)$$

In particular, the initial scale space level G_0 , (with the specified base scale σ_0) is obtained from the discrete input image I by linear filtering using a Gaussian kernel of width

$$\bar{\sigma}_0 = \sqrt{\sigma_0^2 - \sigma_s^2}. \quad (25.25)$$

Alternatively, using the relation $\sigma_m = \sigma_{m-1} \cdot \Delta_\sigma$ (from Eqn. (25.21)), the scale levels G_1, \dots, G_{M-1} could be calculated recursively from the base level G_0 in the form

$$G_m = G_{m-1} * H^{G, \sigma'_m}, \quad (25.26)$$

for $m > 0$, with a sequence of Gaussian kernels H^{G, σ'_m} of width

$$\sigma'_m = \sqrt{\sigma_m^2 - \sigma_{m-1}^2} = \sigma_0 \cdot 2^{m/Q} \cdot \sqrt{1 - 1/\Delta_\sigma^2}. \quad (25.27)$$

Table 25.2 lists the resulting kernel widths for $Q = 3$ levels per octave and base scale $\sigma_0 = 1.6$ over a scale range of 6 octaves. The

value $\bar{\sigma}_m$ denotes the size of the Gaussian kernel required to compute the image at scale m from the discrete input image I (assumed to be sampled with $\sigma_s = 0.5$). σ'_m is the width of the Gaussian kernel to compute level m recursively from the previous level $m-1$. Apparently (though perhaps unexpectedly), the kernel size required for recursive filtering (σ'_m) grows at the same (exponential) rate as the absolute kernel size $\bar{\sigma}_m$.¹²

Table 25.2

Filter sizes required for calculating Gaussian scale levels G_m for the first 6 octaves. Each octave consists of $Q = 3$ levels, placed at increments of Δ_σ along the scale coordinate.

The discrete input image I is assumed to be pre-filtered with σ_s . Column σ_m denotes the absolute scale at level m , starting with the specified base offset scale σ_0 . $\bar{\sigma}_m$ is the width of the Gaussian filter required to calculate level G_m directly from the input image I . Values σ'_m are the widths of the Gaussian kernels required to calculate level G_m from the previous level G_{m-1} . Note that the width of the Gaussian kernels needed for recursive filtering (σ'_m) grows at the same exponential rate as the size of the direct filter ($\bar{\sigma}_m$).

m	σ_m	$\bar{\sigma}_m$	σ'_m
18	102.4000	102.3988	62.2908
17	81.2749	81.2734	49.4402
16	64.5080	64.5060	39.2408
15	51.2000	51.1976	31.1454
14	40.6375	40.6344	24.7201
13	32.2540	32.2501	19.6204
12	25.6000	25.5951	15.5727
11	20.3187	20.3126	12.3601
10	16.1270	16.1192	9.8102
9	12.8000	12.7902	7.7864
8	10.1594	10.1471	6.1800
7	8.0635	8.0480	4.9051
6	6.4000	6.3804	3.8932
5	5.0797	5.0550	3.0900
4	4.0317	4.0006	2.4525
3	3.2000	3.1607	1.9466
2	2.5398	2.4901	1.5450
1	2.0159	1.9529	1.2263
0	1.6000	1.5199	—

$m \dots$ linear scale index

$\sigma_m \dots$ absolute scale at level m
(Eqn. (25.22))

$\bar{\sigma}_m \dots$ relative scale at level m
w.r.t. the original image
(Eqn. (25.24))

$\sigma'_m \dots$ relative scale at level m
w.r.t. the previous level
 $m-1$ (Eqn. (25.27))

$\sigma_s = 0.5$ (sampling scale)

$\sigma_0 = 1.6$ (base scale)

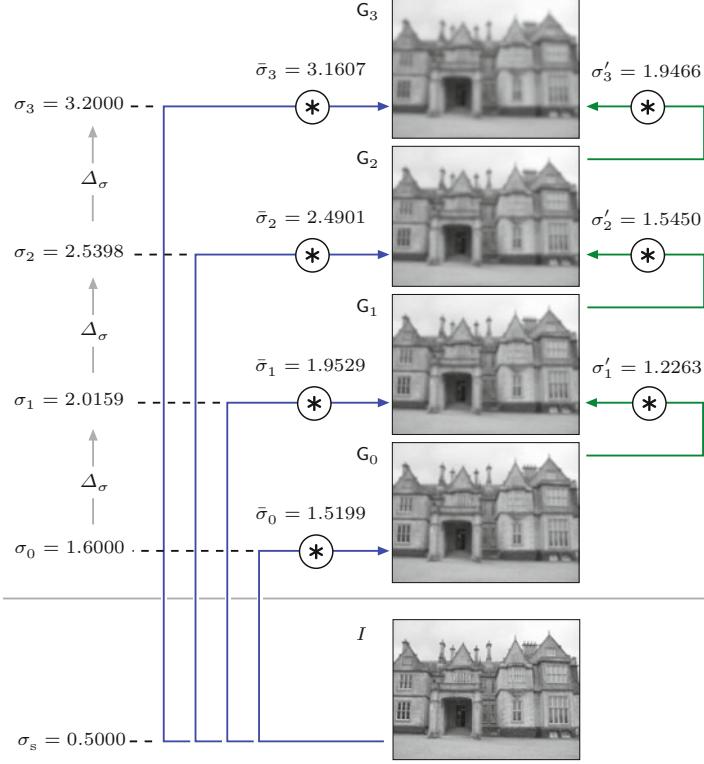
$Q = 3$ (levels per octave)

$\Delta_\sigma = 2^{1/Q} \approx 1.256$

At scale level $m = 16$ and absolute scale $\sigma_{16} = 1.6 \cdot 2^{16/3} \approx 64.5$, for example, the Gaussian filters required to compute G_{16} directly from the input image I has the width $\bar{\sigma}_{16} = (\sigma_{16}^2 - \sigma_s^2)^{1/2} = (64.5080^2 - 0.5^2)^{1/2} \approx 64.5$, while the filter to blur incrementally from the previous scale level has the width $\sigma'_{16} = (\sigma_{16}^2 - \sigma_{15}^2)^{1/2} = (64.5080^2 - 51.1976^2)^{1/2} \approx 39.2$. Since recursive filtering also tends to accrue numerical inaccuracies, this approach does not offer a significant advantage in general. Fortunately, the growth of the Gaussian kernels can be kept small by spatially sub-sampling after each octave, as will be described in Sec. 25.1.4.

The process of constructing a discrete Gaussian scale space using the same parameters as in Table 25.2 is illustrated in Fig. 25.6. Again the input image I is assumed to be pre-filtered at $\sigma_s = 0.5$ due to sampling and the absolute scale of the first level G_0 is set to $\sigma_0 = 1.6$. The scale ratio between successive levels is fixed at $\Delta_\sigma = 2^{1/3} \approx 1.25992$, that is, each octave spans three discrete scale levels. As shown in this figure, each scale level G_m can be calculated either directly from the input image I by filtering with a Gaussian of width $\bar{\sigma}_m$, or recursively from the previous level by filtering with σ'_m .

¹² The ratio of the kernel sizes $\bar{\sigma}_m / \sigma'_m$ converges to $\sqrt{1 - 1/\Delta_\sigma^2}$ (≈ 1.64 for $Q = 3$) and is thus practically constant for larger values of m .



25.1 INTEREST POINTS AT MULTIPLE SCALES

Fig. 25.6

Gaussian scale space construction (first four levels). Parameters are the same as in Table 25.2. The discrete input image I is assumed to be pre-filtered with a Gaussian of width $\sigma_s = 0.5$; the scale of the initial level (base scale offset) is set to $\sigma_0 = 1.6$. The discrete scale space levels G_0, G_1, \dots (at absolute scales $\sigma_0, \sigma_1, \dots$) are slices through the continuous scale space. Scale levels can either be calculated by filtering directly from the discrete image I with Gaussian kernels of width $\sigma_0, \sigma_1, \dots$ (blue arrows) or, alternatively, by recursively filtering with $\sigma'_1, \sigma'_2, \dots$ (green arrows).

25.1.3 LoG/DoG Scale Space

Interest point detection in the SIFT approach is based on finding local maxima in the output of LoG filters over multiple scales. Analogous to the discrete Gaussian scale space described in Sec. 25.1.2, a LoG scale space representation of an image I can be defined as

$$\mathcal{L} = (\mathcal{L}_0, \mathcal{L}_1, \dots, \mathcal{L}_{M-1}), \quad (25.28)$$

with levels $\mathcal{L}_m = I * H^{\text{L}, \sigma_m}$, where $H^{\text{L}, \sigma_m}(x, y) \equiv \hat{L}_{\sigma_m}(x, y)$ is a scale-normalized LoG kernel of width σ_m (see Eqn. (25.10)).

As demonstrated in Eqn. (25.12), the LoG kernel can be approximated by the difference of two Gaussians whose widths differ by a certain ratio κ . Since pairs of adjacent scale layers in the Gaussian scale space are also separated by a fixed scale ratio, it is straightforward to construct a multi-scale DoG representation,

$$\mathcal{D} = (\mathcal{D}_0, \mathcal{D}_1, \dots, \mathcal{D}_{M-2}) \quad (25.29)$$

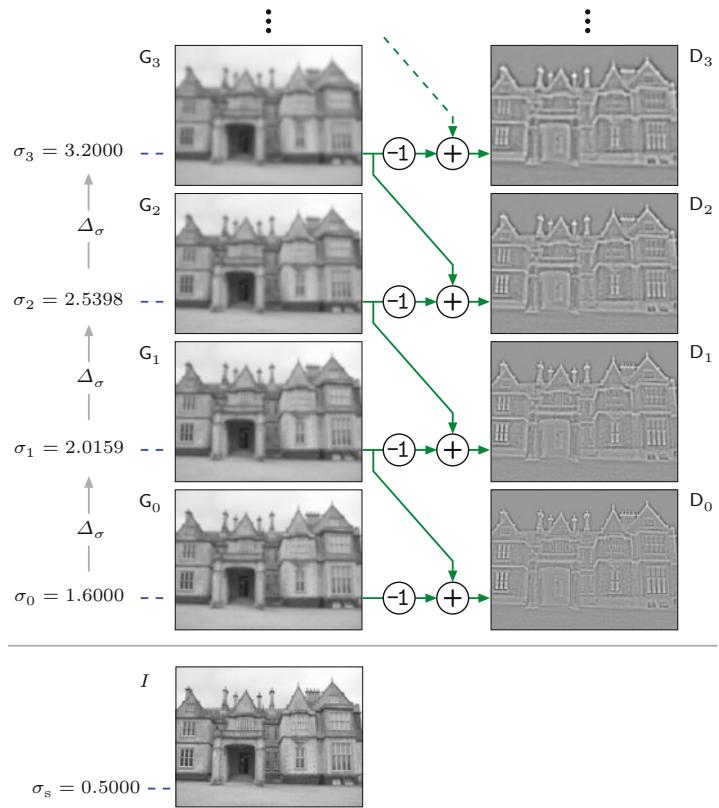
from an existing Gaussian scale space $\mathcal{G} = (G_0, G_1, \dots, G_{M-1})$. The individual levels in the DoG scale space are defined as

$$\mathcal{D}_m = \hat{\lambda} \cdot (G_{m+1} - G_m) \approx \mathcal{L}_m, \quad (25.30)$$

for $m = 0, \dots, M-2$. The constant factor $\hat{\lambda}$ (defined in Eqn. (25.14)) can be omitted in the aforementioned expression, as the relative width of the involved Gaussians,

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.7
DoG scale-space construction. The differences of successive levels G_0, G_1, \dots of the Gaussian scale space (see Fig. 25.6) are used to approximate a LoG scale space. Each DoG-level D_m is calculated as the point-wise difference $G_{m+1} - G_m$ between Gaussian levels G_{m+1} and G_m . The values in D_0, \dots, D_3 are scale-normalized (see Eqn. (25.14)) and mapped to a uniform intensity range for viewing.



$$\kappa = \Delta_\sigma = \frac{\sigma_{m+1}}{\sigma_m} = 2^{1/Q}, \quad (25.31)$$

is simply the fixed scale ratio Δ_σ between successive scale space levels. Note that the DoG approximation does not require any additional normalization to approximate a scale-normalized LoG representation (see Eqns. 25.10 and 25.14). The process of calculating a DoG scale space from a discrete Gaussian scale space is illustrated in Fig. 25.7, using the same parameters as in Table 25.2 and Fig. 25.6.

25.1.4 Hierarchical Scale Space

Despite the fact that 2D Gaussian filter kernels are separable into 1D kernels,¹³ the size of the required filter grows quickly with increasing scale, regardless if a direct or recursive approach is used (as shown in Table 25.2). However, each Gaussian filter operation reduces the bandwidth of the signal inversely proportional to the width of the kernel (see Sec. E.3 in the Appendix). If the image size is kept constant over all scales, the images become increasingly oversampled at higher scale levels. In other words, the sampling rate in a Gaussian scale space can be reduced with increasing scale without losing relevant signal information.

¹³ See also Chapter 5, Sec. 5.3.3.

Octaves and sub-sampling (decimation)

In particular, doubling the scale cuts the bandwidth by half, that is, the signal at scale level 2σ has only half the bandwidth of the signal at level σ . An image signal at scale level 2σ of a Gaussian scale space thus shows only half the bandwidth of the same image at scale level σ . In a Gaussian scale space representation it is thus safe to down-sample the image to half the sample rate after each octave without any loss of information. This suggests a very efficient, “pyramid-like” approach for constructing a DoG scale space, as illustrated in Fig. 25.8.¹⁴

At the start (bottom) of each octave, the image is down-sampled to half the resolution, that is, each pixel in the new octave covers twice the distance of the pixels in the previous octave in every spatial direction. Within each octave, the same small Gaussian kernels can be used for successive filtering, since their relative widths (with respect to the original sampling lattice) also implicitly double at each octave. To describe these relations formally, we use

$$\mathbf{G} = (\mathbf{G}_0, \mathbf{G}_1, \dots, \mathbf{G}_{P-1}) \quad (25.32)$$

to denote a *hierarchical Gaussian scale space* consisting of P octaves. Each octave

$$\mathbf{G}_p = (\mathbf{G}_{p,0}, \mathbf{G}_{p,1}, \dots, \mathbf{G}_{p,Q}), \quad (25.33)$$

consists of $Q+1$ scale levels $\mathbf{G}_{p,q}$, where $p \in [0, P-1]$ is the octave index and $q \in [0, Q]$ is the level index within the containing octave \mathbf{G}_p . With respect to *absolute scale*, a level $\mathbf{G}_{p,q} = \mathbf{G}_p(q)$ in the hierarchical Gaussian scale space corresponds to the level \mathbf{G}_m in the non-hierarchical Gaussian scale space (see Eqn. (25.20)) with index

$$m = Q \cdot p + q. \quad (25.34)$$

As follows from Eqn. (25.22), the *absolute scale* at level $\mathbf{G}_{p,q}$ then is

$$\begin{aligned} \sigma_{p,q} &= \sigma_m = \sigma_0 \cdot \Delta_\sigma^m = \sigma_0 \cdot 2^{m/Q} \\ &= \sigma_0 \cdot 2^{(Qp+q)/Q} = \sigma_0 \cdot 2^{p+q/Q}, \end{aligned} \quad (25.35)$$

where $\sigma_0 = \sigma_{0,0}$ denotes the predefined base scale offset (e.g., $\sigma_0 = 1.6$ in Table 25.2). In particular, the absolute scale of the base level $\mathbf{G}_{p,0}$ of *any* octave \mathbf{G}_p is

$$\sigma_{p,0} = \sigma_0 \cdot 2^p. \quad (25.36)$$

The **decimated scale** $\dot{\sigma}_{p,q}$ is the absolute scale $\sigma_{p,q}$ (Eqn. (25.35)) expressed in the coordinate units of octave \mathbf{G}_p , that is,

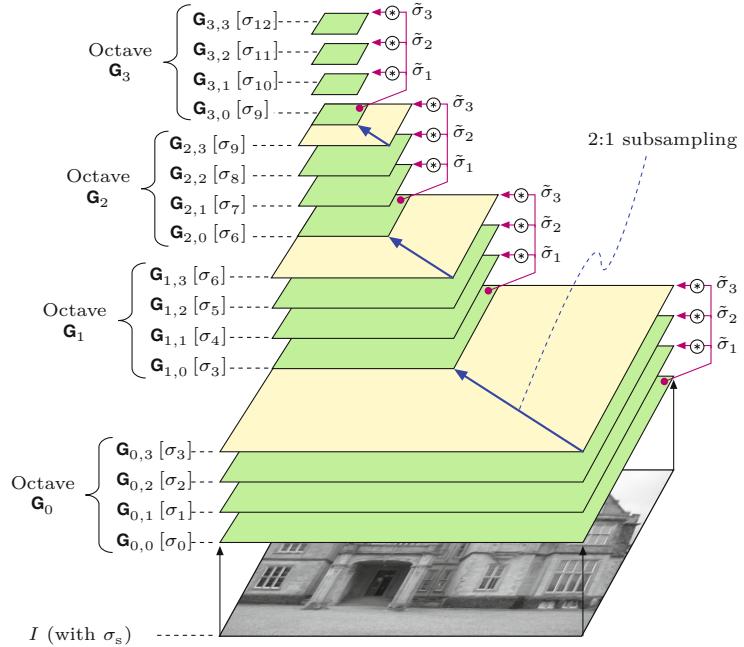
$$\dot{\sigma}_{p,q} = \dot{\sigma}_q = \sigma_{p,q} \cdot 2^{-p} = \sigma_0 \cdot 2^{p+q/Q} \cdot 2^{-p} = \sigma_0 \cdot 2^{q/Q}. \quad (25.37)$$

Note that the decimated scale $\dot{\sigma}_{p,q}$ is independent of the octave index p and therefore $\dot{\sigma}_{p,q} \equiv \dot{\sigma}_q$, for any level index q .

¹⁴ Successive reduction of image resolution by sub-sampling is the core concept of “image pyramid” methods [41].

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.8 Hierarchical Gaussian scale space. Each octave extends over $Q = 3$ scale steps. The base level $\mathbf{G}_{p,0}$ of each octave $p > 0$ is obtained by 2:1 sub-sampling of the top level $\mathbf{G}_{p-1,3}$ of the next-lower octave. At the transition between octaves, the resolution (image size) is cut in half in the x - and y -direction. The absolute scale at octave level $\mathbf{G}_{p,q}$ is σ_m , with $m = Qp + q$. Within each octave, the same set of Gaussian kernels ($\tilde{\sigma}_1$, $\tilde{\sigma}_2$, $\tilde{\sigma}_3$) is used to calculate the following levels from the octave's base level $\mathbf{G}_{p,0}$.



From the octave's base level $\mathbf{G}_{p,0}$, the subsequent levels in the same octave can be calculated by filtering with relatively small Gaussian kernels. The size of the kernel needed to calculate scale-level $\mathbf{G}_{p,q}$ from the octave's base level $\mathbf{G}_{p,0}$ is obtained from the corresponding decimated scales (Eqn. (25.37)) as

$$\tilde{\sigma}_{p,q} = \sqrt{\dot{\sigma}_{p,q}^2 - \dot{\sigma}_{p,0}^2} = \sqrt{(\sigma_0 \cdot 2^{q/Q})^2 - \sigma_0^2} = \sigma_0 \cdot \sqrt{2^{2q/Q} - 1}, \quad (25.38)$$

for $q \geq 0$. Note that $\tilde{\sigma}_q$ is independent of the octave index p and thus the *same* filter kernels can be used at each octave. For example, with $Q = 3$ and $\sigma_0 = 1.6$ (as used in Table 25.2) the resulting kernel widths are

$$\tilde{\sigma}_1 = 1.2263, \quad \tilde{\sigma}_2 = 1.9725, \quad \tilde{\sigma}_3 = 2.7713. \quad (25.39)$$

Also note that, instead of filtering all scale levels $\mathbf{G}_{p,q}$ in an octave from the corresponding base level $\mathbf{G}_{p,0}$, we could calculate them recursively from the next-lower level $\mathbf{G}_{p,q-1}$. While this approach requires even smaller Gaussian kernels (and is thus more efficient), recursive filtering tends to accrue numerical inaccuracies. Nevertheless, the method is used frequently in scale-space implementations.

Decimation between successive octaves

With $M \times N$ being the size of the original image I , every sub-sampling step between octaves cuts the size of the image by half, that is,

$$M_{p+1} \times N_{p+1} = \left\lfloor \frac{M_p}{2} \right\rfloor \times \left\lfloor \frac{N_p}{2} \right\rfloor, \quad (25.40)$$

for octaves with index $p \geq 0$. The resulting image size at octave \mathbf{G}_p is thus

$$M_p \times N_p = \left\lfloor \frac{M_0}{2^p} \right\rfloor \times \left\lfloor \frac{N_0}{2^p} \right\rfloor. \quad (25.41)$$

The base level $\mathbf{G}_{p,0}$ of each octave \mathbf{G}_p (with $p > 0$) is obtained by sub-sampling the top level $\mathbf{G}_{p-1,Q}$ of the next-lower octave \mathbf{G}_{p-1} as

$$\mathbf{G}_{p,0} = \text{Decimate}(\mathbf{G}_{p-1,Q}), \quad (25.42)$$

where $\text{Decimate}(G)$ denotes the 2:1 sub-sampling operation, that is,

$$\mathbf{G}_{p,0}(u, v) \leftarrow \mathbf{G}_{p-1,Q}(2u, 2v), \quad (25.43)$$

for each sample position $(u, v) \in [0, M_p - 1] \times [0, N_p - 1]$. Additional low-pass filtering is not required prior to sub-sampling since the Gaussian smoothing performed in each octave also cuts the bandwidth by half.

The main steps involved in constructing a hierarchical Gaussian scale space are summarized in Alg. 25.1. In summary, the input image I is first blurred to scale σ_0 by filtering with a Gaussian kernel of width $\bar{\sigma}_0$. Within each octave \mathbf{G}_p , the scale levels $\mathbf{G}_{p,q}$ are calculated from the base level $\mathbf{G}_{p,0}$ by filtering with a set of Gaussian filters of width $\tilde{\sigma}_q$ ($q = 1, \dots, Q$). Note that the values $\tilde{\sigma}_q$ and the corresponding Gaussian kernels $H^{G, \tilde{\sigma}_q}$ can be pre-calculated once since they are independent of the octave index p (Alg. 25.1, lines 13–14). The base level $\mathbf{G}_{p,0}$ of each higher octave \mathbf{G}_p is obtained by decimating the top level $\mathbf{G}_{p-1,Q}$ of the previous octave \mathbf{G}_{p-1} . Typical parameter values are $\sigma_s = 0.5$, $\sigma_0 = 1.6$, $Q = 3$, $P = 4$.

Spatial positions in the hierarchical scale space

To properly associate the spatial positions of features detected in different octaves of the hierarchical scale space we define the function

$$\mathbf{x}_0 \leftarrow \text{AbsPos}(\mathbf{x}_p, p),$$

that maps the continuous position $\mathbf{x}_p = (x_p, y_p)$ in the local coordinate system of octave p to the corresponding position $\mathbf{x} = (x, y)$ in the coordinate system of the original full-resolution image I (octave $p = 0$). The function AbsPos can be defined recursively by relating the positions in successive octaves as

$$\text{AbsPos}(\mathbf{x}_p, p) = \begin{cases} \mathbf{x}_p & \text{for } p = 0, \\ \text{AbsPos}(2 \cdot \mathbf{x}_p, p-1) & \text{for } p > 0, \end{cases} \quad (25.44)$$

which gives $\mathbf{x}_0 = \text{AbsPos}(2^p \cdot \mathbf{x}_p, 0)$ and thus

$$\text{AbsPos}(\mathbf{x}_p, p) = 2^p \cdot \mathbf{x}_p. \quad (25.45)$$

Hierarchical LoG/DoG scale space

Analogous to the scheme shown in Fig. 25.7, a *hierarchical* DoG scale space representation is obtained by calculating the difference of adjacent scale levels within each octave of the hierarchical Gaussian scale space, that is,

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Alg. 25.1

Building a hierarchical Gaussian scale space. The input image I is first blurred to scale σ_0 by filtering with a Gaussian kernel of width $\bar{\sigma}_0$ (line 3). In each octave \mathbf{G}_p , the scale levels $\mathbf{G}_{p,q}$ are calculated from the base level $\mathbf{G}_{p,0}$ by filtering with a set of Gaussian filters of width $\tilde{\sigma}_1, \dots, \tilde{\sigma}_Q$ (line 13–14). The base level $\mathbf{G}_{p,0}$ of each higher octave is obtained by sub-sampling the top level $\mathbf{G}_{p-1,Q}$ of the previous octave (line 6).

```

1: BuildGaussianScaleSpace( $I, \sigma_s, \sigma_0, P, Q$ )
   Input:  $I$ , source image;  $\sigma_s$ , sampling scale;  $\sigma_0$ , reference scale
          of the first octave;  $P$ , number of octaves.  $Q$ , number of scale
          steps per octave. Returns a hierarchical Gaussian scale space
          representation  $\mathbf{G}$  of the image  $I$ .
2:  $\bar{\sigma}_0 \leftarrow (\sigma_0^2 - \sigma_s^2)^{1/2}$             $\triangleright$  scale to base of 1st octave, Eq. 25.25
3:  $\mathbf{G}_{\text{init}} \leftarrow I * H^{G, \bar{\sigma}_0}$             $\triangleright$  apply 2D Gaussian filter of width  $\bar{\sigma}_0$ 
4:  $\mathbf{G}_0 \leftarrow \text{MakeGaussianOctave}(\mathbf{G}_{\text{init}}, 0, Q, \sigma_0)$      $\triangleright$  create octave  $\mathbf{G}_0$ 
5: for  $p \leftarrow 1, \dots, P-1$  do                   $\triangleright$  octave index  $p$ 
6:    $\mathbf{G}_{\text{next}} \leftarrow \text{Decimate}(\mathbf{G}_{p-1,Q})$      $\triangleright$  dec. top level of octave  $p-1$ 
7:    $\mathbf{G}_p \leftarrow \text{MakeGaussianOctave}(\mathbf{G}_{\text{next}}, p, Q, \sigma_0)$      $\triangleright$  create octave
      $\mathbf{G}_p$ 
8:    $\mathbf{G} \leftarrow (\mathbf{G}_0, \dots, \mathbf{G}_{P-1})$ 
9: return  $\mathbf{G}$                                  $\triangleright$  hierarchical Gaussian scale space  $\mathbf{G}$ 

10: MakeGaussianOctave( $\mathbf{G}_{\text{base}}, p, Q, \sigma_0$ )
    Input:  $\mathbf{G}_{\text{base}}$ , octave base level;  $p$ , octave index;  $Q$ , number of
          levels per octave;  $\sigma_0$ , reference scale.
11:  $\mathbf{G}_{p,0} \leftarrow \mathbf{G}_{\text{base}}$ 
12: for  $q \leftarrow 1, \dots, Q$  do                   $\triangleright$  level index  $q$ 
13:    $\tilde{\sigma}_q \leftarrow \sigma_0 \cdot \sqrt{2^{2q/Q} - 1}$        $\triangleright$  see Eq. 25.38
14:    $\mathbf{G}_{p,q} \leftarrow \mathbf{G}_{\text{base}} * H^{G, \tilde{\sigma}_q}$      $\triangleright$  apply 2D Gaussian filter of width  $\tilde{\sigma}_q$ 
15:  $\mathbf{G}_p \leftarrow (\mathbf{G}_{p,0}, \dots, \mathbf{G}_{p,Q})$ 
16: return  $\mathbf{G}_p$                                  $\triangleright$  scale space octave  $\mathbf{G}_p$ 

17: Decimate( $\mathbf{G}_{\text{in}}$ )
    Input:  $\mathbf{G}_{\text{in}}$ , Gaussian scale space level.
18:  $(M, N) \leftarrow \text{Size}(\mathbf{G}_{\text{in}})$ 
19:  $M' \leftarrow \lfloor \frac{M}{2} \rfloor, \quad N' \leftarrow \lfloor \frac{N}{2} \rfloor$        $\triangleright$  decimated size
20: Create map  $\mathbf{G}_{\text{out}}: M' \times N' \mapsto \mathbb{R}$ 
21: for all  $(u, v) \in M' \times N'$  do
22:    $\mathbf{G}_{\text{out}}(u, v) \leftarrow \mathbf{G}_{\text{in}}(2u, 2v)$            $\triangleright$  2:1 subsampling
23: return  $\mathbf{G}_{\text{out}}$                                  $\triangleright$  decimated scale level  $\mathbf{G}_{\text{out}}$ 

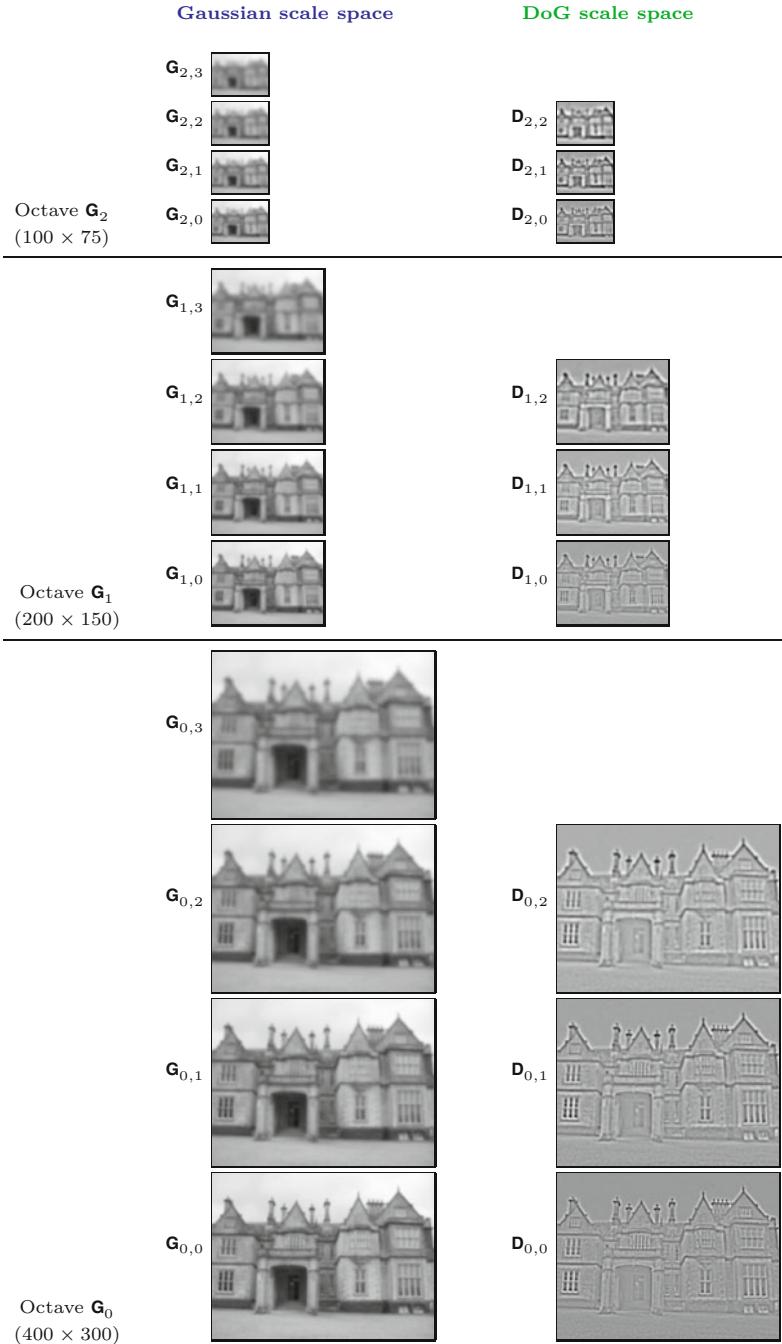
```

$$\mathbf{D}_{p,q} = \mathbf{G}_{p,q+1} - \mathbf{G}_{p,q} \quad (25.46)$$

for level numbers $q \in [0, Q-1]$. Figure 25.9 shows the corresponding Gaussian and DoG scale levels for the previous example over a range of three octaves. To demonstrate the effects of sub-sampling, the same information is shown in Fig. 25.10 and 25.11, with all level images scaled to the same size. Figure 25.11 also shows the absolute values of the DoG response, which are effectively used for detecting interest points at different scale levels. Note how blob-like features stand out and disappear again as the scale varies from fine to coarse. Analogous results obtained from a different image are shown in Figs. 25.12 and 25.13.

25.1.5 Scale Space Structure in SIFT

In the SIFT approach, the absolute value of the DoG response is used to localize interest points at different scales. For this purpose, local maxima are detected in the 3D space spanned by the spatial x/y -positions and the scale coordinate. To determine local maxima along the scale dimension over a full octave, two additional DoG levels,



25.1 INTEREST POINTS AT MULTIPLE SCALES

Fig. 25.9

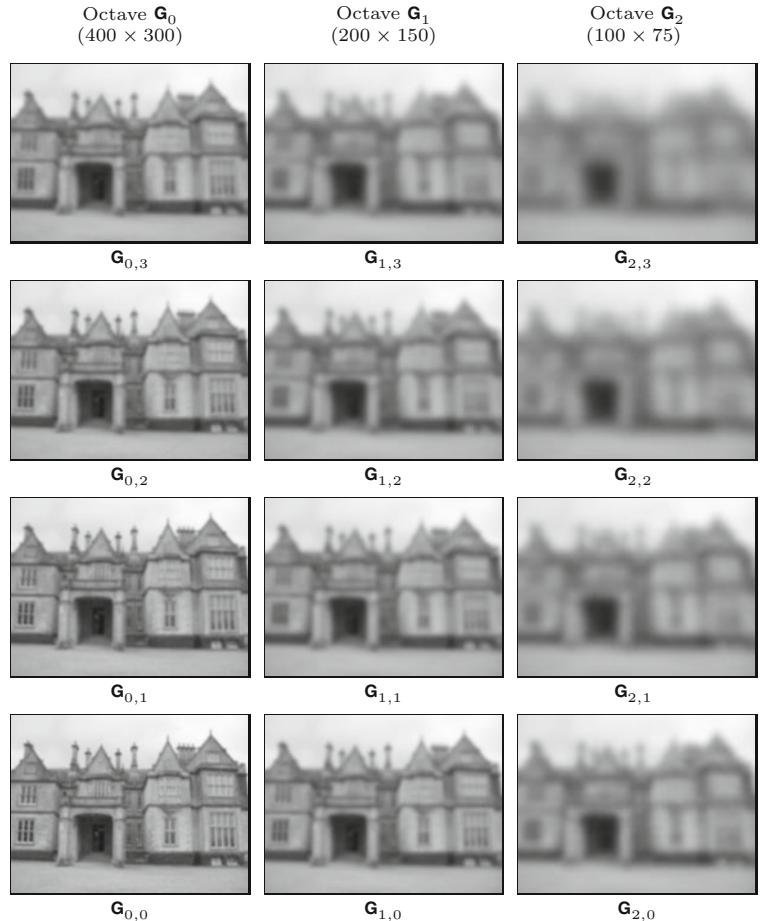
Hierarchical Gaussian and DoG scale space example, with $P = Q = 3$. Gaussian scale space levels $\mathbf{G}_{p,q}$ are shown in the left column, DoG levels $\mathbf{D}_{p,q}$ in the right column. All images are shown at their real scale.

$\mathbf{D}_{p,-1}$ and $\mathbf{D}_{p,Q}$, and two additional Gaussian scale levels, $\mathbf{G}_{p,-1}$ and $\mathbf{G}_{p,Q+1}$, are required in each octave.

In total, each octave \mathbf{G}_p then consists of $Q+3$ Gaussian scale levels $\mathbf{G}_{p,q}$ ($q = -1, \dots, Q+1$) and $Q+2$ DoG levels $\mathbf{D}_{p,q}$ ($q = -1, \dots, Q$), as shown in Fig. 25.14. For the base level $\mathbf{G}_{0,-1}$, the scale index is $m = -1$ and its absolute scale (see Eqns. (25.22) and (25.35)) is

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.10
Hierarchical Gaussian scale space example (castle image). All images are scaled to the same size. Note that $\mathbf{G}_{1,0}$ is merely a sub-sampled copy of $\mathbf{G}_{0,3}$; analogously, $\mathbf{G}_{2,0}$ is sub-sampled from $\mathbf{G}_{1,3}$.



$$\sigma_{0,-1} = \sigma_0 \cdot 2^{-1/Q} = \sigma_0 \cdot \frac{1}{\Delta_\sigma}. \quad (25.47)$$

Thus, with the usual settings ($\sigma_0 = 1.6$ and $Q = 3$), the *absolute* scale values for the six levels of the first octave are

$$\begin{aligned} \sigma_{0,-1} &= 1.2699, & \sigma_{0,0} &= 1.6000, & \sigma_{0,1} &= 2.0159, \\ \sigma_{0,2} &= 2.5398, & \sigma_{0,3} &= 3.2000, & \sigma_{0,4} &= 4.0317. \end{aligned} \quad (25.48)$$

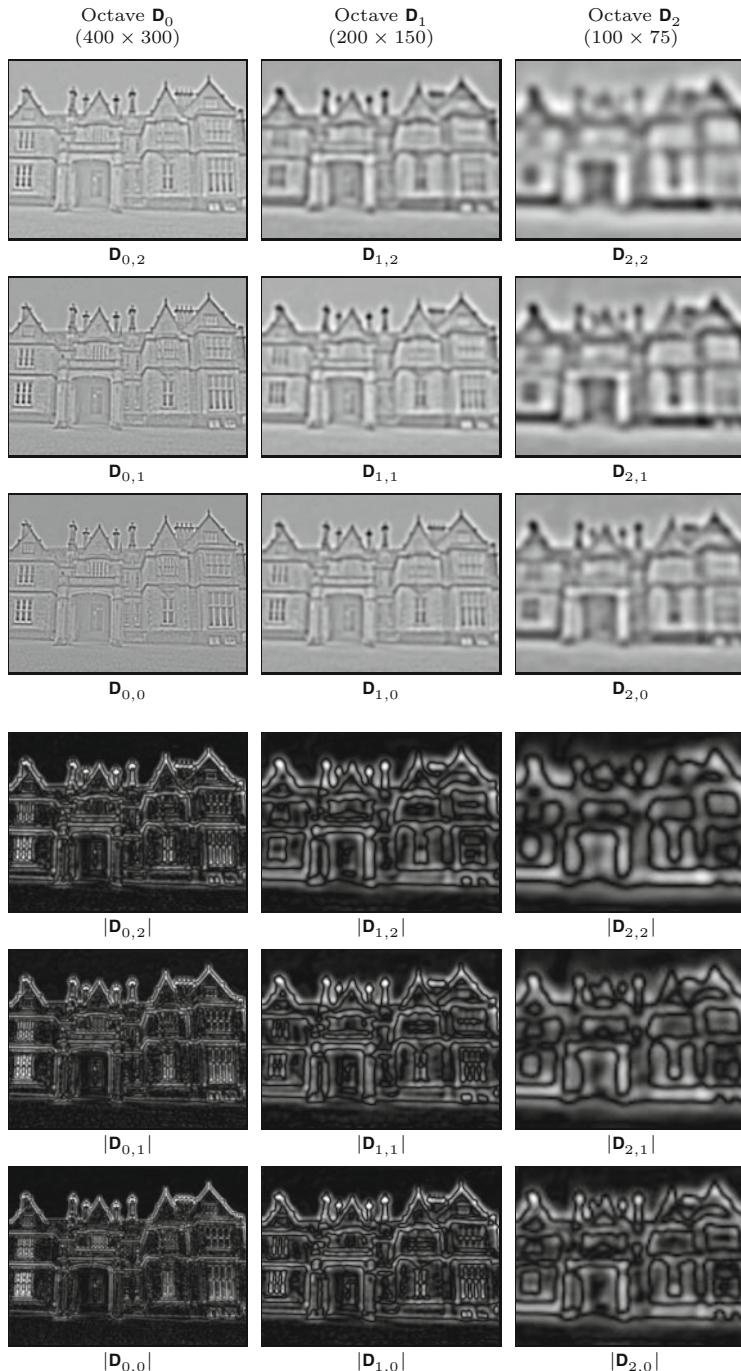
The complete set of scale values for a SIFT scale space with four octaves ($p = 0, \dots, 3$) is listed in [Table 25.3](#).

To construct the Gaussian part of the first scale space octave \mathbf{G}_0 , the initial level $\mathbf{G}_{0,-1}$ is obtained by filtering the input image I with a Gaussian kernel of width

$$\bar{\sigma}_{0,-1} = \sqrt{\sigma_{0,-1}^2 - \sigma_s^2} = \sqrt{1.2699^2 - 0.5^2} \approx 1.1673 \quad (25.49)$$

For the higher octaves ($p > 0$), the initial level ($q = -1$) is obtained by sub-sampling (decimating) level $Q - 1$ of the next-lower octave \mathbf{G}_{p-1} , that is,

$$\mathbf{G}_{p,-1} \leftarrow \text{Decimate}(\mathbf{G}_{p-1,Q-1}), \quad (25.50)$$



25.1 INTEREST POINTS AT MULTIPLE SCALES

Fig. 25.11

Hierarchical DoG scale space example (castle image). The three top rows show the positive and negative DoG values (zero is mapped to intermediate gray). The three bottom rows show the absolute values of the DoG results (zero is mapped to black, maximum values to white). All images are scaled to the size of the original image.

analogous to Eqn. (25.42). The remaining levels $\mathbf{G}_{p,0}, \dots, \mathbf{G}_{p,Q+1}$ of the octave are either calculated by incremental filtering (as described in Fig. 25.6) or by filtering from the octave's initial level $\mathbf{G}_{p,-1}$ with a Gaussian of width $\tilde{\sigma}_{p,q}$ (see Eqn. (25.38)). The advantage of the direct approach is that numerical errors do not accrue across the scale space; the disadvantage is that the kernels are up to 50 % larger than those needed for the incremental approach ($\tilde{\sigma}_{0,4} = 3.8265$ vs.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.12
Hierarchical Gaussian scale space example (**stars** image).

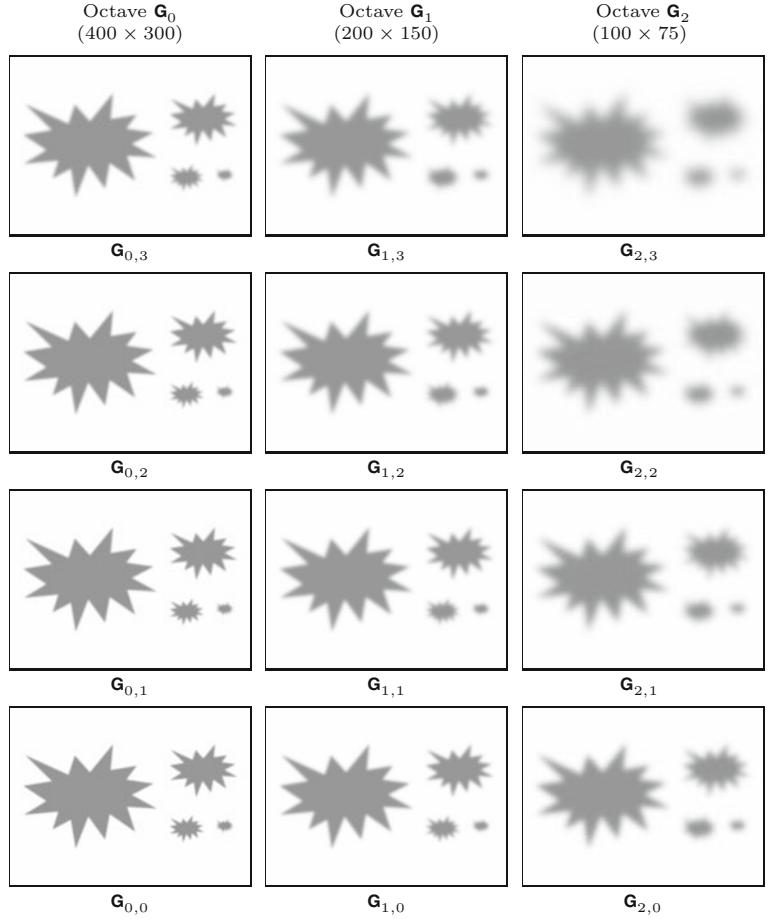


Table 25.3

Absolute and relative scale values for a SIFT scale space with four octaves. Each octave with index $p = 0, \dots, 3$ consists of 6 Gaussian scale layers $\mathbf{G}_{p,q}$, with $q = -1, \dots, 4$. For each scale layer, m is the scale index and $\sigma_{p,q}$ is the corresponding *absolute* scale. Within each octave p , $\tilde{\sigma}_{p,q}$ denotes the *relative* scale with respect to the octave's base layer $\mathbf{G}_{p,-1}$. Each base layer $\mathbf{G}_{p,-1}$ is obtained by sub-sampling (decimating) layer $q = Q - 1 = 2$ in the previous octave, i.e., $\mathbf{G}_{p,-1} = \text{Decimate}(\mathbf{G}_{p-1,Q-1})$, for $p > 0$. The base layer $\mathbf{G}_{0,-1}$ in the bottom octave is derived by Gaussian smoothing of the original image. Note that the relative scale values $\tilde{\sigma}_{p,q} = \tilde{\sigma}_q$ are the same inside every octave (independent of p) and thus the same Gaussian filter kernels can be used for calculating all octaves.

p	q	m	d	$\sigma_{p,q}$	$\dot{\sigma}_q$	$\tilde{\sigma}_q$
3	4	13	8	32.2540	4.0317	3.8265
3	3	12	8	25.6000	3.2000	2.9372
3	2	11	8	20.3187	2.5398	2.1996
3	1	10	8	16.1270	2.0159	1.5656
3	0	9	8	12.8000	1.6000	0.9733
3	-1	8	8	10.1594	1.2699	0.0000
2	4	10	4	16.1270	4.0317	3.8265
2	3	9	4	12.8000	3.2000	2.9372
2	2	8	4	10.1594	2.5398	2.1996
2	1	7	4	8.0635	2.0159	1.5656
2	0	6	4	6.4000	1.6000	0.9733
2	-1	5	4	5.0797	1.2699	0.0000
1	4	7	2	8.0635	4.0317	3.8265
1	3	6	2	6.4000	3.2000	2.9372
1	2	5	2	5.0797	2.5398	2.1996
1	1	4	2	4.0317	2.0159	1.5656
1	0	3	2	3.2000	1.6000	0.9733
1	-1	2	2	2.5398	1.2699	0.0000
0	4	4	1	4.0317	4.0317	3.8265
0	3	3	1	3.2000	3.2000	2.9372
0	2	2	1	2.5398	2.5398	2.1996
0	1	1	1	2.0159	2.0159	1.5656
0	0	0	1	1.6000	1.6000	0.9733
0	-1	-1	1	1.2699	1.2699	0.0000

p ... octave index

q ... level index

m ... linear scale index ($m = Qp + q$)

d ... decimation factor ($d = 2^p$)

$\sigma_{p,q}$... absolute scale (Eqn. (25.35))

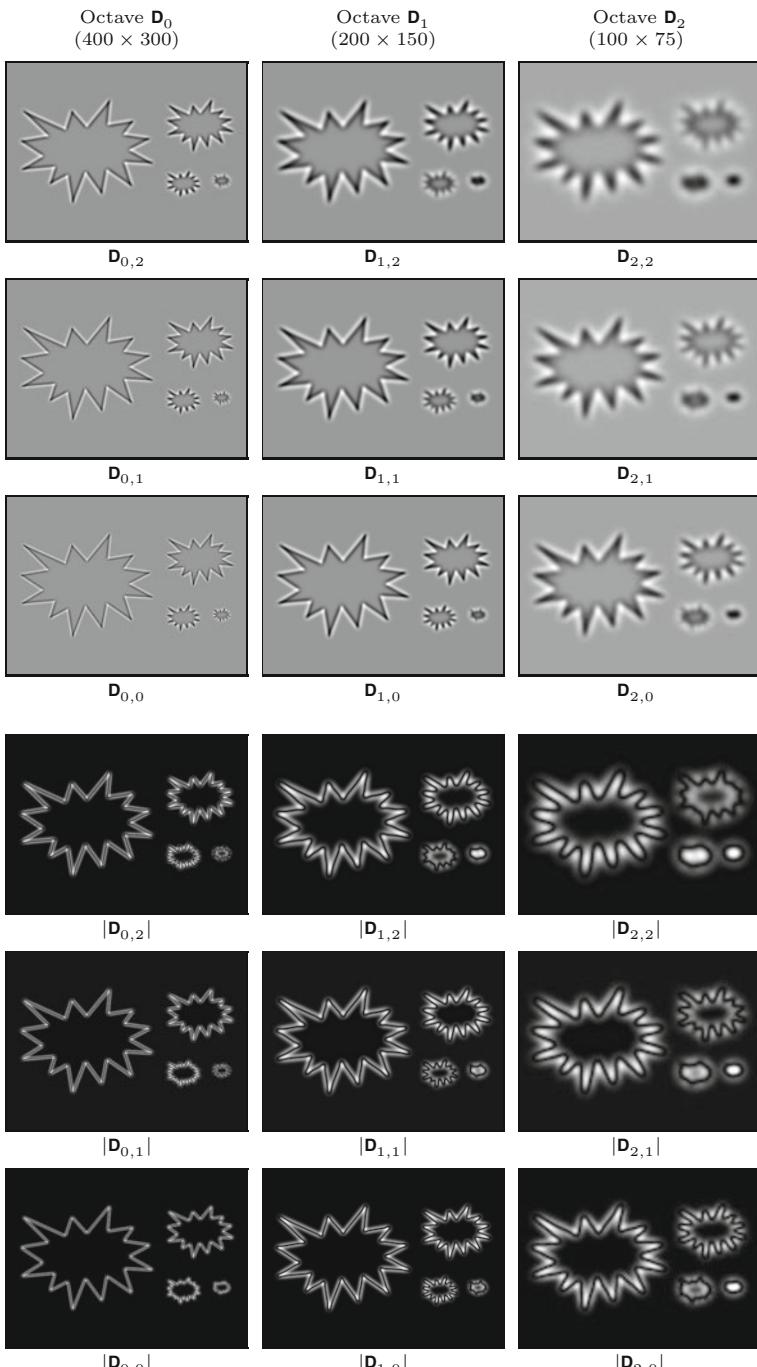
$\dot{\sigma}_q$... decimated scale (Eqn. (25.37))

$\tilde{\sigma}_q$... relative decimated scale w.r.t. octave's base level $\mathbf{G}_{p,-1}$ (Eqn. (25.38))

$P = 3$ (number of octaves)

$Q = 3$ (levels per octave)

$\sigma_0 = 1.6$ (base scale)



25.1 INTEREST POINTS AT MULTIPLE SCALES

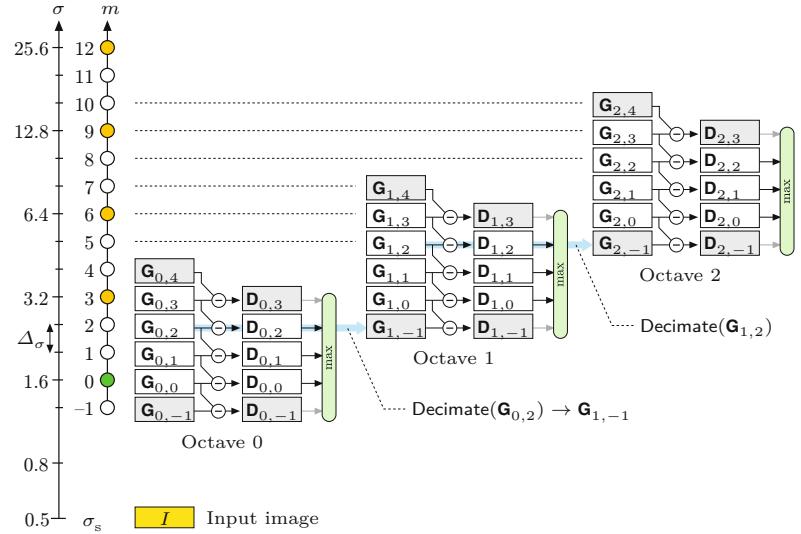
Fig. 25.13

Hierarchical DoG scale space example (stars image). The three top rows show the positive and negative DoG values (zero is mapped to intermediate gray). The three bottom rows show the absolute values of the DoG results (zero is mapped to black, maximum values to white). All images are scaled to the size of the original image.

$\sigma'_{0,4} = 2.4525$). Note that the inner levels $\mathbf{G}_{p,q}$ of all higher octaves (i.e., $p > 0, q \geq 0$) are calculated from the base level $\mathbf{G}_{p,-1}$, using the *same* set of kernels as for the first octave, as listed in Table 25.3. The complete process of building a SIFT scale space is summarized in Alg. 25.2.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.14 Scale space structure for SIFT with $P = 3$ octaves and $Q = 3$ levels per octave. To perform local maximum detection (“max”) over the full octave, $Q + 2$ DoG scale space levels ($\mathbf{D}_{p,-1}, \dots, \mathbf{D}_{p,Q}$) are required. The blue arrows indicate the decimation steps between successive Gaussian octaves. Since the DoG levels are obtained by subtracting pairs of Gaussian scale space levels, $Q + 3$ such levels ($\mathbf{G}_{p,-1}, \dots, \mathbf{G}_{p,Q+1}$) are needed in each octave \mathbf{G}_p . The two vertical axes on the left show the absolute scale (σ) and the discrete scale index (m), respectively. Note that the values along the scale axis are logarithmic with constant multiplicative scale increments $\Delta\sigma = 2^{1/Q}$. The absolute scale of the input image (I) is assumed as $\sigma_s = 0.5$.



25.2 Key Point Selection and Refinement

Key points are identified in three steps: (1) detection of extremal points in the DoG scale space, (2) position refinement by local interpolation, and (3) elimination of edge responses. These steps are detailed in the following and summarized in Algs. 25.3–25.6.

25.2.1 Local Extrema Detection

In the first step, candidate interest points are detected as local extrema in the 3D DoG scale space that we described in the previous section. Extrema detection is performed independently within each octave p . For the sake of convenience we define the 3D scale space coordinate $\mathbf{c} = (u, v, q)$, composed of the spatial position (u, v) and the level index q , as well as the function

$$D(\mathbf{c}) := \mathbf{D}_{p,q+k}(u, v) \quad (25.51)$$

as a short notation for selecting DoG values from a given octave p . Also, for collecting the DoG values in the 3D neighborhood around a scale space position \mathbf{c} , we define the map

$$\mathbf{N}_{\mathbf{c}}(i, j, k) := D(\mathbf{c} + i \cdot \mathbf{e}_i + j \cdot \mathbf{e}_j + k \cdot \mathbf{e}_k), \quad (25.52)$$

with $i, j, k \in \{-1, 0, 1\}$ and the 3D unit vectors

$$\mathbf{e}_i = (1, 0, 0)^T, \quad \mathbf{e}_j = (0, 1, 0)^T, \quad \mathbf{e}_k = (0, 0, 1)^T. \quad (25.53)$$

The neighborhood $\mathbf{N}_{\mathbf{c}}$ includes the center value $D(\mathbf{c})$ and the 26 values of its immediate neighbors (see Fig. 25.15(a)). These values are used to estimate the 3D gradient vector and the Hessian matrix for the 3D scale space position \mathbf{c} , as will be described.

A DoG scale space position \mathbf{c} is accepted as a local extremum (minimum or maximum) if the associated value $D(\mathbf{c}) = \mathbf{N}_{\mathbf{c}}(0, 0, 0)$

```

1: BuildSiftScaleSpace( $I, \sigma_s, \sigma_0, P, Q$ )
Input:  $I$ , source image;  $\sigma_s$ , sampling scale;  $\sigma_0$ , reference scale of
the first octave;  $P$ , number of octaves;  $Q$ , number of scale steps
per octave. Returns a SIFT scale space representation  $\langle \mathbf{G}, \mathbf{D} \rangle$  of
the image  $I$ .
2:  $\sigma_{\text{init}} \leftarrow \sigma_0 \cdot 2^{-1/Q}$             $\triangleright$  abs. scale at level  $(0, -1)$ , Eq. 25.47
3:  $\bar{\sigma}_{\text{init}} \leftarrow \sqrt{\sigma_{\text{init}}^2 - \sigma_s^2}$      $\triangleright$  relative scale w.r.t.  $\sigma_s$ , Eq. 25.49
4:  $\mathbf{G}_{\text{init}} \leftarrow I * H^{G, \bar{\sigma}_{\text{init}}}$            $\triangleright$  2D Gaussian filter with  $\bar{\sigma}_{\text{init}}$ 
5:  $\mathbf{G}_0 \leftarrow \text{MakeGaussianOctave}(\mathbf{G}_{\text{init}}, 0, Q, \sigma_0)$      $\triangleright$  Gauss. octave 0
6: for  $p \leftarrow 1, \dots, P-1$  do                   $\triangleright$  for octaves  $1, \dots, P-1$ 
7:    $\mathbf{G}_{\text{next}} \leftarrow \text{Decimate}(\mathbf{G}_{p-1, Q-1})$        $\triangleright$  see Alg. 25.1
8:    $\mathbf{G}_p \leftarrow \text{MakeGaussianOctave}(\mathbf{G}_{\text{next}}, p, Q, \sigma_0)$      $\triangleright$  octave  $p$ 
9:  $\mathbf{G} \leftarrow (\mathbf{G}_0, \dots, \mathbf{G}_{P-1})$            $\triangleright$  assemble the Gaussian scale space  $\mathbf{G}$ 
10: for  $p \leftarrow 0, \dots, P-1$  do
11:    $\mathbf{D}_p \leftarrow \text{MakeDogOctave}(\mathbf{G}_p, p, Q)$ 
12:  $\mathbf{D} \leftarrow (\mathbf{D}_0, \dots, \mathbf{D}_{P-1})$            $\triangleright$  assemble the DoG scale space  $\mathbf{D}$ 
13: return  $\langle \mathbf{G}, \mathbf{D} \rangle$ 

14: MakeGaussianOctave( $\mathbf{G}_{\text{base}}, p, Q, \sigma_0$ )
Input:  $\mathbf{G}_{\text{base}}$ , Gaussian base level;  $p$ , octave index;  $Q$ , scale steps
per octave,  $\sigma_0$ , reference scale. Returns a new Gaussian octave
 $\mathbf{G}_p$  with  $Q+3$  levels levels.
15:  $\mathbf{G}_{p,-1} \leftarrow \mathbf{G}_{\text{base}}$             $\triangleright$  level  $q = -1$ 
16: for  $q \leftarrow 0, \dots, Q+1$  do           $\triangleright$  levels  $q = -1, \dots, Q+1$ 
17:    $\tilde{\sigma}_q \leftarrow \sigma_0 \cdot \sqrt{2^{2q/Q} - 2^{-2/Q}}$    $\triangleright$  rel. scale w.r.t base level  $\mathbf{G}_{\text{base}}$ 
18:    $\mathbf{G}_{p,q} \leftarrow \mathbf{G}_{\text{base}} * H^{G, \tilde{\sigma}_q}$         $\triangleright$  2D Gaussian filter with  $\tilde{\sigma}_q$ 
19:    $\mathbf{G}_p \leftarrow (\mathbf{G}_{p,-1}, \dots, \mathbf{G}_{p,Q+1})$ 
20: return  $\mathbf{G}_p$ 

21: MakeDogOctave( $\mathbf{G}_p, p, Q$ )
Input:  $\mathbf{G}_p$ , Gaussian octave;  $p$ , octave index;  $Q$ , scale steps per
octave. Returns a new DoG octave  $\mathbf{D}_p$  with  $Q+2$  levels.
22: for  $q \leftarrow -1, \dots, Q$  do
23:    $\mathbf{D}_{p,q} \leftarrow \mathbf{G}_{p,q+1} - \mathbf{G}_{p,q}$            $\triangleright$  diff. of Gaussians, Eq. 25.30
24:    $\mathbf{D}_p \leftarrow (\mathbf{D}_{p,-1}, \mathbf{D}_{p,0}, \dots, \mathbf{D}_{p,Q})$        $\triangleright$  levels  $q = -1, \dots, Q$ 
25: return  $\mathbf{D}_p$ 

```

25.2 KEY POINT

SELECTION AND
REFINEMENT

Alg. 25.2

Building a SIFT scale space. This procedure is an extension of Alg. 25.1 and takes the same parameters. The SIFT scale space (see Fig. 25.14) consists of two components: a hierarchical Gaussian scale space $\mathbf{G} = (\mathbf{G}_0, \dots, \mathbf{G}_{P-1})$ with P octaves and a (derived) hierarchical DoG scale space $\mathbf{D} = (\mathbf{D}_0, \dots, \mathbf{D}_{P-1})$. Each Gaussian octave \mathbf{G}_p holds $Q+3$ levels ($\mathbf{G}_{p,-1}, \dots, \mathbf{G}_{p,Q+1}$). At each Gaussian octave, the lowest level $\mathbf{G}_{p,-1}$ is obtained by decimating level $Q-1$ of the previous octave \mathbf{G}_{p-1} (line 7). Every DoG octave \mathbf{D}_p contains $Q+2$ levels ($\mathbf{D}_{p,-1}, \dots, \mathbf{D}_{p,Q}$). A DoG level $\mathbf{D}_{p,q}$ is calculated as the pointwise difference of two adjacent Gaussian levels $\mathbf{G}_{p,q+1}$ and $\mathbf{G}_{p,q}$ (line 23). Typical parameter settings are $\sigma_s = 0.5$, $\sigma_0 = 1.6$, $Q = 3$, $P = 4$.

is either *negative* and also *smaller* or *positive* and *greater* than all neighboring values. In addition, a minimum difference $t_{\text{extrem}} \geq 0$ can be specified, indicating how much the center value must at least deviate from the surrounding values. The decision whether a given neighborhood N_c contains a local minimum or maximum can thus be expressed as

$$\begin{aligned} \text{IsLocalMin}(N_c) := N_c(0, 0, 0) < 0 \wedge \\ N_c(0, 0, 0) + t_{\text{extrem}} < \min_{\substack{(i,j,k) \neq \\ (0,0,0)}} N_c(i, j, k), \end{aligned} \quad (25.54)$$

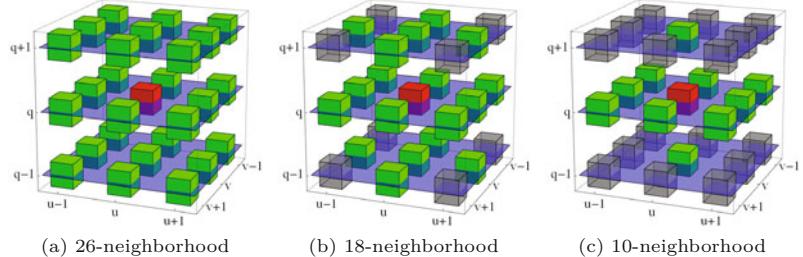
$$\begin{aligned} \text{IsLocalMax}(N_c) := N_c(0, 0, 0) > 0 \wedge \\ N_c(0, 0, 0) - t_{\text{extrem}} < \max_{\substack{(i,j,k) \neq \\ (0,0,0)}} N_c(i, j, k) \end{aligned} \quad (25.55)$$

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.15

Different 3D neighborhoods for detecting local extrema in the DoG scale space. The red cube represents the DoG value at the reference coordinate $\mathbf{c} = (u, v, q)$ at the spatial position (u, v) at scale level q (within some octave p). Full $3 \times 3 \times 3$ neighborhood with 26 elements (a); other types of neighborhoods with 18 or 10 cells may be specified for extrema detection.

respectively, are also commonly used. A local maximum/minimum is detected if the DoG value at the center is greater/smaller than all neighboring values (green cubes).



(see procedure `IsExtremum(Nc)` in Alg. 25.5). As illustrated in Fig. 25.15(b–c), alternative 3D neighborhoods with 18 or 10 cells may be specified for extrema detection.

25.2.2 Position Refinement

Once a local extremum is detected in the DoG scale space, only its *discrete* 3D coordinates $\mathbf{c} = (u, v, q)$ are known, consisting of the spatial grid position (u, v) and the index (q) of the associated scale level. In the second step, a more accurate, *continuous* position for each candidate key point is estimated by fitting a quadratic function to the local neighborhood, as proposed in [37]. This is particularly important at the higher octaves of the scale space, where the spatial resolution becomes increasingly coarse due to successive decimation. Position refinement is based on a local second-order Taylor expansion of the discrete DoG function, which yields a continuous approximation function whose maximum or minimum can be found analytically. Additional details and illustrative examples are provided in Sec. C.3.2 of the Appendix.

At any extremal position $\mathbf{c} = (u, v, q)$ in octave p of the hierarchical DoG scale space \mathbf{D} , the corresponding $3 \times 3 \times 3$ neighborhood $N_D(\mathbf{c})$ is used to estimate the elements of the continuous 3D gradient, that is,

$$\nabla_D(\mathbf{c}) = \begin{pmatrix} d_x \\ d_y \\ d_\sigma \end{pmatrix} \approx \frac{1}{2} \cdot \begin{pmatrix} D(\mathbf{c} + \mathbf{e}_i) - D(\mathbf{c} - \mathbf{e}_i) \\ D(\mathbf{c} + \mathbf{e}_j) - D(\mathbf{c} - \mathbf{e}_j) \\ D(\mathbf{c} + \mathbf{e}_k) - D(\mathbf{c} - \mathbf{e}_k) \end{pmatrix}, \quad (25.56)$$

with $D()$ as defined in Eqn. (25.51). Similarly, the 3×3 Hessian matrix for position \mathbf{c} is obtained as

$$\mathbf{H}_D(\mathbf{c}) = \begin{pmatrix} d_{xx} & d_{xy} & d_{x\sigma} \\ d_{xy} & d_{yy} & d_{y\sigma} \\ d_{x\sigma} & d_{y\sigma} & d_{\sigma\sigma} \end{pmatrix}, \quad (25.57)$$

with the required second order derivatives estimated as

$$\begin{aligned} d_{xx} &= D(\mathbf{c} - \mathbf{e}_i) - 2 \cdot D(\mathbf{c}) + D(\mathbf{c} + \mathbf{e}_i), \\ d_{yy} &= D(\mathbf{c} - \mathbf{e}_j) - 2 \cdot D(\mathbf{c}) + D(\mathbf{c} + \mathbf{e}_j), \\ d_{\sigma\sigma} &= D(\mathbf{c} - \mathbf{e}_k) - 2 \cdot D(\mathbf{c}) + D(\mathbf{c} + \mathbf{e}_k), \\ d_{xy} &= \frac{D(\mathbf{c} + \mathbf{e}_i + \mathbf{e}_j) - D(\mathbf{c} - \mathbf{e}_i + \mathbf{e}_j) - D(\mathbf{c} + \mathbf{e}_i - \mathbf{e}_j) + D(\mathbf{c} - \mathbf{e}_i - \mathbf{e}_j)}{4}, \\ d_{x\sigma} &= \frac{D(\mathbf{c} + \mathbf{e}_i + \mathbf{e}_k) - D(\mathbf{c} - \mathbf{e}_i + \mathbf{e}_k) - D(\mathbf{c} + \mathbf{e}_i - \mathbf{e}_k) + D(\mathbf{c} - \mathbf{e}_i - \mathbf{e}_k)}{4}, \\ d_{y\sigma} &= \frac{D(\mathbf{c} + \mathbf{e}_j + \mathbf{e}_k) - D(\mathbf{c} - \mathbf{e}_j + \mathbf{e}_k) - D(\mathbf{c} + \mathbf{e}_j - \mathbf{e}_k) + D(\mathbf{c} - \mathbf{e}_j - \mathbf{e}_k)}{4}. \end{aligned} \quad (25.58)$$

See the procedures **Gradient**(\mathbf{N}_c) and **Hessian**(\mathbf{N}_c) in Alg. 25.5 (p. 651) for additional details. From the gradient vector $\nabla_D(\mathbf{c})$ and the Hessian matrix $\mathbf{H}_D(\mathbf{c})$, the second order Taylor expansion around point \mathbf{c} is

$$\tilde{D}_c(\mathbf{x}) = D(\mathbf{c}) + \nabla_D^\top(\mathbf{c}) \cdot (\mathbf{x} - \mathbf{c}) + \frac{1}{2}(\mathbf{x} - \mathbf{c})^\top \cdot \mathbf{H}_D(\mathbf{c}) \cdot (\mathbf{x} - \mathbf{c}), \quad (25.59)$$

for the continuous position $\mathbf{x} = (x, y, \sigma)^\top$. The scalar-valued function $\tilde{D}_c(\mathbf{x}) \in \mathbb{R}$, with $\mathbf{c} = (u, v, q)^\top$ and $\mathbf{x} = (x, y, \sigma)^\top$, is a local, *continuous* approximation of the discrete DoG function $\mathbf{D}_{p,q}(u, v)$ at octave p , scale level q , and spatial position u, v . This is a quadratic function with an extremum (maximum or minimum) at position

$$\check{\mathbf{x}} = \begin{pmatrix} \check{x} \\ \check{y} \\ \check{\sigma} \end{pmatrix} = \mathbf{c} + \mathbf{d} = \mathbf{c} - \underbrace{\mathbf{H}_D^{-1}(\mathbf{c}) \cdot \nabla_D(\mathbf{c})}_{\mathbf{d} = \check{\mathbf{x}} - \mathbf{c}} \quad (25.60)$$

with $\mathbf{d} = (x', y', \sigma')^\top = \check{\mathbf{x}} - \mathbf{c}$, under the assumption that the inverse of the Hessian matrix \mathbf{H}_D exists. By inserting the extremal position $\check{\mathbf{x}}$ into Eqn. (25.59), the peak (minimum or maximum) *value* of the continuous approximation function \tilde{D} is found as¹⁵

$$\begin{aligned} D_{\text{peak}}(\mathbf{c}) &= \tilde{D}_c(\check{\mathbf{x}}) = D(\mathbf{c}) + \frac{1}{2} \cdot \nabla_D^\top(\mathbf{c}) \cdot (\check{\mathbf{x}} - \mathbf{c}) \\ &= D(\mathbf{c}) + \frac{1}{2} \cdot \nabla_D^\top(\mathbf{c}) \cdot \mathbf{d}, \end{aligned} \quad (25.61)$$

where $\mathbf{d} = \check{\mathbf{x}} - \mathbf{c}$ (cf. Eqn. (25.60)) denotes the 3D vector between the neighborhood's discrete center position \mathbf{c} and the continuous extremal position $\check{\mathbf{x}}$.

A scale space location \mathbf{c} is only retained as a candidate interest point if the estimated magnitude of the DoG exceeds a given threshold t_{peak} , that is, if

$$|D_{\text{peak}}(\mathbf{c})| > t_{\text{peak}}. \quad (25.62)$$

If the distance $\mathbf{d} = (x', y', \sigma')^\top$ from \mathbf{c} to the estimated (continuous) peak position $\check{\mathbf{x}}$ in Eqn. (25.60) is greater than a predefined limit (typically 0.5) in any spatial direction, the center point $\mathbf{c} = (u, v, q)^\top$ is moved to one of the neighboring DoG cells by maximally ± 1 unit steps along the u, v axes, that is,

$$\mathbf{c} \leftarrow \mathbf{c} + \begin{pmatrix} \min(1, \max(-1, \text{round}(x'))) \\ \min(1, \max(-1, \text{round}(y'))) \\ 0 \end{pmatrix}. \quad (25.63)$$

The q component of \mathbf{c} is not modified in this version, that is, the search continues at the original scale level.¹⁶ Based on the surrounding 3D neighborhood of this new point, a Taylor expansion (Eqn. (25.60)) is again performed to estimate a new peak location. This is repeated until either the peak location is inside the current DoG cell or the allowed number of repositioning steps n_{refine} is reached

¹⁵ See Eqn. (C.64) in Sec. C.3.3 in the Appendix for details.

¹⁶ This is handled differently in other SIFT implementations.

(typically n_{refine} is set to 4 or 5). If successful, the result of this step is a candidate feature point

$$\check{\mathbf{c}} = (\check{x}, \check{y}, \check{q})^T = \mathbf{c} + (x', y', 0)^T. \quad (25.64)$$

Notice that (in this implementation) the scale level q remains unchanged even if the 3D Taylor expansion indicates that the estimated peak is located at another scale level. See procedure `RefineKeyPosition()` in Alg. 25.4 (p. 650) for a concise summary of these steps.

It should be mentioned that the original publication [153] is not particularly explicit about the aforementioned position refinement process and thus slightly different approaches are used in various open-source SIFT implementations. For example, the implementation in *VLFeat*¹⁷ [241] moves to one of the direct neighbors at the same scale level as described earlier, as long as $|x'|$ or $|y'|$ is greater than 0.6. *AutoPano-SIFT*¹⁸ by S. Nowozin calculates the length of the spatial displacement $d = \|(x', y')\|$ and discards the current point if $d > 2$. Otherwise it moves by $\Delta_u = \text{round}(x')$, $\Delta_v = \text{round}(y')$ without limiting the displacement to ± 1 . The *Open-Source SIFT Library*¹⁹ [106] used in *OpenCV* also makes full moves in the spatial directions and, in addition, potentially also changes the scale level by $\Delta_q = \text{round}(\sigma')$ in each iteration.

25.2.3 Suppressing Responses to Edge-Like Structures

In the previous step, candidate interest points were selected as those locations in the DoG scale space where the Taylor approximation had a local maximum and the extrapolated DoG value was above a given threshold (t_{peak}). However, the DoG filter also responds strongly to edge-like structures. At such positions, interest points cannot be located with sufficient stability and repeatability. To eliminate the responses near edges, Lowe suggests the use of the principal curvatures of the 2D DoG result along the spatial x, y axes, using the fact that the principal curvatures of a function are proportional to the eigenvalues of the function's Hessian matrix at a given point.

For a particular lattice point $\mathbf{c} = (u, v, q)$ in DoG scale space, with neighborhood \mathbf{N}_D (see Eqn. (25.52)), the 2×2 Hessian matrix for the spatial coordinates is

$$\mathbf{H}_{xy}(\mathbf{c}) = \begin{pmatrix} d_{xx} & d_{xy} \\ d_{xy} & d_{yy} \end{pmatrix}, \quad (25.65)$$

with d_{xx} , d_{xy} , d_{yy} as defined in Eqn. (25.58), that is, these values can be extracted from the corresponding 3×3 Hessian matrix $\mathbf{H}_D(\mathbf{c})$ (see Eqn. (25.57)).

The matrix $\mathbf{H}_{xy}(\mathbf{c})$ has two eigenvalues λ_1, λ_2 , which we define as being ordered, such that λ_1 has the greater magnitude ($|\lambda_1| \geq |\lambda_2|$). If both eigenvalues for a point \mathbf{c} are of similar magnitude, the function exhibits a high curvature along two orthogonal directions and in this

¹⁷ <http://www.vlfeat.org/overview/sift.html>.

¹⁸ <http://sourceforge.net/projects/hugin/files/autopano-sift-C/>.

¹⁹ <http://robwhess.github.io/opensift/>.

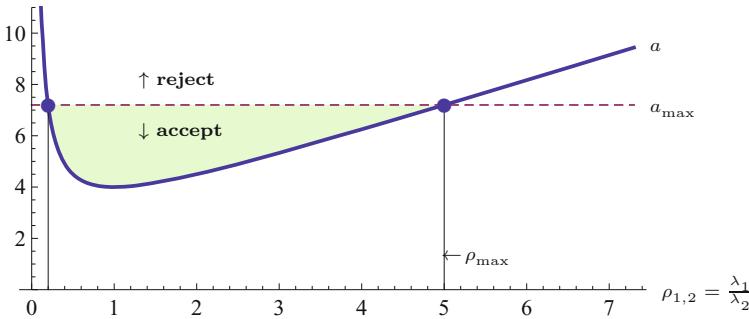


Fig. 25.16

Limiting the ratio of principal curvatures (edge ratio) $\rho_{1,2}$ by specifying a_{\max} . The quantity a (blue line) has a minimum when the eigenvalue ratio $\rho_{1,2} = \frac{\lambda_1}{\lambda_2}$ is one, that is, when the two eigenvalues λ_1, λ_2 are equal, indicating a corner-like event. Typically only one of the eigenvalues is dominant in the vicinity of image lines, such that $\rho_{1,2}$ and a values are significantly increased. In this example, the principal curvature ratio $\rho_{1,2}$ is limited to $\rho_{\max} = 5.0$ by setting $a_{\max} = (5+1)^2/5 = 7.2$ (red line).

case \mathbf{c} is likely to be a good reference point that can be located reliably. In the optimal situation (e.g., near a corner), the ratio of the eigenvalues $\rho = \lambda_1/\lambda_2$ is close to 1. Alternatively, if the ratio ρ is high it can be concluded that a single orientation dominates at this position, as is typically the case in the neighborhood of edges.

To estimate the ratio ρ it is not necessary to calculate the eigenvalues themselves. Following the description in [153], the sum and product of the eigenvalues λ_1, λ_2 can be found as

$$\lambda_1 + \lambda_2 = \text{trace}(\mathbf{H}_{xy}(\mathbf{c})) = d_{xx} + d_{yy}, \quad (25.66)$$

$$\lambda_1 \cdot \lambda_2 = \det(\mathbf{H}_{xy}(\mathbf{c})) = d_{xx} \cdot d_{yy} - d_{xy}^2. \quad (25.67)$$

If the determinant $\det(\mathbf{H}_{xy})$ is *negative*, the principal curvatures of the underlying 2D function have opposite signs and thus point \mathbf{c} can be discarded as not being an extremum. Otherwise, if the signs of both eigenvalues λ_1, λ_2 are the *same*, then the ratio

$$\rho_{1,2} = \frac{\lambda_1}{\lambda_2} \quad (25.68)$$

is positive (with $\lambda_1 = \rho_{1,2} \cdot \lambda_2$), and thus the expression

$$a = \frac{[\text{trace}(\mathbf{H}_{xy}(\mathbf{c}))]^2}{\det(\mathbf{H}_{xy}(\mathbf{c}))} = \frac{(\lambda_1 + \lambda_2)^2}{\lambda_1 \cdot \lambda_2} \quad (25.69)$$

$$= \frac{(\rho_{1,2} \cdot \lambda_2 + \lambda_2)^2}{\rho_{1,2} \cdot \lambda_2^2} = \frac{\lambda_2^2 \cdot (\rho_{1,2} + 1)^2}{\rho_{1,2} \cdot \lambda_2^2} = \frac{(\rho_{1,2} + 1)^2}{\rho_{1,2}} \quad (25.70)$$

depends only on the ratio $\rho_{1,2}$. If the determinant of \mathbf{H}_{xy} is positive, the quantity a has a minimum (4.0) at $\rho_{1,2} = 1$, if the two eigenvalues are equal (see Fig. 25.16). Note that the ratio a is the same for $\rho_{1,2} = \lambda_1/\lambda_2$ or $\rho_{1,2} = \lambda_2/\lambda_1$, since

$$a = \frac{(\rho_{1,2} + 1)^2}{\rho_{1,2}} = \frac{\left(\frac{1}{\rho_{1,2}} + 1\right)^2}{\frac{1}{\rho_{1,2}}}. \quad (25.71)$$

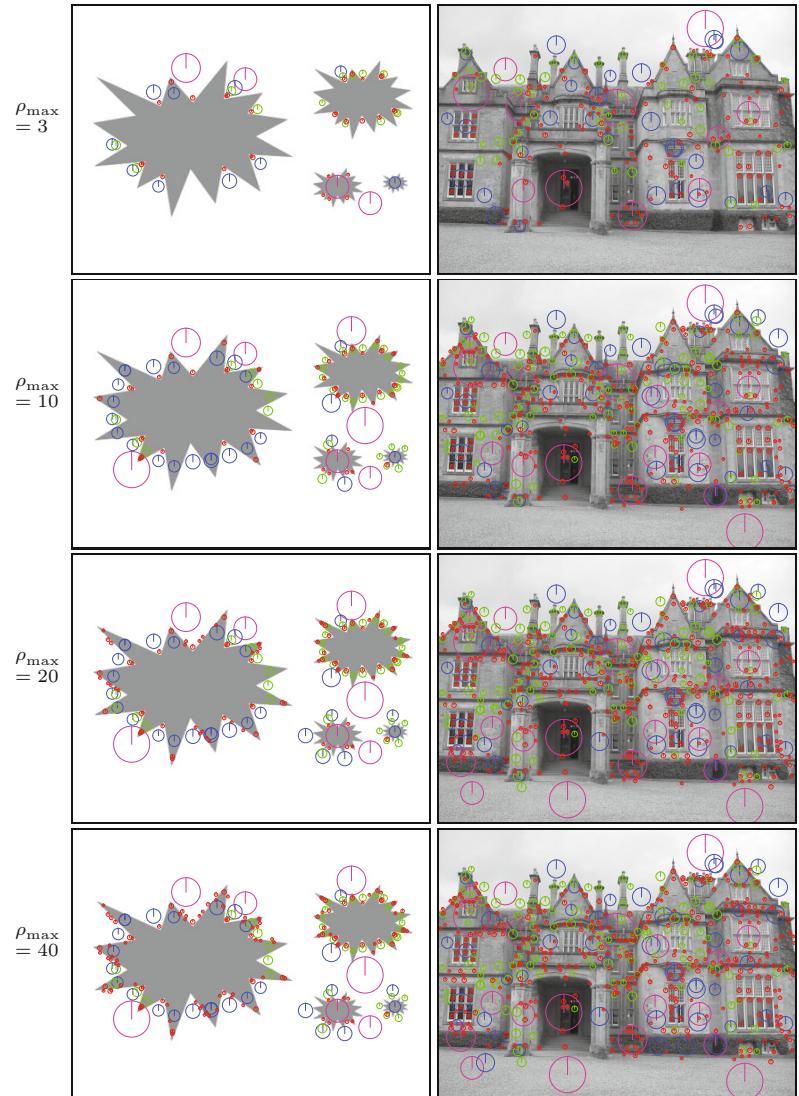
To verify that the eigenvalue ratio $\rho_{1,2}$ at a given position \mathbf{c} is *below* a specified limit ρ_{\max} (making \mathbf{c} a good candidate), it is thus sufficient to check the condition

$$a \leq a_{\max}, \quad \text{with} \quad a_{\max} = \frac{(\rho_{\max} + 1)^2}{\rho_{\max}}, \quad (25.72)$$

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.17

Rejection of edge-like features by controlling the max. curvature ratio ρ_{\max} . The size of the circles is proportional to the scale level at which the corresponding key point was detected, the color indicating the containing octave ($0 = \text{red}$, $1 = \text{green}$, $2 = \text{blue}$, $3 = \text{magenta}$).



without the need to actually calculate the individual eigenvalues λ_1 and λ_2 .²⁰ ρ_{\max} should be greater than 1 and is typically chosen to be in the range $3, \dots, 10$ ($\rho_{\max} = 10$ is suggested in [153]). The resulting value of a_{\max} in Eqn. (25.72) is constant and needs only be calculated once (see Alg. 25.3, line 2). Detection examples for varying values of ρ_{\max} are shown in Fig. 25.17. Note that considerably more candidates appear near edges as ρ_{\max} is raised from 3 to 40.

25.3 Creating Local Descriptors

For each local maximum detected in the hierarchical DoG scale space, a candidate key point is created, which is subsequently refined to

²⁰ A similar trick is used in the *Harris* corner detection algorithm (see Chapter 7).

a continuous position following the steps we have just described (see Eqns. (25.56)–(25.64)). Then, for each refined key point $\mathbf{k}' = (p, q, x, y)$, one or more (up to four) local descriptors are calculated. Multiple (up to four) descriptors may be created for a position if the local orientation is not unique. This process involves the following steps:

1. Find the *dominant* orientation(s) of the key point \mathbf{k}' from the distribution of the gradients at the corresponding Gaussian scale space level.
2. For each dominant orientation, create a separate SIFT *descriptor* at the key point \mathbf{k}' .

25.3.1 Finding Dominant Orientations

Local orientation from Gaussian scale space

Orientation vectors are obtained by sampling the *gradient* values of the hierarchical Gaussian scale space $\mathbf{G}_{p,q}(u, v)$ (see Eqn. (25.32)). For any lattice position (u, v) at octave p and scale level q , the local gradient is calculated as

$$\nabla_{p,q}(u, v) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} = 0.5 \cdot \begin{pmatrix} \mathbf{G}_{p,q}(u+1, v) - \mathbf{G}_{p,q}(u-1, v) \\ \mathbf{G}_{p,q}(u, v+1) - \mathbf{G}_{p,q}(u, v-1) \end{pmatrix}. \quad (25.73)$$

From these gradient vectors, the gradient *magnitude* and *orientation* (i.e., polar coordinates) are found as²¹

$$E_{p,q}(u, v) = \|\nabla_{p,q}(u, v)\| = \sqrt{d_x^2 + d_y^2}, \quad (25.74)$$

$$\phi_{p,q}(u, v) = \angle \nabla_{p,q}(u, v) = \tan^{-1}(d_y/d_x). \quad (25.75)$$

These scalar fields $E_{p,q}$ and $\phi_{p,q}$ are typically pre-calculated for all relevant octaves/levels p, q of the Gaussian scale space \mathbf{G} .

Orientation histograms

To find the dominant orientations for a given key point, a histogram \mathbf{h}_ϕ of the orientation angles is calculated for the gradient vectors collected from a square window around the key point center. Typically the histogram has $n_{\text{orient}} = 36$ bins, that is, the angular resolution is 10° . The orientation histogram is collected from a square region using an isotropic Gaussian weighting function whose width σ_w is proportional to the *decimated scale* $\dot{\sigma}_q$ (see Eqn. (25.37)) of the key point's scale level q . Typically a Gaussian weighting function “with a σ that is 1.5 times that of the scale of the key point” [153] is used, that is,

$$\sigma_w = 1.5 \cdot \dot{\sigma}_q = 1.5 \cdot \sigma_0 \cdot 2^{q/Q}. \quad (25.76)$$

Note that σ_w is independent of the octave index p and thus the same weighting functions are used in each octave. To calculate the *orientation histogram*, the Gaussian gradients around the given key point are collected from a square region of size $2r_w \times 2r_w$, with

²¹ See also Chapter 16, Sec. 16.1.

$$r_w = \lceil 2.5 \cdot \sigma_w \rceil \quad (25.77)$$

amply dimensioned to avoid numerical truncation effects. For the parameters listed in [Table 25.3](#) ($\sigma_0 = 1.6$, $Q = 3$), the values for σ_w (expressed in the octave's coordinate units) are

q	0	1	2	3
σ_w	1.6000	2.0159	2.5398	3.2000
r_w	4	5	6	7

(25.78)

In [Alg. 25.7](#), σ_w and r_w of the Gaussian weighting function are calculated in lines 7 and 8, respectively. At each lattice point (u, v) , the gradient vector $\nabla_{p,q}(u, v)$ is calculated in octave p and level q of the Gaussian scale space \mathbf{G} ([Alg. 25.7](#), line 16). From this, the gradient magnitude $E_{p,q}(u, v)$ and orientation $\phi_{p,q}(u, v)$ are obtained (lines 29–30). The corresponding Gaussian weight is calculated (in line 18) from the spatial distance between the grid point (u, v) and the interest point (x, y) as

$$w_G(u, v) = \exp\left(-\frac{(u-x)^2 + (v-y)^2}{2 \cdot \sigma_w^2}\right). \quad (25.79)$$

For the grid point (u, v) , the quantity to be accumulated into the orientation histogram is

$$z = E_{p,q}(u, v) \cdot w_G(u, v), \quad (25.80)$$

that is, the local gradient magnitude weighted by the Gaussian window function ([Alg. 25.7](#), line 19).

The orientation histogram h_ϕ consists of n_{orient} bins and thus the *continuous* bin number for the angle $\phi(u, v)$ is

$$\kappa_\phi = \frac{n_{\text{orient}}}{2\pi} \cdot \phi(u, v) \quad (25.81)$$

(see [Alg. 25.7](#), line 20). To collect the *continuous* orientations into a histogram with discrete bins, quantization must be performed. The simplest approach is to select the “nearest” bin (by rounding) and to add the associated quantity (denoted z) entirely to the selected bin. Alternatively, to reduce quantization effects, a common technique is to *split* the quantity z onto the two closest bins. Given the continuous bin value κ_ϕ , the indexes of the two closest discrete bins are

$$k_0 = \lfloor \kappa_\phi \rfloor \bmod n_{\text{orient}} \quad \text{and} \quad k_1 = (\lfloor \kappa_\phi \rfloor + 1) \bmod n_{\text{orient}}, \quad (25.82)$$

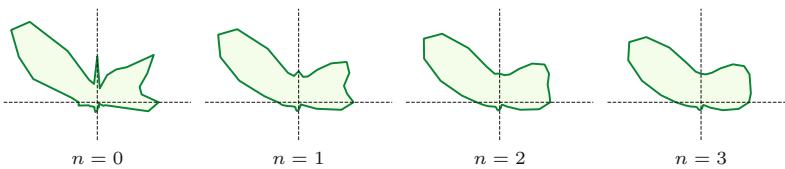
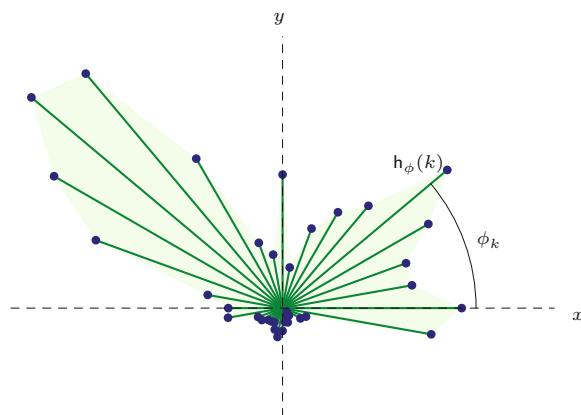
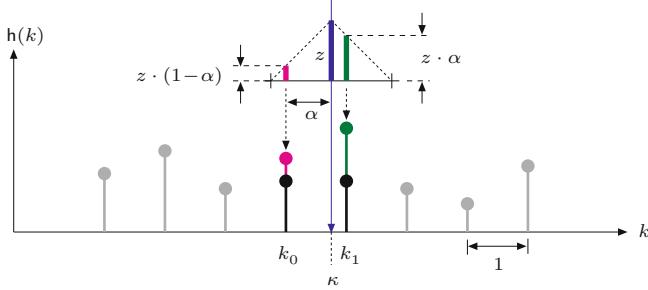
respectively. The quantity z ([Eqn. \(25.80\)](#)) is then partitioned and accumulated into the neighboring bins k_0, k_1 of the orientation histogram h_ϕ in the form

$$\begin{aligned} h_\phi(k_0) &\leftarrow h_\phi(k_0) + (1 - \alpha) \cdot z, \\ h_\phi(k_1) &\leftarrow h_\phi(k_1) + \alpha \cdot z, \end{aligned} \quad (25.83)$$

with $\alpha = \kappa_\phi - \lfloor \kappa_\phi \rfloor$. This process is illustrated by the example in [Fig. 25.18](#) (see also [Alg. 25.7](#), lines 21–25).

Fig. 25.18

Accumulating into multiple histogram bins by linear interpolation. Assume that some quantity z (blue bar) is to be added to the discrete histogram h_ϕ at the continuous position κ_ϕ . The histogram bins adjacent to κ_ϕ are $k_0 = \lfloor \kappa_\phi \rfloor$ and $k_1 = \lfloor \kappa_\phi \rfloor + 1$. The fraction of z accumulated into bin k_1 is $z_1 = z \cdot \alpha$ (red bar), with $\alpha = \kappa_\phi - k_0$. Analogously, the quantity added to bin k_0 is $z_0 = z \cdot (1 - \alpha)$ (green bar).



Orientation histogram smoothing

Figure 25.19 shows a geometric rendering of the orientation histogram that explains the relevance of the cell indexes (discrete angles ϕ_k) and the accumulated quantities (z). Before calculating the dominant orientations, the raw orientation histogram h_ϕ is usually smoothed by applying a (circular) low-pass filter, typically a simple 3-tap Gaussian or box-type filter (see procedure `SmoothCircular()` in Alg. 25.7, lines 6–16).²² Stronger smoothing is achieved by applying the filter multiple times, as illustrated in Fig. 25.20. In practice, two to three smoothing iterations appear to be sufficient.

Locating and interpolating orientation peaks

After smoothing the orientation histogram, the next step is to detect the peak entries in h_ϕ . A bin k is considered a significant orientation peak if $h_\phi(k)$ is a local maximum and its value is not less than a certain fraction of the maximum histogram entry, that is, only if

Fig. 25.19

Orientation histogram example. Each of the 36 radial bars corresponds to one entry in the orientation histogram h_ϕ . The length (radius) of each radial bar with index k is proportional to the accumulated value in the corresponding bin $h_\phi(k)$ and its orientation is ϕ_k .

Fig. 25.20

Smoothing the orientation histogram (from Fig. 25.19) by repeatedly applying a circular low-pass filter with the 1D kernel $H = \frac{1}{4} \cdot (1, 2, 1)$.

²² Histogram smoothing is not mentioned in the original SIFT publication [153] but used in most implementations.

$$\begin{aligned} h_\phi(k) &> h_\phi((k-1) \bmod n_{\text{orient}}) \wedge \\ h_\phi(k) &> h_\phi((k+1) \bmod n_{\text{orient}}) \wedge \\ h_\phi(k) &> t_{\text{domor}} \cdot \max_i h_\phi(i), \end{aligned} \quad (25.84)$$

with $t_{\text{domor}} = 0.8$ as a typical limit.

To achieve a finer angular resolution than provided by the orientation histogram bins (typically spaced at 10° steps) alone, a continuous peak orientation is calculated by quadratic interpolation of the neighboring histogram values. Given a discrete peak index k , the interpolated (continuous) peak position \check{k} is obtained by fitting a quadratic function to the three successive histogram values $h_\phi(k-1)$, $h_\phi(k)$, $h_\phi(k+1)$ as²³

$$\check{k} = k + \frac{h_\phi(k-1) - h_\phi(k+1)}{2 \cdot [h_\phi(k-1) - 2h_\phi(k) + h_\phi(k+1)]}, \quad (25.85)$$

with all indexes taken modulo n_{orient} . From Eqn. (25.81), the (continuous) dominant orientation angle $\theta \in [0, 2\pi)$ is then obtained as

$$\theta = (\check{k} \bmod n_{\text{orient}}) \cdot \frac{2\pi}{n_{\text{orient}}}, \quad (25.86)$$

mit $\theta \in [0, 2\pi)$. In this way, the dominant orientation can be estimated with accuracy much beyond the coarse resolution of the orientation histogram. Note that, in some cases, multiple histogram peaks are obtained for a given key point (see procedure `FindPeakOrientations()` in Alg. 25.6, lines 18–31). In this event, individual SIFT descriptors are created for each dominant orientation at the same key point position (see Alg. 25.3, line 8).

Figure 25.21 shows the orientation histograms for a set of detected key points in two different images after applying a varying number of smoothing steps. It also shows the interpolated dominant orientations θ calculated from the orientation histograms (Eqn. (25.86)) by the corresponding vectors.

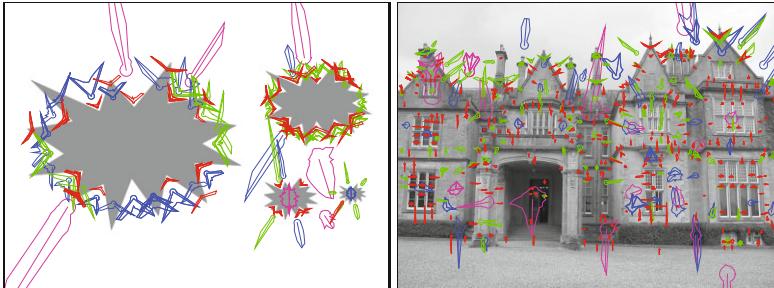
25.3.2 SIFT Descriptor Construction

For each key point $\mathbf{k}' = (p, q, x, y)$ and each dominant orientation θ , a corresponding SIFT descriptor is obtained by sampling the surrounding gradients at octave p and level q of the Gaussian scale space \mathbf{G} .

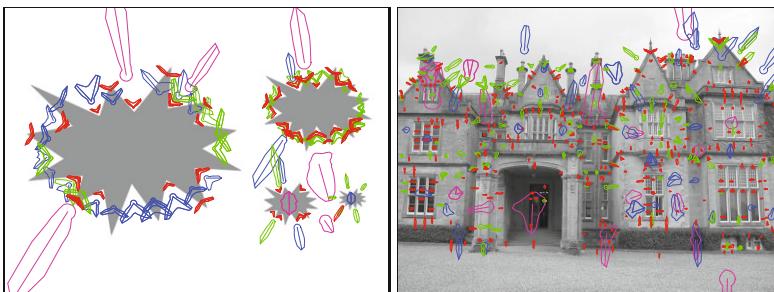
Descriptor geometry

The geometry underlying the calculation of SIFT descriptors is illustrated in Fig. 25.22. The descriptor combines the gradient orientation and magnitude from a square region of size $w_d \times w_d$, which is centered at the (continuous) position (x, y) of the associated feature point and aligned with its dominant orientation θ . The side length of the descriptor is set to $w_d = 10 \cdot \dot{\sigma}_q$, where $\dot{\sigma}_q$ denotes the key point's decimated scale (radius of the inner circle). It depends on the key point's scale level q (see Table 25.4).

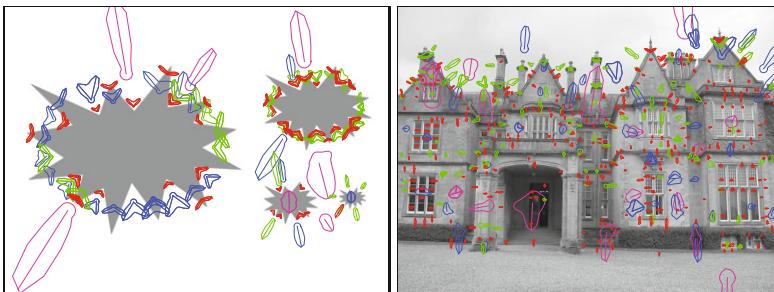
²³ See Sec. C.1.2 in the Appendix for details.



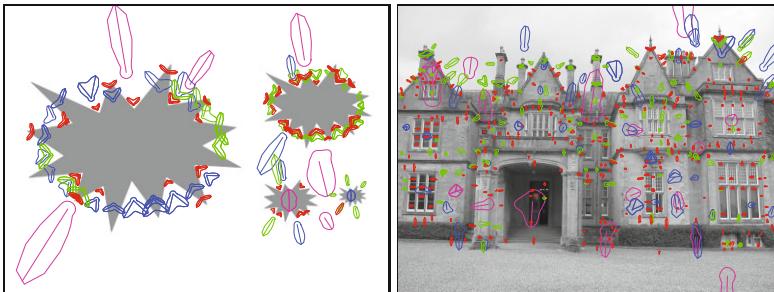
(a) $n = 0$



(b) $n = 1$



(c) $n = 2$



(d) $n = 3$

25.3 CREATING LOCAL DESCRIPTORS

Fig. 25.21

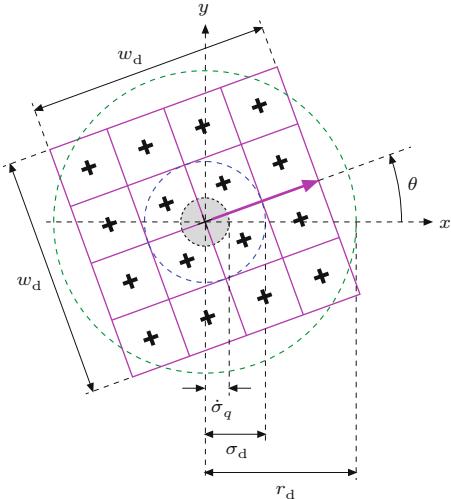
Orientation histograms and dominant orientations (examples). $n = 0, \dots, 3$ smoothing iterations were applied to the orientation histograms. The (interpolated) dominant orientations are shown as radial lines that emanate from each feature's center point. The size of the histogram graphs is proportional to the absolute scale ($\sigma_{p,q}$, see Table 25.3) at which the corresponding key point was detected. The colors indicate the index of the containing scale space octave p (red = 0, green = 1, blue = 2, magenta = 3).

The region is partitioned into $n_{\text{spat}} \times n_{\text{spat}}$ sub-squares of identical size; typically $n_{\text{spat}} = 4$ (see Table 25.5). The contribution of each gradient sample is attenuated by a circular Gaussian function of width $\sigma_d = 0.25 \cdot w_d$ (blue circle). The weights drop off radially and are practically zero at $r_d = 2.5 \cdot \sigma_d$ (green circle in Fig. 25.22). Thus only samples outside this zone need to be included for calculating the descriptor statistics.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.22

Geometry of a SIFT descriptor. The descriptor is calculated from a square support region that is centered at the key point's position (x, y) , aligned to the key point's dominant orientation θ , and partitioned into $n_{\text{spat}} \times n_{\text{spat}}$ (4×4) subsquares. The radius of the inner (gray) circle corresponds to the feature point's decimated scale value ($\dot{\sigma}_q$). The blue circle displays the width (σ_d) of the Gaussian weighting function applied to the gradients; its value is practically zero outside the green circle (r_d).



To achieve rotation invariance, the descriptor region is aligned to the key point's dominant orientation, as determined in the previous steps. To make the descriptor invariant to scale changes, its size w_d (expressed in the grid coordinate units of octave p) is set proportional to the key point's *decimated scale* $\dot{\sigma}_q$ (see Eqn. (25.37)), that is,

$$w_d = s_d \cdot \dot{\sigma}_q = s_d \cdot \sigma_0 \cdot 2^{q/Q}, \quad (25.87)$$

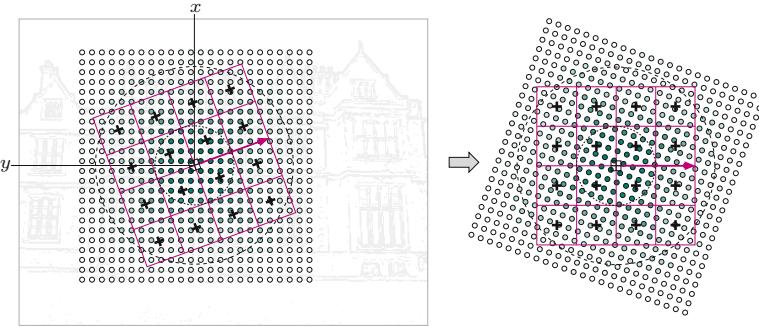
where s_d is a constant size factor. For $s_d = 10$ (see Table 25.5), the descriptor size w_d ranges from 16.0 (at level 0) to 25.4 (at level 2), as listed in Table 25.4. Note that the descriptor size w_d only depends on the scale level index q and is independent of the octave index p . Thus the same descriptor geometry applies to all octaves of the scale space.

Table 25.4

SIFT descriptor dimensions for different scale levels q (for size factor $s_d = 10$ and $Q = 3$ levels per octave). $\dot{\sigma}_q$ is the key point's decimated scale, w_d is the descriptor size, σ_d is the width of the Gaussian weighting function, and r_d is the radius of the descriptor's support region. For $Q = 3$, only scale levels $q = 0, 1, 2$ are relevant. All lengths are expressed in the octave's (i.e., decimated) coordinate units.

q	$\dot{\sigma}_q$	$w_d = s_d \cdot \dot{\sigma}_q$	$\sigma_d = 0.25 \cdot w_d$	$r_d = 2.5 \cdot \sigma_d$
3	3.2000	32.000	8.0000	20.0000
2	2.5398	25.398	6.3495	15.8738
1	2.0159	20.159	5.0398	12.5994
0	1.6000	16.000	4.0000	10.0000
-1	1.2699	12.699	3.1748	7.9369

The descriptor's *spatial resolution* is specified by the parameter n_{spat} . Typically $n_{\text{spat}} = 4$ (as shown in Fig. 25.22) and thus the total number of spatial bins is $n_{\text{spat}} \times n_{\text{spat}} = 16$ (in this case). Each spatial descriptor bin relates to an area of size $(w_d/n_{\text{spat}}) \times (w_d/n_{\text{spat}})$. For example, at scale level $q = 0$ of any octave, $\dot{\sigma}_0 = 1.6$ and the corresponding descriptor size is $w_d = s_d \cdot \dot{\sigma}_0 = 10 \cdot 1.6 = 16.0$ (see Table 25.4). In this case (illustrated in Fig. 25.23), the descriptor covers 16×16 gradient samples, as suggested in [153]. Figure 25.24 shows an example with M-shaped feature point markers aligned to the dominant orientation and scaled to the descriptor region width w_d of the associated scale level.



25.3 CREATING LOCAL DESCRIPTORS

Fig. 25.23

Geometry of the SIFT descriptor in relation to the discrete sample grid of the associated octave (level $q = 0$, parameter $s_d = 10$). In this case, the decimated scale is $\sigma_0 = 1.6$ and the width of the descriptor is $w_d = s_d \cdot \sigma_0 = 10 \cdot 1.6 = 16.0$.

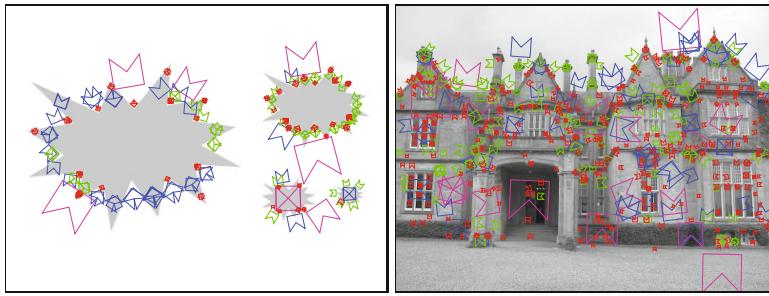


Fig. 25.24

Marked key points aligned to their dominant orientation. Note that multiple feature instances are inserted at key point positions with more than one dominant orientation. The size of the markers is proportional to the absolute scale ($\sigma_{p,q}$, see Table 25.3) at which the corresponding key point was detected. The colors indicate the index of the scale space containing octave p (red = 0, green = 1, blue = 2, magenta = 3).

Gradient features

The actual SIFT descriptor is a feature vector obtained by histogramming the gradient orientations of the Gaussian scale level within the descriptors spatial support region. This requires a 3D histogram $h_\nabla(i, j, k)$, with two spatial dimensions (i, j) for the $n_{\text{spat}} \times n_{\text{spat}}$ sub-regions and one additional dimension (k) for n_{angl} gradient orientations. This histogram thus contains $n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}}$ bins.

Figure 25.25 illustrates this structure for the typical setup, with $n_{\text{spat}} = 4$ and $n_{\text{angl}} = 8$ (see Table 25.5). In this arrangement, eight orientation bins $k = 0, \dots, 7$ are attached to each of the 16 spatial position bins ($A1, \dots, D4$), which makes a total of 128 histogram bins.

For a given key point $\mathbf{k}' = (p, q, x, y)$, the histogram h_∇ accumulates the orientations (angles) of the gradients at the Gaussian scale space level $\mathbf{G}_{p,q}$ within the support region around the (continuous) center coordinate (x, y) . At each grid point (u, v) inside this region, the gradient vector ∇_G is estimated (as described in Eqn. (25.73)), from which the gradient magnitude $E(u, v)$ and orientation $\phi(u, v)$ are calculated (see Eqns. (25.74)–(25.75) and lines 27–31 in Alg. 25.7). For efficiency reasons, $E(u, v)$ and $\phi(u, v)$ are typically pre-calculated for all relevant scale levels.

Each gradient sample contributes to the gradient histogram h_∇ a particular quantity z that depends on the gradient magnitude E and the distance of the sample point (u, v) from the key point's center (x, y) . Again a Gaussian weighting function (of width σ_d) is used to attenuate samples with increasing spatial distance; thus the resulting accumulated quantity is

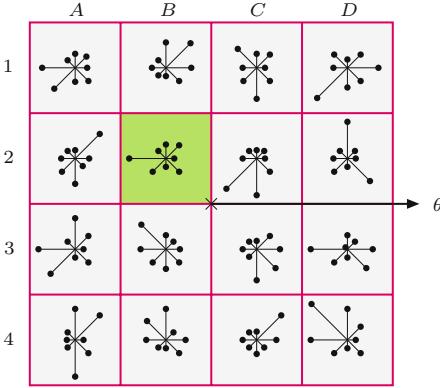
25 SCALE-INARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.25

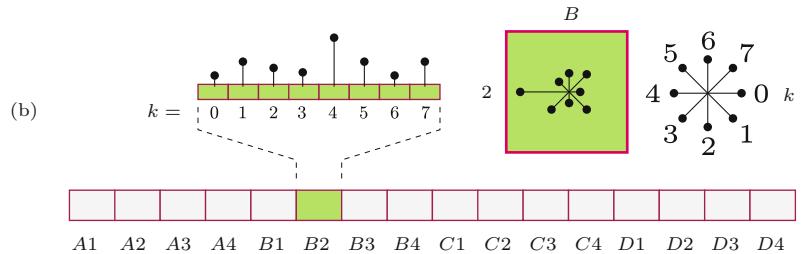
SIFT descriptor structure for $n_{\text{spat}} = 4$ and $n_{\text{angl}} = 8$. Eight orientation bins $k = 0, \dots, 7$ are provided for each of the 16 spatial bins $ij = A1, \dots, D4$.

Thus the gradient histogram h_{∇} holds 128 cells that are arranged to a 1D feature vector $(A1_0, A1_1, \dots, D4_6, D4_7)$ as shown in (b).

(a)



(b)



$$z(u, v) = R(u, v) \cdot w_G = R(u, v) \cdot \exp\left(-\frac{(u-x)^2 + (v-y)^2}{2\sigma_d^2}\right). \quad (25.88)$$

The width σ_d of the Gaussian function $w_G()$ is proportional to the side length of the descriptor region, with

$$\sigma_d = 0.25 \cdot w_d = 0.25 \cdot s_d \cdot \dot{\sigma}_q. \quad (25.89)$$

The weighting function drops off radially from the center and is practically zero at distance $r_d = 2.5 \cdot \sigma_d$. Therefore, only gradient samples that are closer to the key point's center than r_d (green circle in Fig. 25.22) need to be considered in the gradient histogram calculation (see Alg. 25.8, lines 7 and 17). For a given key point $\mathbf{k}' = (p, q, x, y)$, sampling of the Gaussian gradients can thus be confined to the grid points (u, v) inside the square region bounded by $x \pm r_d$ and $y \pm r_d$ (see Alg. 25.8, lines 8–10 and 15–16). Each sample point (u, v) is then subjected to the affine transformation

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \frac{1}{w_d} \cdot \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix} \cdot \begin{pmatrix} u-x \\ v-y \end{pmatrix}, \quad (25.90)$$

which performs a rotation by the dominant orientation θ and maps the original (rotated) square of size $w_d \times w_d$ to the unit square with coordinates $u', v' \in [-0.5, +0.5]$ (see Fig. 25.23).

To make feature vectors rotation invariant, the individual gradient orientations $\phi(u, v)$ are rotated by the dominant orientation, that is,

$$\phi'(u, v) = (\phi(u, v) - \theta) \bmod 2\pi, \quad (25.91)$$

with $\phi'(u, v) \in [0, 2\pi]$, such that the relative orientation is preserved.

For each gradient sample, with the continuous coordinates (u', v', ϕ') , the corresponding quantity $z(u, v)$ (Eqn. (25.88)) is accumulated into the 3D gradient histogram \mathbf{h}_∇ . For a complete description of this step see procedure `UpdateGradientHistogram()` in Alg. 25.9. It first maps the coordinates (u', v', ϕ') (see Eqn. (25.90)) to the continuous histogram position (i', j', k') by

$$\begin{aligned} i' &= n_{\text{spat}} \cdot u' + 0.5 \cdot (n_{\text{spat}} - 1), \\ j' &= n_{\text{spat}} \cdot v' + 0.5 \cdot (n_{\text{spat}} - 1), \\ k' &= \phi' \cdot \frac{n_{\text{angl}}}{2\pi}, \end{aligned} \quad (25.92)$$

such that $i', j' \in [-0.5, n_{\text{spat}} - 0.5]$ and $k' \in [0, n_{\text{angl}}]$.

Analogous to inserting into a continuous position of a 1D histogram by linear interpolation over *two* bins (see Fig. 25.18), the quantity z is distributed over *eight* neighboring histogram bins by *tri-linear* interpolation. The quantiles of z contributing to the individual histogram bins are determined by the distances of the coordinates (i', j', k') from the discrete indexes (i, j, k) of the affected histogram bins. The indexes (i, j, k) are found as the set of possible combinations $\{i_0, i_1\} \times \{j_0, j_1\} \times \{k_0, k_1\}$, with

$$\begin{aligned} i_0 &= \lfloor i' \rfloor, & i_1 &= (i_0 + 1), \\ j_0 &= \lfloor j' \rfloor, & j_1 &= (j_0 + 1), \\ k_0 &= \lfloor k' \rfloor \bmod n_{\text{angl}}, & k_1 &= (k_0 + 1) \bmod n_{\text{angl}}, \end{aligned} \quad (25.93)$$

and the corresponding quantiles (weights) are

$$\begin{aligned} \alpha_0 &= \lfloor i' \rfloor + 1 - i' = i_1 - i', & \alpha_1 &= 1 - \alpha_0, \\ \beta_0 &= \lfloor j' \rfloor + 1 - j' = j_1 - j', & \beta_1 &= 1 - \beta_0, \\ \gamma_0 &= \lfloor k' \rfloor + 1 - k' = k_1 - k', & \gamma_1 &= 1 - \gamma_0, \end{aligned} \quad (25.94)$$

and the (eight) affected bins of the gradient histogram are finally updated as

$$\begin{aligned} \mathbf{h}_\nabla(i_0, j_0, k_0) &\leftarrow^+ z \cdot \alpha_0 \cdot \beta_0 \cdot \gamma_0, \\ \mathbf{h}_\nabla(i_1, j_0, k_0) &\leftarrow^+ z \cdot \alpha_1 \cdot \beta_0 \cdot \gamma_0, \\ \mathbf{h}_\nabla(i_0, j_1, k_0) &\leftarrow^+ z \cdot \alpha_0 \cdot \beta_1 \cdot \gamma_0, \\ &\vdots &&\vdots \\ \mathbf{h}_\nabla(i_1, j_1, k_1) &\leftarrow^+ z \cdot \alpha_1 \cdot \beta_1 \cdot \gamma_1. \end{aligned} \quad (25.95)$$

Attention must be paid to the fact that the coordinate k represents an orientation and must therefore be treated in a *circular* manner, as illustrated in Fig. 25.26 (also see Alg. 25.9, lines 11–12).

For each histogram bin, the range of contributing gradient samples covers half of each neighboring bin, that is, the support regions of neighboring bins overlap, as illustrated in Fig. 25.27.

Normalizing SIFT descriptors

The elements of the gradient histogram \mathbf{h}_∇ are the raw material for the SIFT feature vectors \mathbf{f}_{sift} . The process of calculating the feature vectors from the gradient histogram is described in Alg. 25.10.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.26

3D structure of the gradient histogram, with $n_{\text{spat}} \times n_{\text{spat}} = 4 \times 4$ bins for the spatial dimensions (i, j) and $n_{\text{angl}} = 8$ bins along the orientation axis (k) . For the histogram to accumulate a quantity z into some continuous position (i', j', k') , eight adjacent bins receive different quantiles of z that are determined by tri-linear interpolation (a). Note that the bins along the orientation axis ϕ are treated circularly; for example, bins at $k = 0$ are also considered adjacent to the bins at $k = 7$ (b).

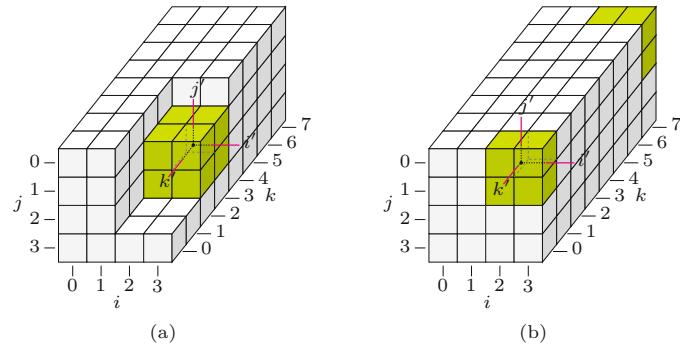
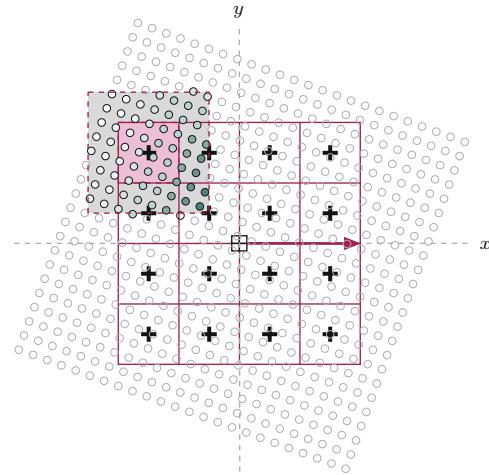


Fig. 25.27

Overlapping support regions in the gradient field. Due to the tri-linear interpolation used in the histogram calculation, the spatial regions associated with the cells of the orientation histogram h_{∇} overlap. The shading of the circles indicates the weight w_G assigned to each sample by the Gaussian weighting function, whose value depends on the distance of each sample from the key point's center (see Eqn. (25.88)).



Initially, the 3D gradient histogram h_{∇} (which contains continuous values) of size $n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}}$ is flattened to a 1D vector \mathbf{f} of length $n_{\text{spat}}^2 \cdot n_{\text{angl}}$ (typ. 128), with

$$\mathbf{f}((i \cdot n_{\text{spat}} + j) \cdot n_{\text{angl}} + k) \leftarrow h_{\nabla}(i, j, k), \quad (25.96)$$

for $i, j = 0, \dots, n_{\text{spat}} - 1$ and $k = 0, \dots, n_{\text{angl}} - 1$. The elements in \mathbf{f} are thus arranged in the same order as shown in Fig. 25.25, with the orientation index k being the fastest moving and the spatial index i being the slowest (see Alg. 25.10, lines 3–8).²⁴

Changes in image contrast have a linear impact upon the gradient magnitude and thus also upon the values of the feature vector \mathbf{f} . To eliminate these effects, the vector \mathbf{f} is subsequently normalized to

$$\mathbf{f}(m) \leftarrow \frac{1}{\|\mathbf{f}\|} \cdot \mathbf{f}(m), \quad (25.97)$$

for all m , such that \mathbf{f} has unit norm (see Alg. 25.10, line 9). Since the gradient is calculated from local pixel differences, changes in absolute

²⁴ Note that different ordering schemes for arranging the elements of the feature vector are used in various SIFT implementations. For successful matching, the ordering of the elements must be identical, of course.

brightness do not affect the gradient magnitude, unless saturation occurs. Such nonlinear illumination changes tend to produce peak gradient values, which are compensated for by clipping the values of \mathbf{f} to a predefined maximum t_{fclip} , that is,

$$\mathbf{f}(m) \leftarrow \min(\mathbf{f}(m), t_{\text{fclip}}), \quad (25.98)$$

with typically $t_{\text{fclip}} = 0.2$, as suggested in [153] (see Alg. 25.10, line 10). After this step, \mathbf{f} is normalized once again, as in Eqn. (25.97). Finally, the real-valued feature vector \mathbf{f} is converted to an integer vector by

$$\mathbf{f}_{\text{sift}}(m) \leftarrow \min(\text{round}(\mathbf{s}_{\text{fscale}} \cdot \mathbf{f}(m)), 255), \quad (25.99)$$

with $\mathbf{s}_{\text{fscale}}$ being a predefined constant (typ. $\mathbf{s}_{\text{fscale}} = 512$). The elements of \mathbf{f}_{sift} are in the range $[0, 255]$ to be conveniently encoded and stored as a byte sequence (see Alg. 25.10, line 12).

The final SIFT descriptor for a given key point $\mathbf{k}' = (p, q, x, y)$ is a tuple

$$\mathbf{s} = \langle x', y', \sigma, \theta, \mathbf{f}_{\text{sift}} \rangle, \quad (25.100)$$

which contains the key point's interpolated position x', y' (in original image coordinates), the absolute scale σ , its dominant orientation θ , and the corresponding integer-valued gradient feature vector \mathbf{f}_{sift} (see Alg. 25.8, line 27). Remember that multiple SIFT descriptors may be produced for different dominant orientations located at the same key point position. These will have the same position and scale values but different θ and \mathbf{f}_{sift} data.

25.4 SIFT Algorithm Summary

This section contains a collection of algorithms that summarizes the SIFT feature extraction process described in the previous sections of this chapter.

Algorithm 25.3 shows the top-level procedure `GetSiftFeatures(I)`, which returns a sequence of SIFT feature descriptors for the given image I . The remaining parts of Alg. 25.3 describe the key point detection as extrema of the DOG scale space. The refinement of key point positions is covered in Alg. 25.4. Algorithm 25.5 contains the procedures used for neighborhood operations, detecting local extrema, and the calculation of the gradient and Hessian matrix in 3D. Algorithm 25.6 covers the operations related to finding the dominant orientations at a given key point location, based on the orientation histogram that is calculated in Alg. 25.7. The final formation of the SIFT descriptors is described in Alg. 25.8, which is based on the procedures defined in Algs. 25.9 and 25.10. The global constants used throughout these algorithms are listed in [Table 25.5](#), together with the corresponding Java identifiers in the associated source code (see Sec. 25.7).

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Table 25.5
Predefined constants used in the SIFT algorithms (Algs. 25.3–25.11).

Scale space parameters			
Symbol	Java id.	Value	Description
Q	Q	3	scale steps (levels) per octave
P	P	4	number of scale space octaves
σ_s	sigma_s	0.5	sampling scale (nominal smoothing of the input image)
σ_0	sigma_0	1.6	base scale of level 0 (base smoothing)

Key-point detection

Symbol	Java id.	Value	Description
n_{orient}	n_Orient	36	number of orientation bins (angular resolution) used for calculating the dominant key point orientation
n_{refine}	n_Refine	5	max. number of iterations for repositioning a key point
n_{smooth}	n_Smooth	2	number of smoothing iterations applied to the orientation histogram
ρ_{max}	rho_Max	10.0	max. ratio of principal curvatures (3, ..., 10)
t_{domor}	t_DomOr	0.8	min. value in orientation histogram for selecting dominant orientations (rel. to max. entry)
t_{extrm}	t_Extrm	0.0	min. difference w.r.t. any neighbor for extrema detection
t_{mag}	t_Mag	0.01	min. DoG magnitude for initial key point candidates
t_{peak}	t_Peak	0.01	min. DoG magnitude at interpolated peaks

Feature descriptor

Symbol	Java id.	Value	Description
n_{spat}	n_Spat	4	number of spatial descriptor bins along each x/y axis
n_{angl}	n_Angl	16	number of angular descriptor bins
s_d	s_Desc	10.0	spatial size factor of descriptor (relative to feature scale)
s_{fscale}	s_Fscale	512.0	scale factor for converting normalized feature values to byte values in [0, 255]
t_{fclip}	t_Fclip	0.2	max. value for clipping elements of normalized feature vectors

Feature matching

Symbol	Java id.	Value	Description
ρ_{max}	rho_ax	0.8	max. ratio of best and second-best matching feature distance

25.5 Matching SIFT Features

Most applications of SIFT features aim at locating corresponding interest points in two or more images of the same scene, for example, for matching stereo pairs, panorama stitching, or feature tracking. Other applications like self-localization or object recognition might use a large database of model descriptors and the task is to match these to the SIFT features detected in a new image or video sequence. All these applications require possibly large numbers of pairs of SIFT features to be compared reliably and efficiently.

25.5.1 Feature Distance and Match Quality

In a typical situation, two sequences of SIFT features $S^{(a)}$ and $S^{(b)}$ are extracted independently from a pair of input images I_a, I_b , that is,

$$S^{(a)} = (\mathbf{s}_1^{(a)}, \mathbf{s}_2^{(a)}, \dots, \mathbf{s}_{N_a}^{(a)}) \quad \text{and} \quad S^{(b)} = (\mathbf{s}_1^{(b)}, \mathbf{s}_2^{(b)}, \dots, \mathbf{s}_{N_b}^{(b)}).$$

The goal is to find matching descriptors in the two feature sets. The similarity between a given pair of descriptors, $\mathbf{s}_i = \langle x_i, y_i, \sigma_i, \theta_i, \mathbf{f}_i \rangle$ and $\mathbf{s}_j = \langle x_j, y_j, \sigma_j, \theta_j, \mathbf{f}_j \rangle$, is measured by the *distance* between the corresponding feature vectors $\mathbf{f}_i, \mathbf{f}_j$, that is,

```

1: GetSiftFeatures( $I$ )
   Input:  $I$ , the source image (scalar-valued).
   Returns a sequence of SIFT feature descriptors detected in  $I$ .
2:  $\langle \mathbf{G}, \mathbf{D} \rangle \leftarrow \text{BuildSiftScaleSpace}(I, \sigma_s, \sigma_0, P, Q)$            ▷ Alg. 25.2
3:  $C \leftarrow \text{GetKeyPoints}(\mathbf{D})$ 
4:  $S \leftarrow ()$                                 ▷ empty list of SIFT descriptors
5: for all  $k' \in C$  do                      ▷  $k' = (p, q, x, y)$ 
6:    $A \leftarrow \text{GetDominantOrientations}(\mathbf{G}, k')$                          ▷ Alg. 25.6
7:   for all  $\theta \in A$  do
8:      $s \leftarrow \text{MakeSiftDescriptor}(\mathbf{G}, k', \theta)$                          ▷ Alg. 25.8
9:      $S \leftarrow S \cup (s)$ 
10: return  $S$ 

11: GetKeypoints( $\mathbf{D}$ )
     $\mathbf{D}$ : DoG scale space (with  $P$  octaves, each containing  $Q$  levels).
    Returns a set of key points located in  $\mathbf{D}$ .
12:  $C \leftarrow ()$                                 ▷ empty list of key points
13: for  $p \leftarrow 0, \dots, P-1$  do          ▷ for all octaves  $p$ 
14:   for  $q \leftarrow 0, \dots, Q-1$  do          ▷ for all scale levels  $q$ 
15:      $E \leftarrow \text{FindExtrema}(\mathbf{D}, p, q)$ 
16:     for all  $k \in E$  do                  ▷  $k = (p, q, u, v)$ 
17:        $k' \leftarrow \text{RefineKeyPosition}(\mathbf{D}, k)$                          ▷ Alg. 25.4
18:       if  $k' \neq \text{nil}$  then            ▷  $k' = (p, q, x, y)$ 
19:          $C \leftarrow C \cup (k')$            ▷ add refined key point  $k'$ 
20: return  $C$ 

21: FindExtrema( $\mathbf{D}, p, q$ )
22:  $\mathbf{D}_{p,q} \leftarrow \text{GetScaleLevel}(\mathbf{D}, p, q)$ 
23:  $(M, N) \leftarrow \text{Size}(\mathbf{D}_{p,q})$ 
24:  $E \leftarrow ()$                                 ▷ empty list of extrema
25: for  $u \leftarrow 1, \dots, M-2$  do
26:   for  $v \leftarrow 1, \dots, N-2$  do
27:     if  $|\mathbf{D}_{p,q}(u, v)| > t_{\text{mag}}$  then
28:        $k \leftarrow (p, q, u, v)$ 
29:        $N_c \leftarrow \text{GetNeighborhood}(\mathbf{D}, k)$                          ▷ Alg. 25.5
30:       if  $\text{IsExtremum}(N_c)$  then           ▷ Alg. 25.5
31:          $E \leftarrow E \cup (k)$            ▷ add  $k$  to  $E$ 
32: return  $E$ 

```

$$\text{dist}(\mathbf{s}_i, \mathbf{s}_j) := \|\mathbf{f}_i - \mathbf{f}_j\|, \quad (25.101)$$

where $\|\cdot\cdot\cdot\|$ denotes an appropriate norm (typically Euclidean, alternatives will be discussed further).²⁵

Note that this distance is measured between individual points distributed in a high-dimensional (typically 128-dimensional) vector space that is only sparsely populated. Since there is *always* a best-matching counterpart for a given descriptor, matches may occur between unrelated features even if the correct feature is not contained in the target set. This is particularly critical if feature matching is used to determine whether two images show any correspondence at all.

Obviously, significant matches should exhibit small feature distances but setting a *fixed limit* on the acceptable feature distance

25.5 MATCHING SIFT FEATURES

Alg. 25.3

SIFT feature extraction (part 1). Top-level SIFT procedure. Global parameters: $\sigma_s, \sigma_0, t_{\text{mag}}, Q, P$ (see Table 25.5).

²⁵ See also Sec. B.1.2 in the Appendix.

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Alg. 25.4
SIFT feature extraction
(part 2). Position refinement.
Global parameters: n_{refine} ,
 t_{peak} , ρ_{max} (see Table 25.5).

1: **RefineKeyPosition(\mathbf{D}, \mathbf{k})**
Input: \mathbf{D} , hierarchical DoG scale space; $\mathbf{k} = (p, q, u, v)$, candidate (extremal) position.
Returns a refined key point \mathbf{k}' or nil if no proper key point could be localized at or near the extremal position \mathbf{k} .

```

2:    $a_{\text{max}} \leftarrow \frac{(\rho_{\text{max}}+1)^2}{\rho_{\text{max}}}$                                 ▷ see Eq. 25.72
3:    $\mathbf{k}' \leftarrow \text{nil}$                                          ▷ refined key point
4:    $done \leftarrow \text{false}$ 
5:    $n \leftarrow 1$                                               ▷ number of repositioning steps
6:   while  $\neg done \wedge n \leq n_{\text{refine}} \wedge \text{IsInside}(\mathbf{D}, \mathbf{k})$  do
7:      $\mathbf{N}_c \leftarrow \text{GetNeighborhood}(\mathbf{D}, \mathbf{k})$                       ▷ Alg. 25.5
8:      $\nabla = \begin{pmatrix} d_x \\ d_x \\ d_\sigma \end{pmatrix} \leftarrow \text{Gradient}(\mathbf{N}_c)$       ▷ Alg. 25.5
9:      $\mathbf{H}_D = \begin{pmatrix} d_{xx} & d_{xy} & d_{x\sigma} \\ d_{xy} & d_{yy} & d_{y\sigma} \\ d_{x\sigma} & d_{y\sigma} & d_{\sigma\sigma} \end{pmatrix} \leftarrow \text{Hessian}(\mathbf{N}_c)$       ▷ Alg. 25.5
10:    if  $\det(\mathbf{H}_D) = 0$  then                               ▷  $\mathbf{H}_D$  is not invertible
11:       $done \leftarrow \text{true}$                          ▷ ignore this point and finish
12:    else
13:       $\mathbf{d} = \begin{pmatrix} x' \\ y' \\ \sigma' \end{pmatrix} \leftarrow -\mathbf{H}_D^{-1} \cdot \nabla$           ▷ Eq. 25.60
14:      if  $|x'| < 0.5 \wedge |y'| < 0.5$  then ▷ stay in the same DoG cell
15:         $done \leftarrow \text{true}$ 
16:         $D_{\text{peak}} \leftarrow \mathbf{N}_c(0, 0, 0) + \frac{1}{2} \cdot \nabla^\top \cdot \mathbf{d}$           ▷ Eq. 25.61
17:         $\mathbf{H}_{xy} \leftarrow \begin{pmatrix} d_{xx} & d_{xy} \\ d_{xy} & d_{yy} \end{pmatrix}$           ▷ extract 2D Hessian from  $\mathbf{H}_D$ 
18:        if  $|D_{\text{peak}}| > t_{\text{peak}} \wedge \det(\mathbf{H}_{xy}) > 0$  then
19:           $a \leftarrow \frac{[\text{trace}(\mathbf{H}_{xy})]^2}{\det(\mathbf{H}_{xy})}$           ▷ Eq. 25.69
20:          if  $a < a_{\text{max}}$  then          ▷ suppress edges, Eq. 25.72
21:             $\mathbf{k}' \leftarrow \mathbf{k} + (0, 0, x', y')^\top$           ▷ refined key point
22:          else
23:            Move to a neighboring DoG position at same level  $p, q$ :
24:             $u' \leftarrow \min(1, \max(-1, \text{round}(x'))) \triangleright$  move by max.  $\pm 1$ 
25:             $v' \leftarrow \min(1, \max(-1, \text{round}(y'))) \triangleright$  move by max.  $\pm 1$ 
26:             $\mathbf{k} \leftarrow \mathbf{k} + (0, 0, u', v')^\top$ 
27:   return  $\mathbf{k}'$           ▷  $\mathbf{k}'$  is either a refined key point position or nil

```

turns out to be inappropriate in practice, since some descriptors are more discriminative than others. The solution proposed in [153] is to compare the distance obtained for the *best* feature match to that of the *second-best* match. For a given reference descriptor $\mathbf{s}_r \in S^{(a)}$, the best match is defined as the descriptor $\mathbf{s}_1 \in S^{(b)}$ which has the smallest distance from \mathbf{s}_r in the multi-dimensional feature space, that is,

$$\mathbf{s}_1 = \underset{\mathbf{s}_j \in S^{(b)}}{\operatorname{argmin}} \operatorname{dist}(\mathbf{s}_r, \mathbf{s}_j), \quad (25.102)$$

```

1: IsInside( $\mathbf{D}, k$ )                                ▷ Checks if coordinate  $k = (p, q, u, v)$  is inside the DoG scale space
    $\mathbf{D}$ .
2:  $(p, q, u, v) \leftarrow k$ 
3:  $(M, N) \leftarrow \text{Size}(\text{GetScaleLevel}(\mathbf{D}, p, q))$ 
4: return  $(0 < u < M-1) \wedge (0 < v < N-1) \wedge (0 \leq q < Q)$ 


---


5: GetNeighborhood( $\mathbf{D}, k$ )                      ▷  $k = (p, q, u, v)$ 
   Collects and returns the  $3 \times 3 \times 3$  neighborhood values around
   position  $k$  in the hierarchical DoG scale space  $\mathbf{D}$ .
6: Create map  $\mathbf{N}_c : \{-1, 0, 1\}^3 \mapsto \mathbb{R}$ 
7: for all  $(i, j, k) \in \{-1, 0, 1\}^3$  do ▷ collect  $3 \times 3 \times 3$  neighborhood
8:  $\mathbf{N}_c(i, j, k) \leftarrow \mathbf{D}_{p, q+k}(u+i, v+j)$ 
9: return  $\mathbf{N}_c$ 


---


10: IsExtremum( $\mathbf{N}_c$ )                           ▷  $\mathbf{N}_c$  is a  $3 \times 3 \times 3$  map
    Determines if the center of the 3D neighborhood  $\mathbf{N}_c$  is either a
    local minimum or maximum by the threshold  $t_{\text{extrm}} \geq 0$ . Returns
    a boolean value (i.e., true or false).
11:  $c \leftarrow \mathbf{N}_c(0, 0, 0)$                       ▷ center DoG value
12:  $isMin \leftarrow c < 0 \wedge (c + t_{\text{extrm}}) < \min_{\substack{(i, j, k) \neq \\ (0, 0, 0)}} \mathbf{N}_c(i, j, k)$     ▷ s. Eq. 25.54
13:  $isMax \leftarrow c > 0 \wedge (c - t_{\text{extrm}}) > \max_{\substack{(i, j, k) \neq \\ (0, 0, 0)}} \mathbf{N}_c(i, j, k)$     ▷ s. Eq. 25.55
14: return  $isMin \vee isMax$ 


---


15: Gradient( $\mathbf{N}_c$ )                           ▷  $\mathbf{N}_c$  is a  $3 \times 3 \times 3$  map
    Returns the estim. gradient vector ( $\nabla$ ) for the 3D neighborhood
     $\mathbf{N}_c$ .
16:  $d_x \leftarrow 0.5 \cdot (\mathbf{N}_c(1, 2, 1) - \mathbf{N}_c(1, 0, 1))$ 
17:  $d_y \leftarrow 0.5 \cdot (\mathbf{N}_c(1, 1, 2) - \mathbf{N}_c(1, 1, 0))$           ▷ see Eq. 25.56
18:  $d_\sigma \leftarrow 0.5 \cdot (\mathbf{N}_c(2, 1, 1) - \mathbf{N}_c(0, 1, 1))$ 
19:  $\nabla \leftarrow (d_x, d_y, d_\sigma)^\top$ 
20: return  $\nabla$ 


---


21: Hessian( $\mathbf{N}_c$ )                           ▷  $\mathbf{N}_c$  is a  $3 \times 3 \times 3$  map
    Returns the estim. Hessian matrix ( $\mathbf{H}$ ) for the neighborhood  $\mathbf{N}_c$ .
22:  $d_{xx} \leftarrow \mathbf{N}_c(-1, 0, 0) - 2 \cdot \mathbf{N}_c(0, 0, 0) + \mathbf{N}_c(1, 0, 0)$     ▷ see Eq. 25.58
23:  $d_{yy} \leftarrow \mathbf{N}_c(0, -1, 0) - 2 \cdot \mathbf{N}_c(0, 0, 0) + \mathbf{N}_c(0, 1, 0)$ 
24:  $d_{\sigma\sigma} \leftarrow \mathbf{N}_c(0, 0, -1) - 2 \cdot \mathbf{N}_c(0, 0, 0) + \mathbf{N}_c(0, 0, 1)$ 
25:  $d_{xy} \leftarrow [\mathbf{N}_c(1, 1, 0) - \mathbf{N}_c(-1, 1, 0) - \mathbf{N}_c(1, -1, 0) + \mathbf{N}_c(-1, -1, 0)] / 4$ 
26:  $d_{x\sigma} \leftarrow [\mathbf{N}_c(1, 0, 1) - \mathbf{N}_c(-1, 0, 1) - \mathbf{N}_c(1, 0, -1) + \mathbf{N}_c(-1, 0, -1)] / 4$ 
27:  $d_{y\sigma} \leftarrow [\mathbf{N}_c(0, 1, 1) - \mathbf{N}_c(0, -1, 1) - \mathbf{N}_c(0, 1, -1) + \mathbf{N}_c(0, -1, -1)] / 4$ 


---


28: 
$$\mathbf{H} \leftarrow \begin{pmatrix} d_{xx} & d_{xy} & d_{x\sigma} \\ d_{xy} & d_{yy} & d_{y\sigma} \\ d_{x\sigma} & d_{y\sigma} & d_{\sigma\sigma} \end{pmatrix}$$

29: return  $\mathbf{H}$ 

```

and the primary distance is $d_{r,1} = \text{dist}(\mathbf{s}_r, \mathbf{s}_1)$. Analogously, the second-best matching descriptor is

$$\mathbf{s}_2 = \underset{\substack{\mathbf{s}_j \in S^{(b)} \\ \mathbf{s}_j \neq \mathbf{s}_1}}{\operatorname{argmin}} \text{dist}(\mathbf{s}_r, \mathbf{s}_j), \quad (25.103)$$

and the corresponding distance is $d_{r,2} = \text{dist}(\mathbf{s}_r, \mathbf{s}_2)$, with $d_{r,1} \leq d_{r,2}$. Reliable matches are expected to have a distance to the primary

25.5 MATCHING SIFT FEATURES

Alg. 25.5

SIFT feature extraction
(part 3): Neighborhood operations. Global parameters:
 Q , t_{extrm} (see Table 25.5).

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Alg. 25.6
SIFT feature extraction (part 4): Key point orientation assignment. Global parameters: n_{smooth} , t_{domor} (see Table 25.5).

```

1: GetDominantOrientations( $\mathbf{G}, k'$ )
   Input:  $\mathbf{G}$ , hierarchical Gaussian scale space;  $k' = (p, q, x, y)$ , refined key point at octave  $p$ , scale level  $q$  and spatial position  $x, y$  (in octave's coordinates).
   Returns a list of dominant orientations for the key point  $k'$ .
2:  $\mathbf{h}_\phi \leftarrow \text{GetOrientationHistogram}(\mathbf{G}, k')$                                 ▷ Alg. 25.7
3:  $\text{SmoothCircular}(\mathbf{h}_\phi, n_{\text{smooth}})$ 
4:  $A \leftarrow \text{FindPeakOrientations}(\mathbf{h}_\phi)$ 
5: return  $A$ 
6: SmoothCircular( $\mathbf{x}, n_{\text{iter}}$ )
   Smooths the real-valued vector  $\mathbf{x} = (x_0, \dots, x_{n-1})$  circularly using the 3-element kernel  $H = (h_0, h_1, h_2)$ , with  $h_1$  as the hot-spot. The filter operation is applied  $n_{\text{iter}}$  times and “in place”, i.e., the vector  $\mathbf{x}$  is modified.
7:  $(h_0, h_1, h_2) \leftarrow \frac{1}{4} \cdot (1, 2, 1)$                                          ▷ 1D filter kernel
8:  $n \leftarrow \text{Size}(\mathbf{x})$ 
9: for  $i \leftarrow 1, \dots, n_{\text{iter}}$  do
10:    $s \leftarrow \mathbf{x}(0)$ 
11:    $p \leftarrow \mathbf{x}(n-1)$ 
12:   for  $j \leftarrow 0, \dots, n-2$  do
13:      $c \leftarrow \mathbf{x}(j)$ 
14:      $\mathbf{x}(j) \leftarrow h_0 \cdot p + h_1 \cdot \mathbf{x}(j) + h_2 \cdot \mathbf{x}(j+1)$ 
15:      $p \leftarrow c$ 
16:    $\mathbf{x}(n-1) \leftarrow h_0 \cdot p + h_1 \cdot \mathbf{x}(n-1) + h_2 \cdot s$ 
17: return
18: FindPeakOrientations( $\mathbf{h}_\phi$ )
   Returns a (possibly empty) sequence of dominant directions (angles) obtained from the orientation histogram  $\mathbf{h}_\phi$ .
19:  $n \leftarrow \text{Size}(\mathbf{h}_\phi)$ 
20:  $A \leftarrow ()$ 
21:  $h_{\max} \leftarrow \max_{0 \leq i < n} \mathbf{h}_\phi(i)$ 
22: for  $k \leftarrow 0, \dots, n-1$  do
23:    $h_c \leftarrow \mathbf{h}(k)$ 
24:   if  $h_c > t_{\text{domor}} \cdot h_{\max}$  then    ▷ only accept dominant peaks
25:      $h_p \leftarrow \mathbf{h}_\phi((k-1) \bmod n)$ 
26:      $h_n \leftarrow \mathbf{h}_\phi((k+1) \bmod n)$ 
27:     if  $(h_c > h_p) \wedge (h_c > h_n)$  then    ▷ local max. at index  $k$ 
28:        $\check{k} \leftarrow k + \frac{h_p - h_n}{2 \cdot (h_p - 2 \cdot h_c + h_n)}$  ▷ quadr. interpol., Eq. 25.85
29:        $\theta \leftarrow (\check{k} \cdot \frac{2\pi}{n}) \bmod 2\pi$  ▷ domin. orientation, Eq. 25.86
30:        $A \leftarrow A \cup (\theta)$ 
31: return  $A$ 

```

feature \mathbf{s}_1 that is considerably smaller than the distance to any other feature in the target set. In the case of a weak or ambiguous match, on the other hand, it is likely that other matches exist at a distance similar to $d_{r,1}$, including the second-best match \mathbf{s}_2 . Comparing the best and the second-best distances thus provides information about the likelihood of a false match. For this purpose, we define the *feature distance ratio*

$$\rho_{\text{match}}(\mathbf{s}_r, \mathbf{s}_1, \mathbf{s}_2) := \frac{d_{r,1}}{d_{r,2}} = \frac{\text{dist}(\mathbf{s}_r, \mathbf{s}_1)}{\text{dist}(\mathbf{s}_r, \mathbf{s}_2)}, \quad (25.104)$$

1: **GetOrientationHistogram(\mathbf{G}, k')**

Input: \mathbf{G} , hierarchical Gaussian scale space; $k' = (p, q, x, y)$, refined key point at octave p , scale level q and relative position x, y .

Returns the gradient orientation histogram for key point k' .

```

2:  $\mathbf{G}_{p,q} \leftarrow \text{GetScaleLevel}(\mathbf{G}, p, q)$ 
3:  $(M, N) \leftarrow \text{Size}(\mathbf{G}_{p,q})$ 
4: Create a new map  $\mathbf{h}_\phi : [0, n_{\text{orient}} - 1] \mapsto \mathbb{R}$ .  $\triangleright$  new histogram  $\mathbf{h}_\phi$ 
5: for  $i \leftarrow 0, \dots, n_{\text{orient}} - 1$  do  $\triangleright$  initialize  $\mathbf{h}_\phi$  to zero
6:    $\mathbf{h}_\phi(i) \leftarrow 0$ 
7:  $\sigma_w \leftarrow 1.5 \cdot \sigma_0 \cdot 2^{q/Q}$   $\triangleright \sigma$  of Gaussian weight fun., see Eq. 25.76
8:  $r_w \leftarrow \max(1, 2.5 \cdot \sigma_w)$   $\triangleright$  rad. of weight fun., see Eq. 25.77
9:  $u_{\min} \leftarrow \max(\lfloor x - r_w \rfloor, 1)$ 
10:  $u_{\max} \leftarrow \min(\lceil x + r_w \rceil, M - 2)$ 
11:  $v_{\min} \leftarrow \max(\lfloor y - r_w \rfloor, 1)$ 
12:  $v_{\max} \leftarrow \min(\lceil y + r_w \rceil, N - 2)$ 
13: for  $u \leftarrow u_{\min}, \dots, u_{\max}$  do
14:   for  $v \leftarrow v_{\min}, \dots, v_{\max}$  do
15:      $r^2 \leftarrow (u - x)^2 + (v - y)^2$ 
16:     if  $r^2 < r_w^2$  then
17:        $(E, \phi) \leftarrow \text{GetGradientPolar}(\mathbf{G}_{p,q}, u, v)$   $\triangleright$  see below
18:        $w_G \leftarrow \exp\left(-\frac{(u-x)^2 + (v-y)^2}{2\sigma_w^2}\right)$   $\triangleright$  Gaussian weight
19:        $z \leftarrow E \cdot w_G$   $\triangleright$  quantity to accumulate
20:        $\kappa_\phi \leftarrow \frac{n_{\text{orient}}}{2\pi} \cdot \phi$   $\triangleright \kappa_\phi \in [-\frac{n_{\text{orient}}}{2}, +\frac{n_{\text{orient}}}{2}]$ 
21:        $\alpha \leftarrow \kappa_\phi - \lfloor \kappa_\phi \rfloor$   $\triangleright \alpha \in [0, 1]$ 
22:        $k_0 \leftarrow \lfloor \kappa_\phi \rfloor \bmod n_{\text{orient}}$   $\triangleright$  lower bin index
23:        $k_1 \leftarrow (k_0 + 1) \bmod n_{\text{orient}}$   $\triangleright$  upper bin index
24:        $\mathbf{h}_\phi(k_0) \leftarrow (1 - \alpha) \cdot z$   $\triangleright$  update bin  $k_0$ 
25:        $\mathbf{h}_\phi(k_1) \leftarrow \alpha \cdot z$   $\triangleright$  update bin  $k_1$ 
26: return  $\mathbf{h}_\phi$ 

```

27: **GetGradientPolar($\mathbf{G}_{p,q}, u, v$)**

Returns the gradient magnitude (E) and orientation (ϕ) at position (u, v) of the Gaussian scale level $\mathbf{G}_{p,q}$.

```

28:  $\begin{pmatrix} d_x \\ d_y \end{pmatrix} \leftarrow 0.5 \cdot \begin{pmatrix} \mathbf{G}_{p,q}(u+1, v) - \mathbf{G}_{p,q}(u-1, v) \\ \mathbf{G}_{p,q}(u, v+1) - \mathbf{G}_{p,q}(u, v-1) \end{pmatrix}$   $\triangleright$  gradient at  $u, v$ 
29:  $E \leftarrow (d_x^2 + d_y^2)^{1/2}$   $\triangleright$  gradient magnitude
30:  $\phi \leftarrow \text{ArcTan}(d_x, d_y)$   $\triangleright$  gradient orientation ( $-\pi \leq \phi \leq \pi$ )
31: return  $(E, \phi)$ 

```

such that $\rho_{\text{match}} \in [0, 1]$. If the distance $d_{r,1}$ between s_r and the primary feature s_1 is small compared to the secondary distance $d_{r,2}$, then the value of ρ_{match} is small as well. Thus, large values of ρ_{match} indicate that the corresponding match (between s_r and s_1) is likely to be weak or ambiguous. Matches are only accepted if they are sufficiently distinctive, for example, by enforcing the condition

$$\rho_{\text{match}}(s_r, s_1, s_2) \leq \rho_{\text{max}}, \quad (25.105)$$

where $\rho_{\text{max}} \in [0, 1]$ is a predefined constant (see Table 25.5). The complete matching process, using the Euclidean distance norm and sequential search, is summarized in Alg. 25.11. Other common options for distance measurement are the L_1 and L_∞ norms.

25.5 MATCHING SIFT FEATURES

Alg. 25.7

SIFT feature extraction (part 5): Calculation of the orientation histogram and gradients from Gaussian scale levels. Global parameters: n_{orient} (see Table 25.5).

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Alg. 25.8
SIFT feature extraction (part 6): Calculation of SIFT descriptors. Global parameters: Q , σ_0 , s_d , n_{spat} , n_{angl} (see Table 25.5).

```

1: MakeSiftDescriptor( $\mathbf{G}, k', \theta$ )
   Input:  $\mathbf{G}$ , hierarchical Gaussian scale space;  $k' = (p, q, x, y)$ , refined key point;  $\theta$ , dominant orientation.
   Returns a new SIFT descriptor for the key point  $k'$ .
2:  $\mathbf{G}_{p,q} \leftarrow \text{GetScaleLevel}(\mathbf{G}, p, q)$ 
3:  $(M, N) \leftarrow \text{Size}(\mathbf{G}_{p,q})$ 
4:  $\dot{\sigma}_q \leftarrow \sigma_0 \cdot 2^{q/Q}$                                  $\triangleright$  decimated scale at level  $q$ 
5:  $w_d \leftarrow s_d \cdot \dot{\sigma}_q$                                  $\triangleright$  descriptor size is prop. to key point scale
6:  $\sigma_d \leftarrow 0.25 \cdot w_d$                                  $\triangleright$  width of Gaussian weighting function
7:  $r_d \leftarrow 2.5 \cdot \sigma_d$                                  $\triangleright$  cutoff radius of weighting function
8:  $u_{\min} \leftarrow \max(|x - r_d|, 1)$ 
9:  $u_{\max} \leftarrow \min(|x + r_d|, M - 2)$ 
10:  $v_{\min} \leftarrow \max(|y - r_d|, 1)$ 
11:  $v_{\max} \leftarrow \min(|y + r_d|, N - 2)$ 
12: Create map  $\mathbf{h}_\nabla : n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}} \mapsto \mathbb{R}$   $\triangleright$  gradient histogram
      $\mathbf{h}_\nabla$ 
13: for all  $(i, j, k) \in n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}}$  do
14:    $\mathbf{h}_\nabla(i, j, k) \leftarrow 0$                                  $\triangleright$  initialize  $\mathbf{h}_\nabla$  to zero
15:   for  $u \leftarrow u_{\min}, \dots, u_{\max}$  do
16:     for  $v \leftarrow v_{\min}, \dots, v_{\max}$  do
17:        $r^2 \leftarrow (u - x)^2 + (v - y)^2$ 
18:       if  $r^2 < r_d^2$  then
           Map to canonical coord. frame, with  $u', v' \in [-\frac{1}{2}, +\frac{1}{2}]$ :
           
$$\begin{pmatrix} u' \\ v' \end{pmatrix} \leftarrow \frac{1}{w_d} \cdot \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix} \cdot \begin{pmatrix} u - x \\ v - y \end{pmatrix}$$

19:        $(E, \phi) \leftarrow \text{GetGradientPolar}(\mathbf{G}_{p,q}, u, v)$        $\triangleright$  Alg. 25.7
20:        $\phi' \leftarrow (\phi - \theta) \bmod 2\pi$                                  $\triangleright$  normalize gradient angle
21:        $w_G \leftarrow \exp(-\frac{r^2}{2\sigma_d^2})$                                  $\triangleright$  Gaussian weight
22:        $z \leftarrow E \cdot w_G$                                  $\triangleright$  quantity to accumulate
23:        $\text{UpdateGradientHistogram}(\mathbf{h}_\nabla, u', v', \phi', z)$   $\triangleright$  Alg. 25.9
24:    $\mathbf{f}_{\text{sift}} \leftarrow \text{MakeFeatureVector}(\mathbf{h}_\nabla)$                                  $\triangleright$  see Alg. 25.10
25:    $\sigma \leftarrow \sigma_0 \cdot 2^{p+q/Q}$                                  $\triangleright$  absolute scale, Eq. 25.35
26:   
$$\begin{pmatrix} x' \\ y' \end{pmatrix} \leftarrow 2^p \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$
                                 $\triangleright$  real position, Eq. 25.45
27:    $s \leftarrow \langle x', y', \sigma, \theta, \mathbf{f}_{\text{sift}} \rangle$                                  $\triangleright$  create a new SIFT descriptor
28: return  $s$ 

```

25.5.2 Examples

The following examples were calculated on pairs of stereographic images taken at the beginning of the 20th century.²⁶ From each of the two frames of a stereo picture, a sequence of (ca. 1000) SIFT descriptors (marked by blue rectangles) was extracted with identical parameter settings. Matching was done by enumerating all possible descriptor pairs from the left and the right image, calculating their (Euclidean) distance, and showing the 25 closest matches obtained from ca. 1000 detected key points in each frame. Only the

²⁶ The images used in Figs. 25.28–25.31 are historic stereographs made publicly available by the *Library of Congress* (www.loc.gov).

```

1: UpdateGradientHistogram( $\mathbf{h}_\nabla, u', v', \phi', z)$ 
   Input:  $\mathbf{h}_\nabla$ , gradient histogram of size  $n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}}$ , with
           $\mathbf{h}_\nabla(i, j, k) \in \mathbb{R}$ ;  $u', v' \in [-0.5, 0.5]$ , normalized spatial position;
           $\phi' \in [0, 2\pi]$ , normalized gradient orientation;  $z \in \mathbb{R}$ , quantity to
          be accumulated into  $\mathbf{h}_\nabla$ .
   Returns nothing but modifies the histogram  $\mathbf{h}_\nabla$ .
2:  $i' \leftarrow n_{\text{spat}} \cdot u' + 0.5 \cdot (n_{\text{spat}} - 1)$                                  $\triangleright$  see Eq. 25.92
3:  $j' \leftarrow n_{\text{spat}} \cdot v' + 0.5 \cdot (n_{\text{spat}} - 1)$                                  $\triangleright -0.5 \leq i', j' \leq n_{\text{spat}} - 0.5$ 
4:  $k' \leftarrow n_{\text{angl}} \cdot \frac{\phi'}{2\pi}$                                                $\triangleright -\frac{n_{\text{angl}}}{2} \leq k' \leq \frac{n_{\text{angl}}}{2}$ 
5:  $i_0 \leftarrow \lfloor i' \rfloor$ 
6:  $i_1 \leftarrow i_0 + 1$ 
7:  $\mathbf{i} \leftarrow (i_0, i_1)$                                           $\triangleright$  see Eq. 25.93;  $\mathbf{i}(0) = i_0$ ,  $\mathbf{i}(1) = i_1$ 
8:  $j_0 \leftarrow \lfloor j' \rfloor$ 
9:  $j_1 \leftarrow j_0 + 1$ 
10:  $\mathbf{j} \leftarrow (j_0, j_1)$                                           $\triangleright \mathbf{j}(0) = j_0$ ,  $\mathbf{j}(1) = j_1$ 
11:  $k_0 \leftarrow \lfloor k' \rfloor \bmod n_{\text{angl}}$ 
12:  $k_1 \leftarrow (k_0 + 1) \bmod n_{\text{angl}}$ 
13:  $\mathbf{k} \leftarrow (k_0, k_1)$                                           $\triangleright \mathbf{k}(0) = k_0$ ,  $\mathbf{k}(1) = k_1$ 
14:  $\alpha_0 \leftarrow i_1 - i'$                                           $\triangleright$  see Eq. 25.94
15:  $\alpha_1 \leftarrow 1 - \alpha_0$ 
16:  $A \leftarrow (\alpha_0, \alpha_1)$                                           $\triangleright A(0) = \alpha_0$ ,  $A(1) = \alpha_1$ 
17:  $\beta_0 \leftarrow j_1 - j'$ 
18:  $\beta_1 \leftarrow 1 - \beta_0$ 
19:  $B \leftarrow (\beta_0, \beta_1)$                                           $\triangleright B(0) = \beta_0$ ,  $B(1) = \beta_1$ 
20:  $\gamma_0 \leftarrow 1 - (k' - \lfloor k' \rfloor)$ 
21:  $\gamma_1 \leftarrow 1 - \gamma_0$ 
22:  $C \leftarrow (\gamma_0, \gamma_1)$                                           $\triangleright C(0) = \gamma_0$ ,  $C(1) = \gamma_1$ 

Distribute quantity  $z$  among (up to) 8 adjacent histogram bins:
23: for all  $a \in \{0, 1\}$  do
24:    $i \leftarrow \mathbf{i}(a)$ 
25:   if  $(0 \leq i < n_{\text{spat}})$  then
26:      $w_a \leftarrow A(a)$ 
27:     for all  $b \in \{0, 1\}$  do
28:        $j \leftarrow \mathbf{j}(b)$ 
29:       if  $(0 \leq j < n_{\text{spat}})$  then
30:          $w_b \leftarrow B(b)$ 
31:         for all  $c \in \{0, 1\}$  do
32:            $k \leftarrow \mathbf{k}(c)$ 
33:            $w_c \leftarrow C(c)$ 
34:            $\mathbf{h}_\nabla(i, j, k) \leftarrow^+ z \cdot w_a \cdot w_b \cdot w_c$        $\triangleright$  see Eq. 25.95
35: return

```

25.5 MATCHING SIFT FEATURES

Alg. 25.9

SIFT feature extraction (part 7): Updating the gradient descriptor histogram. The quantity z pertaining to the continuous position (u', v', ϕ') is to be accumulated into the 3D histogram \mathbf{h}_∇ (u', v' are normalized spatial coordinates, ϕ' is the orientation). The quantity z is distributed over up to eight neighboring histogram bins (see Fig. 25.26) by tri-linear interpolation. Note that the orientation coordinate ϕ' receives special treatment because it is circular. Global parameters: n_{spat} , n_{angl} (see Table 25.5).

best 25 matches are shown in the examples. Feature matches are numbered according to their goodness, that is, label “1” denotes the best-matching descriptor pair (with the smallest feature distance). Selected details from these results are shown in Fig. 25.29. Unless otherwise noted, all SIFT parameters are set to their default values (see Table 25.5).

Although the use of the Euclidean (L_2) norm for measuring the distances between feature vectors in Eqn. (25.101) is suggested in [153], other norms have been considered [130, 181, 227] to improve

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Alg. 25.10

SIFT feature extraction (part 8): Converting the orientation histogram to a SIFT feature vector. Global parameters: n_{spat} , n_{angl} , t_{fclip} , s_{fscale} (see Table 25.5).

```

1: MakeSiftFeatureVector( $\mathbf{h}_\nabla$ )
   Input:  $\mathbf{h}_\nabla$ , gradient histogram of size  $n_{\text{spat}} \times n_{\text{spat}} \times n_{\text{angl}}$ .
   Returns a 1D integer (unsigned byte) vector obtained from  $\mathbf{h}_\nabla$ .
2: Create map  $f : [0, n_{\text{spat}}^2 \cdot n_{\text{angl}} - 1] \mapsto \mathbb{R}$      $\triangleright$  new 1D vector  $f$ 
3:  $m \leftarrow 0$ 
4: for  $i \leftarrow 0, \dots, n_{\text{spat}} - 1$  do                                 $\triangleright$  flatten  $\mathbf{h}_\nabla$  into  $f$ 
5:   for  $j \leftarrow 0, \dots, n_{\text{spat}} - 1$  do
6:     for  $k \leftarrow 0, \dots, n_{\text{angl}} - 1$  do
7:        $f(m) \leftarrow \mathbf{h}_\nabla(i, j, k)$ 
8:        $m \leftarrow m + 1$ 
9: Normalize( $f$ )
10: ClipPeaks( $f$ ,  $t_{\text{fclip}}$ )
11: Normalize( $f$ )
12:  $f_{\text{sift}} \leftarrow \text{MapToBytes}(f, s_{\text{fscale}})$ 
13: return  $f_{\text{sift}}$ 


---


14: Normalize( $\mathbf{x}$ )
   Scales vector  $\mathbf{x}$  to unit norm. Returns nothing, but  $\mathbf{x}$  is modified.
15:  $n \leftarrow \text{Size}(\mathbf{x})$ 
16:  $s \leftarrow \sum_{i=0}^{n-1} \mathbf{x}(i)$ 
17: for  $i \leftarrow 0, \dots, n-1$  do
18:    $\mathbf{x}(i) \leftarrow \frac{1}{s} \cdot \mathbf{x}(i)$ 
19: return


---


20: ClipPeaks( $\mathbf{x}, x_{\text{max}}$ )
   Limits the elements of  $\mathbf{x}$  to  $x_{\text{max}}$ . Returns nothing, but  $\mathbf{x}$  is modified.
21:  $n \leftarrow \text{Size}(\mathbf{x})$ 
22: for  $i \leftarrow 0, \dots, n-1$  do
23:    $\mathbf{x}(i) \leftarrow \min(\mathbf{x}(i), x_{\text{max}})$ 
24: return


---


25: MapToBytes( $\mathbf{x}, s$ )
   Converts the real-valued vector  $\mathbf{x}$  to an integer (unsigned byte) valued vector with elements in  $[0, 255]$ , using the scale factor  $s > 0$ .
26:  $n \leftarrow \text{Size}(\mathbf{x})$ 
27: Create a new map  $\mathbf{x}_{\text{int}} : [0, n-1] \mapsto [0, 255]$      $\triangleright$  new byte vector
28: for  $i \leftarrow 0, \dots, n-1$  do
29:    $a \leftarrow \text{round}(s \cdot \mathbf{x}(i))$                                  $\triangleright a \in \mathbb{N}_0$ 
30:    $\mathbf{x}_{\text{int}}(i) \leftarrow \min(a, 255)$                                  $\triangleright \mathbf{x}_{\text{int}}(i) \in [0, 255]$ 
31: return  $\mathbf{x}_{\text{int}}$ 

```

the statistical robustness and noise resistance. In Fig. 25.30, matching results are shown using the L_1 , L_2 , and L_∞ norms, respectively. Note that the resulting sets of top-ranking matches are almost the same with different distance norms, but the ordering of the strongest matches does change.

Figure 25.31 demonstrates the effectiveness of selecting feature matches based on the ratio between the distances to the best and the second-best match (see Eqns. (25.102)–(25.103)). Again the figure shows the 25 top-ranking matches based on the minimum (L_2) feature distance. With the maximum distance ratio ρ_{max} set to 1.0, rejection is practically turned off with the result that several false or ambiguous matches are among the top-ranking feature matches (Fig. 25.31(a)).

```

1: MatchDescriptors( $S^{(a)}, S^{(b)}, \rho_{\max}$ )
   Input:  $S^{(a)}, S^{(b)}$ , two sets of SIFT descriptors;  $\rho_{\max}$ , max. ratio
          of best and second-best matching distance (s. Eq. 25.105).
   Returns a sorted list of matches  $\mathbf{m}_{ij} = \langle \mathbf{s}_a, \mathbf{s}_b, d_{ij} \rangle$ , with  $\mathbf{s}_a \in$ 
           $S^{(a)}, \mathbf{s}_b \in S^{(b)}$  and  $d_{ij}$  being the distance between  $\mathbf{s}_a, \mathbf{s}_b$  in feature
          space.

2:  $M \leftarrow ()$                                  $\triangleright$  empty sequence of matches
3: for all  $\mathbf{s}_a \in S^{(a)}$  do
4:    $s_1 \leftarrow \text{nil}, d_{r,1} \leftarrow \infty$        $\triangleright$  best nearest neighbor
5:    $s_2 \leftarrow \text{nil}, d_{r,2} \leftarrow \infty$        $\triangleright$  second-best nearest neighbor
6:   for all  $\mathbf{s}_b \in S^{(b)}$  do
7:      $d \leftarrow \text{Dist}(\mathbf{s}_a, \mathbf{s}_b)$ 
8:     if  $d < d_{r,1}$  then                       $\triangleright d$  is a new ‘best’ distance
9:        $s_2 \leftarrow s_1, d_{r,2} \leftarrow d_{r,1}$ 
10:       $s_1 \leftarrow \mathbf{s}_b, d_{r,1} \leftarrow d$ 
11:    else
12:      if  $d < d_{r,2}$  then  $\triangleright d$  is a new ‘second-best’ distance
13:         $s_2 \leftarrow \mathbf{s}_b, d_{r,2} \leftarrow d$ 
14:      if  $(s_2 \neq \text{nil}) \wedge (\frac{d_{r,1}}{d_{r,2}} \leq \rho_{\max})$  then  $\triangleright$  Eqns. (25.104–25.105)
15:         $\mathbf{m} \leftarrow \langle \mathbf{s}_a, \mathbf{s}_1, d_{r,1} \rangle$             $\triangleright$  add a new match
16:         $M \cup \{\mathbf{m}\}$ 
17: Sort(M)                                      $\triangleright$  sort M to ascending distance  $d_{r,1}$ 
18: return M

19: Dist( $\mathbf{s}_a, \mathbf{s}_b$ )
   Input: descriptors  $\mathbf{s}_a = \langle x_a, y_a, \sigma_a, \theta_a, \mathbf{f}_a \rangle, \mathbf{s}_b = \langle x_b, y_b, \sigma_b, \theta_b,$ 
           $\mathbf{f}_b \rangle$ . Returns the Euclidean distance between feature vectors  $\mathbf{f}_a$ 
          and  $\mathbf{f}_b$ .
20:  $d \leftarrow \|\mathbf{f}_a - \mathbf{f}_b\|$ 
21: return d

```

With ρ_{\max} set to 0.8 and finally 0.5, the number of false matches is effectively reduced (Fig. 25.31(b,c)).²⁷

25.6 Efficient Feature Matching

The task of finding the best match based on the minimum distance in feature space is called “nearest-neighbor” search. If performed exhaustively, evaluating all possible matches between two descriptor sets $S^{(a)}$ and $S^{(b)}$ of size N_a and N_b , respectively, requires $N_a \cdot N_b$ feature distance calculations and comparisons. While this may be acceptable for small feature sets (with maybe up to 1000 descriptors each), this linear (brute-force) approach becomes prohibitively expensive for large feature sets with possibly millions of candidates, as required, for example, in the context of image database indexing or robot self-localization. Although efficient methods for exact nearest-neighbor search based on tree structures exist, such as the k -d tree method [80], it has been shown that these methods lose their effectiveness with increasing dimensionality of the search space.

25.6 EFFICIENT FEATURE MATCHING

Alg. 25.11

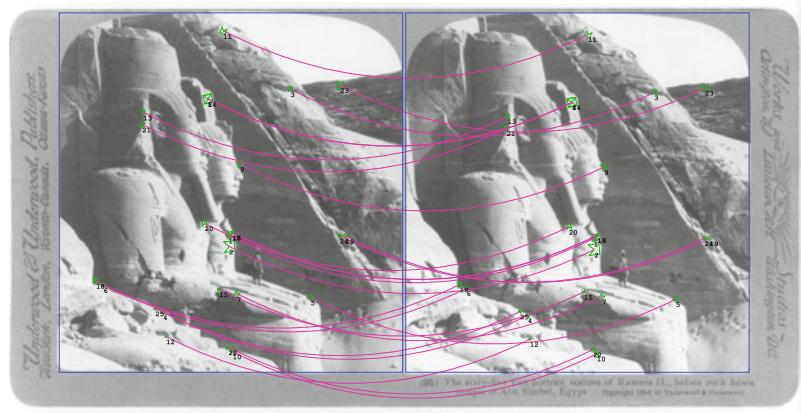
SIFT feature matching using Euclidean feature distance and linear search. The returned sequence of SIFT matches is sorted to ascending distance between corresponding feature pairs. Function $\text{Dist}(\mathbf{s}_a, \mathbf{s}_b)$ demonstrates the calculation of the Euclidean (L_2) feature distance, other options are the L_1 and L_∞ norms.

²⁷ $\rho_{\max} = 0.8$ is recommended in [153].

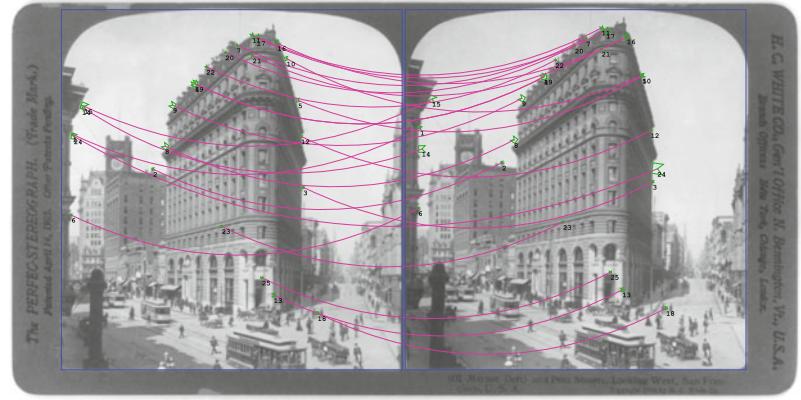
25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.28

SIFT feature matching examples on pairs of stereo images. Shown are the 25 best matches obtained with the L_2 feature distance and $\rho_{\max} = 0.8$.



(a)



(b)



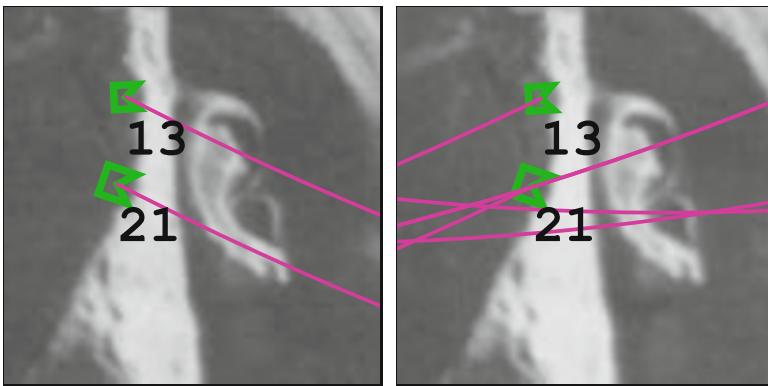
(c)

In fact, no algorithms are known that significantly outperform exhaustive (linear) nearest neighbor search in feature spaces that are more than about 10-dimensional [153]. SIFT feature vectors are 128-dimensional and therefore exact nearest-neighbor search is not a viable option for efficient matching between large descriptor sets.

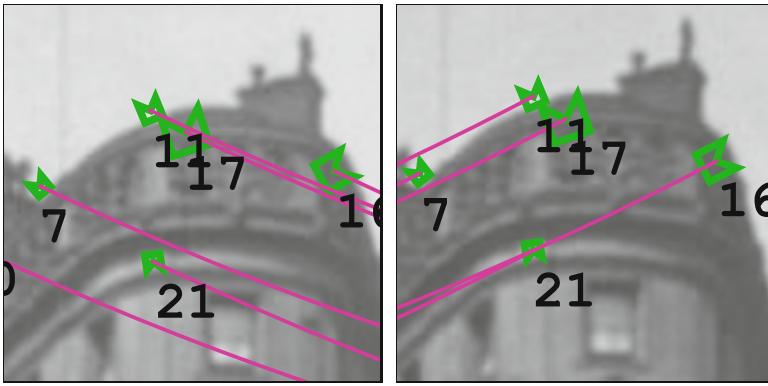
The approach taken in [21, 153] abandons exact nearest-neighbor search in favor of finding an *approximate* solution with substantially reduced effort, based on ideas described in [9]. This so-called

Left frame

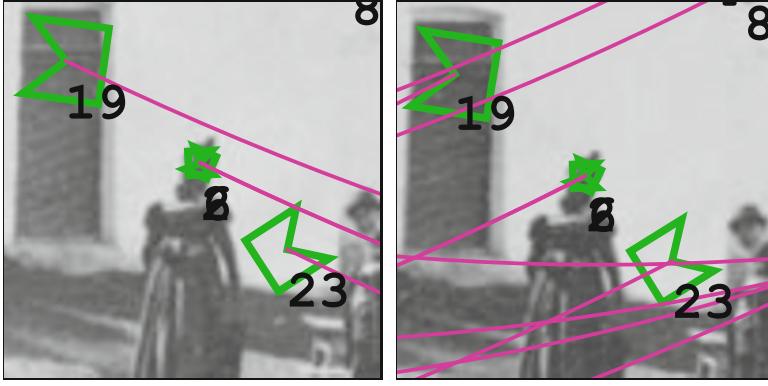
Right frame



(a)



(b)



(c)

25.6 EFFICIENT FEATURE MATCHING

Fig. 25.29

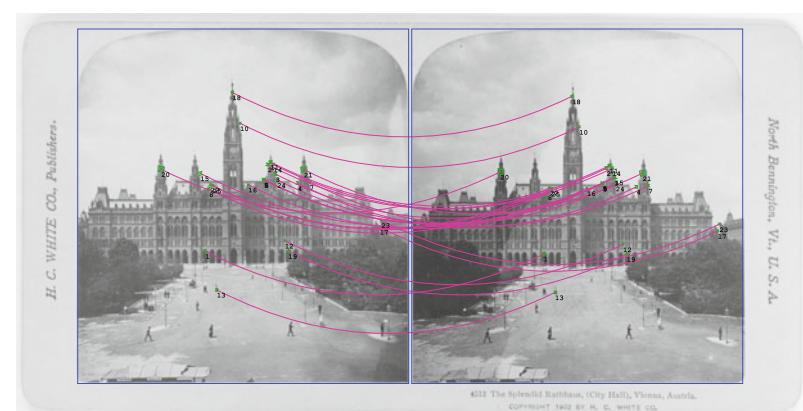
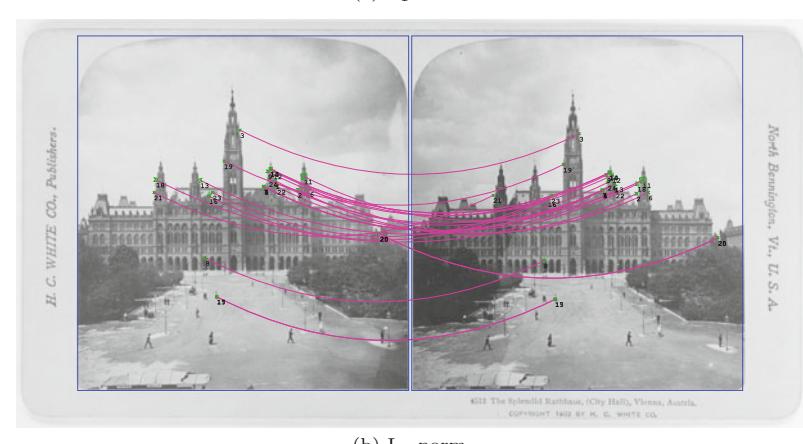
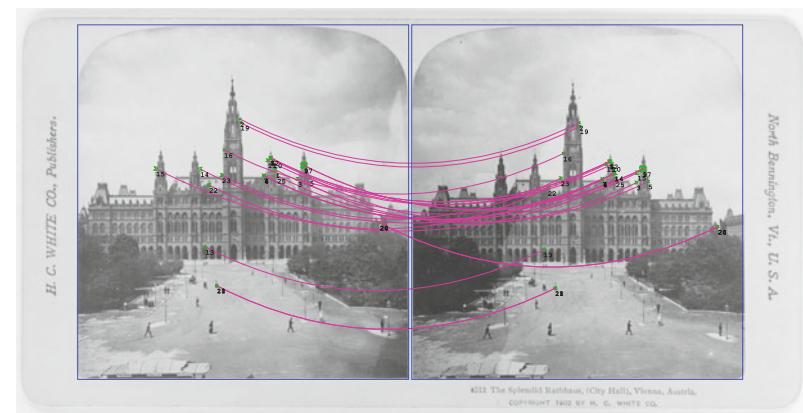
Stereo matching examples (enlarged details from Fig. 25.28).

“best-bin-first” method uses a modified k -d algorithm, which searches neighboring feature space partitions in the order of their closest distance from the given feature vector. To limit the exploration to a small fraction of the feature space, the search is cut off after checking the first 200 candidates, which results in a substantial speedup without compromising the search results, particularly when combined with feature selection based on the ratio of primary and secondary distances (see Eqns. (25.104)–(25.105)). Additional details can be found in [21].

25 SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

Fig. 25.30

Using different distance norms for feature matching. L_1 (a), L_2 (b), and L_∞ norm (c). All other parameters are set to their default values (see Table 25.5).



Approximate nearest-neighbor search in high-dimensional spaces is not only essential for practical SIFT matching in real time, but is a general problem with numerous applications in various disciplines and continued research. Open-source implementations of several different methods are available as software libraries.

25.7 JAVA IMPLEMENTATION

Fig. 25.31

Rejection of weak or ambiguous matches by limiting the ratio of primary and secondary match distance ρ_{\max} (see Eqns. (25.104)–(25.105)).



(a) $\rho_{\max} = 1.0$



(b) $\rho_{\max} = 0.8$



(c) $\rho_{\max} = 0.5$

25.7 Java Implementation

A new and complete Java implementation of the SIFT method has been written from ground up to complement the algorithms described in this chapter. Space limitations do not permit a full listing here, but the entire implementation and additional examples can be found in the source code section of this book's website. Most Java methods are named and structured identically to the procedures listed in the algorithms for easy identification. Note, however, that this imple-

mentation is again written for instructional clarity and readability. The code is neither tuned for efficiency nor is it intended to be used in a production environment.

25.7.1 SIFT Feature Extraction

The key class in this Java library is `SiftDetector`, which implements a SIFT detector for a given floating-point image. The following example illustrates its basic use for a given `ImageProcessor` object `ip`:

```
...
FloatProcessor I = ip.convertToFloatProcessor();
SiftDetector sd = new SiftDetector(I);
List<SiftDescriptor> S = sd.getSiftFeatures();
... // process descriptor set S
```

The initial work of setting up the required Gaussian and DoG scale space structures for the given image `I` is accomplished by the constructor in `new SiftDetector(I)`.

The method `getSiftFeatures()` then performs the actual feature detection process and returns a sequence of `SiftDescriptor` objects (`S`) for the image `I`. Each extracted `SiftDescriptor` in `S` holds information about its image position (`x, y`), its absolute scale σ (`scale`) and its dominant orientation θ (`orientation`). It also contains an invariant, 128-element, `int`-type feature vector f_{sift} (see Alg. 25.8).

The SIFT detector uses a large set of parameters that are set to their default values (see Table 25.5) if the simple constructor `new SiftDetector(I)` is used, as in the previous example. All parameters can be adjusted individually by passing a parameter object (of type `SiftDetector.Parameters`) to its constructor, as in the following example, which shows feature extraction from two images `A, B` using identical parameters:

```
...
FloatProcessor Ia = A.convertToFloatProcessor();
FloatProcessor Ib = B.convertToFloatProcessor();
...
SiftDetector.Parameters params =
    new SiftDetector.Parameters();
params.sigma_s = 0.5; // modify individual parameters
params.sigma_0 = 1.6;
...
SiftDetector sda = new SiftDetector(Ia, params);
SiftDetector sdb = new SiftDetector(Ib, params);
List<SiftDescriptor> SA = sda.getSiftFeatures();
List<SiftDescriptor> SB = sdb.getSiftFeatures();
...
// process descriptor sets SA and SB
```

Finding matching descriptors from a pair of SIFT descriptor sets **S_a**, **S_b** is accomplished by the class **SiftMatcher**.²⁸ One descriptor set (**S_a**) is considered the “reference” or “model” set and used to initialize a new **SiftMatcher** object, as shown in the following example. The actual matches are then calculated by invoking the method **matchDescriptors()**, which implements the procedure **MatchDescriptors()** outlined in Alg. 25.11. It takes the second descriptor set (**S_b**) as the only argument. The following code segment continues from the previous example:

```
...
SiftMatcher.Parameters params =
    new SiftMatcher.Parameters();
// set matcher parameters here (see below)
SiftMatcher matcher = new SiftMatcher(SA, params);
List<SiftMatch> matches = matcher.matchDescriptors(SB);
...
// process matches
```

As noted, certain parameters of class **SiftMatcher** can be set individually, for example,

```
params.norm = FeatureDistanceNorm.L1; // L1, L2, or Linf
params.ratioMax = 0.8; //  $\rho_{\max}$ , max. ratio of best and second-best match
params.sort = true; // set to true if sorting of matches is desired
```

The method **matchDescriptors()** in this prototypical implementation performs an exhaustive search over all possible descriptor pairs in the two sets **S_a** and **S_b**. To implement efficient approximate nearest-neighbor search (see Sec. 25.6), one would pre-calculate the required search tree structures for the model descriptor set (**S_a**) once inside **SiftMatcher**’s constructor method. The same matcher object could then be reused to match against multiple descriptor sets without the need to recalculate the search tree structure over and over again. This is particularly effective when the given model set is large.

25.8 Exercises

Exercise 25.1. As claimed in Eqn. (25.12), the 2D LoG function $L_\sigma(x, y)$ can be approximated by the DoG in the form $L_\sigma(x, y) \approx \lambda \cdot (G_{\kappa\sigma}(x, y) - G_\sigma(x, y))$. Create a combined plot, similar to the one in Fig. 25.5(b), showing the 1D cross sections of the LoG and DoG functions (with $\sigma = 1.0$ and $y = 0$). Compare both functions by varying the values of $\kappa = 2.00, 1.25, 1.10, 1.05$, and 1.01 . How does the approximation change as κ approaches 1, and what happens if κ becomes exactly 1?

Exercise 25.2. Test the performance of the SIFT feature detection and matching on pairs of related images under (a) changes of image brightness and contrast, (b) image rotation, (c) scale changes,

²⁸ File `imagingbook.sift.SiftMatcher.java`.

(d) adding (synthetic) noise. Choose (or shoot) your own test images, show the results in a suitable way and document the parameters used.

Exercise 25.3. Evaluate the SIFT mechanism for tracking features in video sequences. Search for a suitable video sequence with good features to track and process the images frame-by-frame.²⁹ Then match the SIFT features detected in pairs of successive frames by connecting the best-matching features, as long as the “match quality” is above a predefined threshold. Visualize the resulting feature trajectories. Could other properties of the SIFT descriptors (such as position, scale, and dominant orientation) be used to improve tracking stability?

²⁹ In ImageJ, choose an AVI video short enough to fit into main memory and open it as an image stack.

Fourier Shape Descriptors

Fourier descriptors are an interesting method for modeling 2D shapes that are described as closed contours. Unlike polylines or splines, which are explicit and local descriptions of the contour, Fourier descriptors are *global* shape representations, that is, each component stands for a particular characteristic of the entire shape. If one component is changed, the whole shape will change. The advantage is that it is possible to capture coarse shape properties with only a few numeric values, and the level of detail can be increased (or decreased) by adding (or removing) descriptor elements. In the following, we describe what is called “cartesian” (or “elliptical”) Fourier descriptors, how they can be used to model the shape of closed 2D contours and how they can be adapted to compare shapes in a translation-, scale-, and rotation-invariant fashion.

26.1 Closed Curves in the Complex Plane

Any continuous curve C in the 2D plane can be expressed as a function $f: \mathbb{R} \rightarrow \mathbb{R}^2$, with

$$f(t) = \begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} f_x(t) \\ f_y(t) \end{pmatrix}, \quad (26.1)$$

with the continuous parameter t being varied over the range $[0, t_{\max}]$. If the curve is closed, then $f(0) = f(t_{\max})$ and $f(t) = f(t + t_{\max})$. Note that $f_x(t)$, $f_y(t)$ are independent, real-valued functions, and t is the *path length* along the curve.

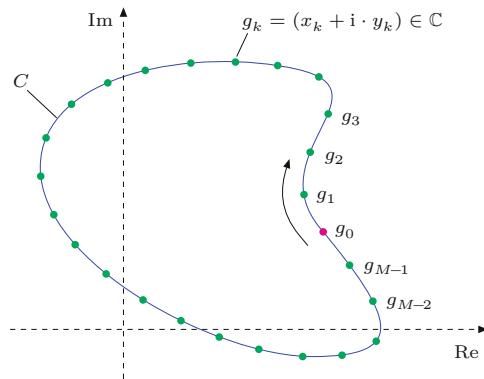
26.1.1 Discrete 2D Curves

Sampling a closed curve C at M regularly spaced positions t_0, t_1, \dots, t_{M-1} , with $t_i - t_{i-1} = \Delta_t = \text{Length}(C)/M$, results in a sequence (vector) of discrete 2D coordinates $V = (\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{M-1})$, with

$$\mathbf{v}_k = (x_k, y_k) = f(t_k). \quad (26.2)$$

Fig. 26.1

A closed, continuous 2D curve C , represented as a sequence of M uniformly placed samples $\mathbf{g} = (g_0, g_1, \dots, g_{M-1})$ in the complex plane.



Since the curve C is closed, the vector V represents a discrete function that is infinite and periodic, that is,

$$\mathbf{v}_k = \mathbf{v}_{k+pM}, \quad (26.3)$$

for $0 \leq k < M$ and any $p \in \mathbb{Z}$.

Contour points in the complex plane

Any 2D contour sample $\mathbf{v}_k = (x_k, y_k)$ can be interpreted as a point g_k in the complex plane,

$$g_k = x_k + i \cdot y_k, \quad (26.4)$$

with x_k and y_k taken as the real and imaginary components, respectively.¹ The result is a sequence (vector) of complex values

$$\mathbf{g} = (g_0, g_1, \dots, g_{M-1}), \quad (26.5)$$

representing the discrete 2D contour (see Fig. 26.1).

Regular position sampling

The assumption of input data being obtained by regular sampling is quite fundamental in traditional discrete Fourier analysis. In practice, contours of objects are typically not available as regularly sampled point sequences. For example, if an object has been segmented as a binary region, the coordinates of its boundary pixels could be used as the original contour sequence. However, the number of boundary pixels is usually too large to be used directly and their positions are not strictly uniformly spaced (at least under 8-connectivity). To produce a useful contour sequence from a region boundary, one could choose an arbitrary contour point as the start position \mathbf{x}_0 and then sample the x/y positions along the contour at regular (equidistant) steps, treating the centers of the boundary pixels as the vertices of a closed polygon. Algorithm 26.1 shows how to calculate a predefined number of contour points on an arbitrary polygon, such that the path

¹ Instead of $g \leftarrow x + i \cdot y$, we sometimes use the short notation $g \leftarrow (x, y)$ or $g \leftarrow \mathbf{v}$ for assigning the components of a 2D vector $\mathbf{v} = (x, y) \in \mathbb{R}^2$ to a complex variable $g \in \mathbb{C}$.

```

1: SamplePolygonUniformly( $V, M$ )
Input:  $V = (\mathbf{v}_0, \dots, \mathbf{v}_{N-1})$ , a sequence of  $N$  points representing
the vertices of a 2D polygon;  $M$ , number of desired sample points.
Returns a sequence  $\mathbf{g} = (g_0, \dots, g_{M-1})$  of complex values rep-
resenting points sampled uniformly along the path of the input
polygon  $V$ .
2:  $N \leftarrow |V|$ 
3:  $\Delta \leftarrow \frac{1}{M} \cdot \text{PathLength}(V)$   $\triangleright$  const. segment length  $\Delta$ 
4: Create map  $\mathbf{g}: [0, M-1] \rightarrow \mathbb{C}$   $\triangleright$  complex point sequence  $\mathbf{g}$ 
5:  $\mathbf{g}(0) \leftarrow \text{Complex}(V(0))$ 
6:  $i \leftarrow 0$   $\triangleright$  index of polygon segment  $\langle \mathbf{v}_i, \mathbf{v}_{i+1} \rangle$ 
7:  $k \leftarrow 1$   $\triangleright$  index of next point to be added to  $\mathbf{g}$ 
8:  $\alpha \leftarrow 0$   $\triangleright$  path position of polygon vertex  $\mathbf{v}_i$ 
9:  $\beta \leftarrow \Delta$   $\triangleright$  path position of next point to be added to  $\mathbf{g}$ 
10: while ( $i < N$ )  $\wedge$  ( $k < M$ ) do
11:    $\mathbf{v}_A \leftarrow V(i)$ 
12:    $\mathbf{v}_B \leftarrow V((i+1) \bmod N)$ 
13:    $\delta \leftarrow \|\mathbf{v}_B - \mathbf{v}_A\|$   $\triangleright$  length of segment  $\langle \mathbf{v}_A, \mathbf{v}_B \rangle$ 
14:   while ( $\beta \leq \alpha + \delta$ )  $\wedge$  ( $k < M$ ) do
15:      $\mathbf{x} \leftarrow \mathbf{v}_A + \frac{\beta - \alpha}{\delta} \cdot (\mathbf{v}_B - \mathbf{v}_A)$   $\triangleright$  linear path interpolation
16:      $\mathbf{g}(k) \leftarrow \text{Complex}(\mathbf{x})$ 
17:      $k \leftarrow k + 1$ 
18:      $\beta \leftarrow \beta + \Delta$ 
19:      $\alpha \leftarrow \alpha + \delta$ 
20:    $i \leftarrow i + 1$ 
21: return  $\mathbf{g}$ .

```

```

22: PathLength( $V$ )  $\triangleright$  returns the path length of the closed polygon  $V$ 
23:    $N \leftarrow |V|$ 
24:    $L \leftarrow 0$ 
25:   for  $i \leftarrow 0, \dots, N-1$  do
26:      $\mathbf{v}_A \leftarrow V(i)$ 
27:      $\mathbf{v}_B \leftarrow V((i+1) \bmod N)$ 
28:      $L \leftarrow L + \|\mathbf{v}_B - \mathbf{v}_A\|$ 
29:   return  $L$ .

```

length between the sample points is uniform. This algorithm is used in all examples involving contours obtained from binary regions.

Note that if the shape is given as an arbitrary polygon, the corresponding Fourier descriptor can also be calculated directly (and exactly) from the vertices of the polygon, without sub-sampling the polygon contour path at all. This “trigonometric” variant of the Fourier descriptor calculation is described in Sec. 26.3.7.

26.2 Discrete Fourier Transform (DFT)

Fourier descriptors are obtained by applying the 1D Discrete Fourier Transform (DFT)² to the complex-valued vector \mathbf{g} of 2D contour points (Eqn. (26.5)). The DFT is a transformation of a finite, complex-valued *signal* vector $\mathbf{g} = (g_0, g_1, \dots, g_{M-1})$ to a complex-valued *spec-*

26.2 DISCRETE FOURIER TRANSFORM (DFT)

Alg. 26.1

Regular sampling of a polygon path. Given a sequence V of 2D points representing the vertices of a closed polygon, $\text{SamplePolygonUniformly}(V, M)$ returns a sequence of M complex values \mathbf{g} on the polygon V , such that $\mathbf{g}(0) \equiv V(0)$ and all remaining points $\mathbf{g}(k)$ are uniformly positioned along the polygon path. See Alg. 26.9 for an alternate solution.

² See Chapter 18, Sec. 18.3.

trum $\mathbf{G} = (G_0, G_1, \dots, G_{M-1})$.³ Both the signal and the spectrum are of the same length (M) and periodic. In the following, we typically use k to denote the index in the time or space domain,⁴ and m for a frequency index in the spectral domain.

26.2.1 Forward Fourier Transform

The discrete Fourier spectrum $\mathbf{G} = (G_0, G_1, \dots, G_{M-1})$ is calculated from the discrete, complex-valued signal $\mathbf{g} = (g_0, g_1, \dots, g_{M-1})$ using the forward DFT, defined as⁵

$$G_m = \frac{1}{M} \cdot \sum_{k=0}^{M-1} g_k \cdot e^{-i \cdot 2\pi m \cdot \frac{k}{M}} = \frac{1}{M} \cdot \sum_{k=0}^{M-1} g_k \cdot e^{-i \cdot \omega_m \cdot \frac{k}{M}} \quad (26.6)$$

$$= \frac{1}{M} \cdot \sum_{k=0}^{M-1} \underbrace{[x_k + i \cdot y_k]}_{g_k} \cdot [\cos(\underbrace{2\pi m \frac{k}{M}}_{\omega_m}) - i \cdot \sin(\underbrace{2\pi m \frac{k}{M}}_{\omega_m})] \quad (26.7)$$

$$= \frac{1}{M} \cdot \sum_{k=0}^{M-1} [x_k + i \cdot y_k] \cdot [\cos(\omega_m \frac{k}{M}) - i \cdot \sin(\omega_m \frac{k}{M})], \quad (26.8)$$

for $0 \leq m < M$.⁶ Note that $\omega_m = 2\pi m$ denotes the *angular frequency* for the frequency index m . By applying the usual rules of complex multiplication, we obtain the *real* (Re) and *imaginary* (Im) parts of the spectral coefficients $G_m = (A_m + i \cdot B_m)$ explicitly as

$$A_m = \text{Re}(G_m) = \frac{1}{M} \sum_{k=0}^{M-1} [x_k \cdot \cos(\omega_m \frac{k}{M}) + y_k \cdot \sin(\omega_m \frac{k}{M})], \quad (26.9)$$

$$B_m = \text{Im}(G_m) = \frac{1}{M} \sum_{k=0}^{M-1} [y_k \cdot \cos(\omega_m \frac{k}{M}) - x_k \cdot \sin(\omega_m \frac{k}{M})]. \quad (26.10)$$

The DFT is defined for any signal length $M \geq 1$. If the signal length M is a power of two (that is, $M = 2^n$ for some $n \in \mathbb{N}$), the Fast Fourier Transform (FFT)⁷ can be used in place of the DFT for improved performance.

26.2.2 Inverse Fourier Transform (Reconstruction)

The inverse DFT reconstructs the original signal \mathbf{g} from a given spectrum \mathbf{G} . The formulation is almost symmetrical (except for the scale

³ In most traditional applications of the DFT (e.g. in acoustic processing), the signals are real-valued, that is, the imaginary components of the samples are zero. The Fourier spectrum is generally complex-valued, but it is symmetric for real-valued signals.

⁴ We use k instead of the usual i as the running index to avoid confusion with the imaginary constant “i” (despite the deliberate use of different glyphs).

⁵ This definition deviates slightly from the one used in Chapter 18, Sec. 18.3 but is otherwise equivalent.

⁶ Recall that $z = x + iy = |z| \cdot (\cos \psi + i \cdot \sin \psi) = |z| \cdot e^{i\psi}$, with $\psi = \tan^{-1}(y/x)$.

⁷ See Chapter 18, Sec. 18.4.2.

1: **FourierDescriptorUniform(\mathbf{g})**

Input: $\mathbf{g} = (g_0, \dots, g_{M-1})$, a sequence of M complex values, representing regularly sampled 2D points along a contour path.

Returns a Fourier descriptor \mathbf{G} of length M .

```

2:    $M \leftarrow |\mathbf{g}|$ 
3:   Create map  $\mathbf{G}: [0, M-1] \rightarrow \mathbb{C}$ 
4:   for  $m \leftarrow 0, \dots, M-1$  do
5:      $A \leftarrow 0, B \leftarrow 0$             $\triangleright$  real/imag. part of coefficient  $G_m$ 
6:     for  $k \leftarrow 0, \dots, M-1$  do
7:        $g \leftarrow \mathbf{g}(k)$ 
8:        $x \leftarrow \text{Re}(g), y \leftarrow \text{Im}(g)$ 
9:        $\phi \leftarrow 2 \cdot \pi \cdot m \cdot \frac{k}{M}$ 
10:       $A \leftarrow A + x \cdot \cos(\phi) + y \cdot \sin(\phi)$             $\triangleright$  Eq. 26.10
11:       $B \leftarrow B - x \cdot \sin(\phi) + y \cdot \cos(\phi)$ 
12:       $G(m) \leftarrow \frac{1}{M} \cdot (A + i \cdot B)$ 
13:   return  $\mathbf{G}$ .

```

26.2 DISCRETE FOURIER TRANSFORM (DFT)

Alg. 26.2

Calculating the Fourier descriptor for a sequence of uniformly sampled contour points. The complex-valued contour points in C represent 2D positions sampled uniformly along the contour path. Applying the DFT to \mathbf{g} yields the raw Fourier descriptor \mathbf{G} .

factor and the different signs in the exponent) to the forward transformation in Eqns. (26.6)–(26.8); its full expansion is

$$g_k = \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot \omega_m \cdot \frac{k}{M}} \quad (26.11)$$

$$= \sum_{m=0}^{M-1} \underbrace{[\text{Re}(G_m) + i \cdot \text{Im}(G_m)]}_{G_m} \cdot [\cos(\underbrace{2\pi m \frac{k}{M}}_{\omega_m}) + i \cdot \sin(\underbrace{2\pi m \frac{k}{M}}_{\omega_m})] \quad (26.12)$$

$$= \sum_{m=0}^{M-1} [A_m + i \cdot B_m] \cdot [\cos(\omega_m \frac{k}{M}) + i \cdot \sin(\omega_m \frac{k}{M})]. \quad (26.13)$$

Again we can expand Eqn. (26.13) to obtain the real and imaginary parts of the reconstructed signal, that is, the x/y -components of the corresponding curve points $g_k = (x_k, y_k)$ as

$$x_k = \text{Re}(g_k) = \sum_{m=0}^{M-1} [\text{Re}(G_m) \cdot \cos(2\pi m \frac{k}{M}) - \text{Im}(G_m) \cdot \sin(2\pi m \frac{k}{M})], \quad (26.14)$$

$$y_k = \text{Im}(g_k) = \sum_{m=0}^{M-1} [\text{Im}(G_m) \cdot \cos(2\pi m \frac{k}{M}) + \text{Re}(G_m) \cdot \sin(2\pi m \frac{k}{M})], \quad (26.15)$$

for $0 \leq k < M$. If *all* coefficients of the spectrum are used, this reconstruction is *exact*, that is, the resulting discrete points g_k are identical to the original contour points.⁸

With the aforementioned formulation we can not only reconstruct the discrete contour points g_k from the DFT spectrum, but also a smooth, interpolating curve as the sum of continuous sine and cosine components. To calculate *arbitrary* points on this curve, we replace the discrete quantity $\frac{k}{M}$ in Eqn. (26.15) by the continuous parameter t in the range $[0, 1]$. We must be careful about the frequencies, though. To achieve the desired *smooth* interpolation, the set of *lowest* possible

⁸ Apart from inaccuracies caused by finite floating-point precision.

frequencies ω_m must be used,⁹ that is,

$$x(t) = \sum_{m=0}^{M-1} [\operatorname{Re}(G_m) \cdot \cos(\omega_m \cdot t) - \operatorname{Im}(G_m) \cdot \sin(\omega_m \cdot t)], \quad (26.16)$$

$$y(t) = \sum_{m=0}^{M-1} [\operatorname{Im}(G_m) \cdot \cos(\omega_m \cdot t) + \operatorname{Re}(G_m) \cdot \sin(\omega_m \cdot t)], \quad (26.17)$$

$$\text{with } \omega_m = \begin{cases} 2\pi m & \text{for } m \leq (M \div 2), \\ 2\pi(m-M) & \text{for } m > (M \div 2), \end{cases} \quad (26.18)$$

where \div denotes the quotient (i.e., integer division). Alternatively, we could write Eqn. (26.17) in the form

$$x(t) = \sum_{m=-\frac{M-1}{2}}^{\frac{M-1}{2}} [\operatorname{Re}(G_{m \bmod M}) \cdot \cos(2\pi mt) - \operatorname{Im}(G_{m \bmod M}) \cdot \sin(2\pi mt)], \quad (26.19)$$

$$y(t) = \sum_{m=-\frac{M-1}{2}}^{\frac{M-1}{2}} [\operatorname{Im}(G_{m \bmod M}) \cdot \cos(2\pi mt) + \operatorname{Re}(G_{m \bmod M}) \cdot \sin(2\pi mt)]. \quad (26.20)$$

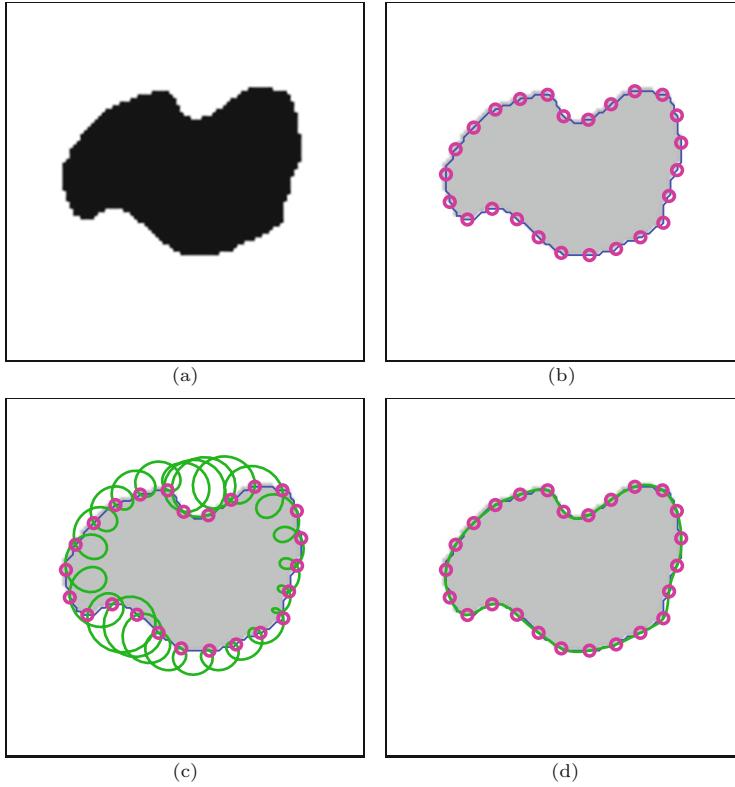
This formulation is used for the purpose of shape reconstruction from Fourier descriptors in Alg. 26.4.

[Figure \(26.2\)](#) shows the reconstruction of the discrete contour points as well as the calculation of a continuous outline from the DFT spectrum obtained from a sequence of discrete contour positions. The original sample points were taken at $M = 25$ uniformly spaced positions along the region's contour. The discrete points in [Fig. 26.2\(b\)](#) are exactly reconstructed from the complete DFT spectrum, as specified in Eqn. (26.15). The interpolated (green) outline in [Fig. 26.2\(c\)](#) was calculated with Eqn. (26.15) for continuous positions, based on the frequencies $m = 0, \dots, M-1$. The oscillations of the resulting curve are explained by the high-frequency components. Note that the curve still passes exactly through each of the original sample points, in fact, these can be perfectly reconstructed from *any* contiguous range of M coefficients and the corresponding harmonic frequencies. The smooth interpolation in [Fig. 26.2\(d\)](#), based on the symmetric low-frequency coefficients $m = -\frac{M-1}{2}, \dots, \frac{M-1}{2}$ (see Eqn. (26.20)) shows no such oscillations, since no high-frequency components are included.

26.2.3 Periodicity of the DFT Spectrum

When we apply the DFT, we implicitly assume that both the signal vector $\mathbf{g} = (g_0, g_1, \dots, g_{M-1})$ and the spectral vector $\mathbf{G} = (G_0, G_1, \dots, G_{M-1})$ represent discrete, periodic functions of infinite extent

⁹ Due to the periodicity of the discrete spectrum, any summation over M successive frequencies ω_m can be used to reconstruct the original discrete x/y samples. However, a smooth interpolation between the discrete x/y samples can only be obtained from the set of *lowest* frequencies in the range $[-\frac{M}{2}, +\frac{M}{2}]$ centered around the zero frequency, as in Eqns. (26.17) and (26.20).



26.2 DISCRETE FOURIER TRANSFORM (DFT)

Fig. 26.2

Contour reconstruction by inverse DFT. Original image (a), $M = 25$ uniformly spaced sample points on the region's contour (b). Continuous contour (green line) reconstructed by using frequencies ω_m with $m = 0, \dots, 24$ (c). Note that despite the oscillations introduced by the high frequencies, the continuous contour passes exactly through the original sample points. Smooth interpolation reconstructed with Eqn. (26.17) from the lowest-frequency coefficients in the symmetric range $m = -12, \dots, +12$ (d).

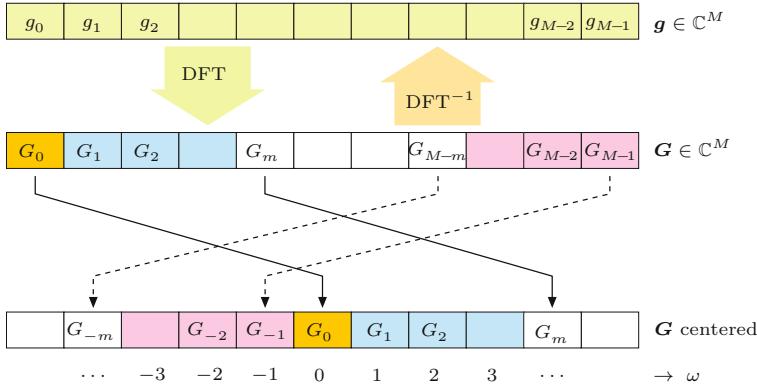


Fig. 26.3

Applying the DFT to a complex-valued vector \mathbf{g} of length M yields the complex-valued spectrum \mathbf{G} that is also of length M . The DFT spectrum is infinite and periodic with M , thus $G_{-m} = G_{M-m}$, as illustrated by the centered representation of the DFT spectrum (bottom). ω at the bottom denotes the harmonic number (multiple of the fundamental frequency) associated with each coefficient.

(see [39, Ch. 13] for details). Due to this periodicity, $\mathbf{G}(0) = \mathbf{G}(M)$, $\mathbf{G}(1) = \mathbf{G}(M+1)$, etc. In general,

$$\mathbf{G}(q \cdot M + m) = \mathbf{G}(m) \quad \text{and} \quad \mathbf{G}(m) = \mathbf{G}(m \bmod M), \quad (26.21)$$

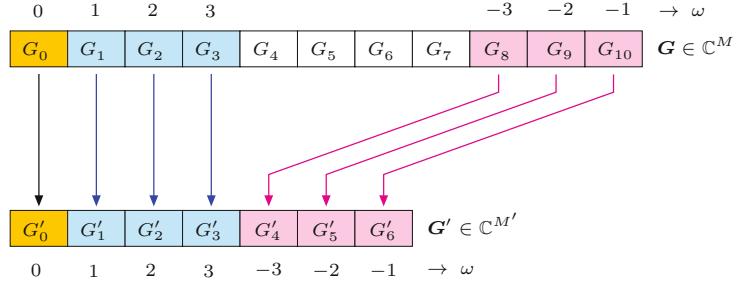
for arbitrary integers $q, m \in \mathbb{Z}$. Also, since $(-m \bmod M) = (M-m) \bmod M$, we can state that

$$\mathbf{G}(-m) = \mathbf{G}(M-m), \quad (26.22)$$

for any $m \in \mathbb{Z}$, such that $\mathbf{G}(-1) = \mathbf{G}(M-1)$, $\mathbf{G}(-2) = \mathbf{G}(M-2)$, etc., as illustrated in Fig. 26.3.

Fig. 26.4
Truncating a DFT spectrum from $M = 11$ to $M' = 7$ coefficients, as specified in Eqns. (26.23) and (26.24). Coefficients G_4, \dots, G_7 are discarded ($M' \div 2 = 3$).

Note that the associated harmonic number ω remains the same for each coefficient.



26.2.4 Truncating the DFT Spectrum

In the original formulation in Eqns. (26.6)–(26.8), the DFT is applied to a signal \mathbf{g} of length M and yields a discrete Fourier spectrum \mathbf{G} with M coefficients. Thus the signal and the spectrum have the same length. For shape representation, it is often useful to work with a truncated spectrum, that is, a reduced number of low-frequency Fourier coefficients.

By truncating a spectrum we mean the removal of coefficients above a certain harmonic number, which are (considering positive and negative frequencies) located around the center of the coefficient vector. Truncating a given spectrum \mathbf{G} of length $|\mathbf{G}| = M$ to a shorter spectrum \mathbf{G}' of length $M' \leq M$ is done as

$$\mathbf{G}'(m) \leftarrow \begin{cases} \mathbf{G}(m) & \text{for } 0 \leq m \leq M' \div 2, \\ \mathbf{G}(M - M' + m) & \text{for } M' \div 2 < m < M', \end{cases} \quad (26.23)$$

or simply

$$\mathbf{G}'(m \bmod M') \leftarrow \mathbf{G}(m \bmod M), \quad (26.24)$$

for $(M' \div 2 - M' + 1) \leq m \leq (M' \div 2)$. This works for M and M' being even or odd. The example in Fig. 26.4 illustrates how an original DFT spectrum \mathbf{G} of length $M = 11$ is truncated to \mathbf{G}' with only $M' = 7$ coefficients.

Of course it is also possible to calculate the truncated spectrum directly from the contour samples, without going through the full DFT spectrum. With M being the length of the signal vector \mathbf{g} and $M' \leq M$ the desired length of the (truncated) spectrum \mathbf{G}' , Eqn. (26.6) modifies to

$$\mathbf{G}'(m \bmod M') = \frac{1}{M} \cdot \sum_{k=0}^{M-1} g_k \cdot e^{-i2\pi m \frac{k}{M}}, \quad (26.25)$$

for m in the same range as in Eqn. (26.24). This approach is more efficient than truncating the complete spectrum, since unneeded coefficients are never calculated. Algorithm 26.3, which is a modified version of Alg. 26.2, summarizes the steps we have described.

Since some of the coefficients are missing, it is not possible to reconstruct the original signal vector \mathbf{g} from the truncated DFT spectrum \mathbf{G}' . However, the calculation of a partial reconstruction is possible, for example, using the formulation in Eqn. (26.20). In this

```

1: FourierDescriptorUniform( $\mathbf{g}, M'$ )
Input:  $\mathbf{g} = (g_0, \dots, g_{M-1})$ , a sequence of  $M$  complex values,
representing regularly sampled 2D points along a contour path.
 $M'$ , the number of Fourier coefficients ( $M' \leq M$ ).
Returns a truncated Fourier descriptor  $\mathbf{G}$  of length  $M'$ .
2:  $M \leftarrow |\mathbf{g}|$ 
3: Create map  $\mathbf{G}: [0, M'-1] \rightarrow \mathbb{C}$ 
4: for  $m \leftarrow (M' \div 2 - M' + 1), \dots, (M' \div 2)$  do
5:    $A \leftarrow 0, B \leftarrow 0$             $\triangleright$  real/imag. part of coefficient  $G_m$ 
6:   for  $k \leftarrow 0, \dots, M-1$  do
7:      $g \leftarrow \mathbf{g}(k)$ 
8:      $x \leftarrow \text{Re}(g), y \leftarrow \text{Im}(g)$ 
9:      $\phi \leftarrow 2 \cdot \pi \cdot m \cdot \frac{k}{M}$ 
10:     $A \leftarrow A + x \cdot \cos(\phi) + y \cdot \sin(\phi)$             $\triangleright$  Eq. 26.10
11:     $B \leftarrow B - x \cdot \sin(\phi) + y \cdot \cos(\phi)$ 
12:    $\mathbf{G}(m \bmod M') \leftarrow \frac{1}{M} \cdot (A + i \cdot B)$ 
13: return  $\mathbf{G}$ .

```

case, the discarded (high-frequency) coefficients are simply assumed to have zero values (see Sec. 26.3.6 for more details).

26.3 Geometric Interpretation of Fourier Coefficients

The contour reconstructed by the inverse transformation (Eqn. (26.15)) is the sum of M terms, one for each Fourier coefficient $G_m = (A_m, B_m)$. Each of these M terms represents a particular 2D shape in the spatial domain and the original contour can be obtained by point-wise addition of the individual shapes. So what are the spatial shapes that correspond to the individual Fourier coefficients?

26.3.1 Coefficient G_0 Corresponds to the Contour's Centroid

We first look only at the specific Fourier coefficient G_0 with frequency index $m = 0$. Substituting $m = 0$ and $\omega_0 = 0$ in Eqn. (26.10), we get

$$A_0 = \frac{1}{M} \sum_{k=0}^{M-1} [x_k \cdot \cos(0) + y_k \cdot \sin(0)] \quad (26.26)$$

$$= \frac{1}{M} \sum_{k=0}^{M-1} [x_k \cdot 1 + y_k \cdot 0] = \frac{1}{M} \sum_{k=0}^{M-1} x_k = \bar{x}, \quad (26.27)$$

$$B_0 = \frac{1}{M} \sum_{k=0}^{M-1} [y_k \cdot \cos(0) - x_k \cdot \sin(0)] \quad (26.28)$$

$$= \frac{1}{M} \sum_{k=0}^{M-1} [y_k \cdot 1 - x_k \cdot 0] = \frac{1}{M} \sum_{k=0}^{M-1} y_k = \bar{y}. \quad (26.29)$$

Thus $G_0 = (A_0, B_0) = (\bar{x}, \bar{y})$ is simply the average of the x/y -coordinates, that is, the *centroid* of the original contour points g_k (see

26.3 GEOMETRIC INTERPRETATION OF FOURIER COEFFICIENTS

Alg. 26.3

Calculating a truncated Fourier descriptor for a sequence of uniformly sampled contour points (adapted from Alg. 26.2). The M complex-valued contour points in \mathbf{g} represent 2D positions sampled uniformly along the contour path. The resulting Fourier descriptor \mathbf{G} contains only M' coefficients for the M' lowest harmonic frequencies.

	G_{-j}	G_{-2}	G_{-1}	G_0	G_1	G_2		G_j	\mathbf{G}
--	----------	----------	----------	-------	-------	-------	--	-------	--------------

Fig. 26.5
DFT coefficient G_0 corresponds to the centroid of the contour points.

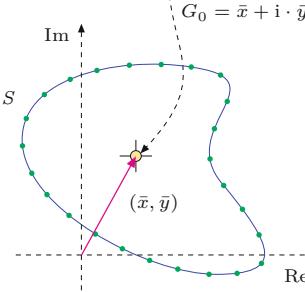


Fig. 26.5.¹⁰ If we apply the *inverse Fourier transform* (Eqn. (26.15)) by ignoring (i.e., zeroing) all coefficients except G_0 , we get the *partial reconstruction*¹¹ of the 2D contour coordinates $g_k^{(0)} = (x_k^{(0)}, y_k^{(0)})$ as

$$x_k^{(0)} = [A_0 \cdot \cos(\omega_0 \frac{k}{M}) - B_0 \cdot \sin(\omega_0 \frac{k}{M})] \quad (26.30)$$

$$= \bar{x} \cdot \cos(0) - \bar{y} \cdot \sin(0) = \bar{x} \cdot 1 - \bar{y} \cdot 0 = \bar{x}, \quad (26.31)$$

$$y_k^{(0)} = [B_0 \cdot \cos(\omega_0 \frac{k}{M}) + A_0 \cdot \sin(\omega_0 \frac{k}{M})] \quad (26.32)$$

$$= \bar{y} \cdot \cos(0) + \bar{x} \cdot \sin(0) = \bar{y} \cdot 1 + \bar{x} \cdot 0 = \bar{y}. \quad (26.33)$$

Thus the contribution of the spectral value G_0 is the *centroid* of the reconstructed shape (see Fig. 26.5). If we perform a partial reconstruction of the contour using only the spectral coefficient G_0 , then all contour points

$$g_0^{(0)} = g_1^{(0)} = \dots = g_k^{(0)} = \dots = g_{M-1}^{(0)} = (\bar{x}, \bar{y}) \quad (26.34)$$

would have the same (centroid) coordinate. This is because G_0 is the coefficient for the zero frequency and thus the sine and cosine terms in Eqns. (26.27) and (26.29) are constant. Alternatively, if we reconstruct the signal by *omitting* G_0 (i.e., $\mathbf{g}^{(1,\dots,M-1)}$), the resulting contour is identical to the original shape, except that it is centered at the coordinate origin.

26.3.2 Coefficient G_1 Corresponds to a Circle

Next, we look at the geometric interpretation of $G_1 = (A_1, B_1)$, that is, the coefficient with frequency index $m = 1$, which corresponds to the angular frequency $\omega_1 = 2\pi$. Assuming that all coefficients G_m in the DFT spectrum are set to zero, except the single coefficient G_1 ,

¹⁰ Note that the centroid of a boundary is generally not the same as the centroid of the enclosed region.

¹¹ We use the notation $\mathbf{g}^{(m)} = (g_0^{(m)}, g_1^{(m)}, \dots, g_{M-1}^{(m)})$ for the *partial reconstruction* of the contour \mathbf{g} from only a single Fourier coefficient G_m . For example, $\mathbf{g}^{(0)}$ is the reconstruction from the zero-frequency coefficient G_0 only. Analogously, we use $\mathbf{g}^{(a,b,c)}$ to denote a partial reconstruction based on selected Fourier coefficients G_a, G_b, G_c .

we get the partially reconstructed contour points $\mathbf{g}^{(1)}$ by Eqn. (26.11) as

$$g_k^{(1)} = G_1 \cdot e^{i \cdot 2\pi \cdot \frac{k}{M}} \quad (26.35)$$

$$= [A_1 + i \cdot B_1] \cdot [\cos(2\pi \frac{k}{M}) + i \cdot \sin(2\pi \frac{k}{M})], \quad (26.36)$$

for $0 \leq k < M$. Remember that the complex values of $e^{i\varphi}$ describe a *unit circle* in the complex plane that performs one full (counter-clockwise) revolution, as the angle φ runs from $0, \dots, 2\pi$. Analogously, $e^{i2\pi t}$ also describes a complete unit circle as t goes from 0 to 1. Since the term $\frac{k}{M}$ (for $0 \leq k < M$) also varies from 0 to 1 in Eqn. (26.36), the M reconstructed contour points are placed on a circle at equal angular steps. Multiplying $e^{i \cdot 2\pi t}$ by a complex factor z stretches the *radius* of the circle by $|z|$, and also changes the *phase* (starting angle) of the circle by an angle θ , that is,

$$z \cdot e^{i \cdot \varphi} = |z| \cdot e^{i \cdot (\varphi + \theta)}, \quad (26.37)$$

with $\theta = \angle z = \arg(z) = \tan^{-1}(\text{Im}(z)/\text{Re}(z))$.

We now see that the points $g_k^{(1)} = G_1 \cdot e^{i \cdot 2\pi k / M}$, generated by Eqn. (26.36), are positioned uniformly on a circle with radius $r_1 = |G_1|$ and starting angle (phase)

$$\theta_1 = \angle G_1 = \tan^{-1}\left(\frac{\text{Im}(G_1)}{\text{Re}(G_1)}\right) = \tan^{-1}\left(\frac{B_1}{A_1}\right). \quad (26.38)$$

This point sequence is traversed in counter-clockwise direction for $k = 0, \dots, M-1$ at frequency $m = 1$, that is, the circle performs one full revolution while the contour is traversed once. The circle is centered at the coordinate origin $(0, 0)$, its radius is $|G_1|$, and its starting point (Eqn. (26.36) for $k = 0$) is

$$g_0^{(1)} = G_1 \cdot e^{i \cdot 2\pi m \cdot \frac{0}{M}} = G_1 \cdot e^{i \cdot 2\pi 1 \cdot \frac{0}{M}} = G_1 \cdot e^0 = G_1, \quad (26.39)$$

as illustrated in Fig. 26.6.

26.3.3 Coefficient G_m Corresponds to a Circle with Frequency m

Based on the aforementioned result for the frequency index $m = 1$, we can easily generalize the geometric interpretation of Fourier coefficients with arbitrary index $m > 0$. Using Eqn. (26.11), the partial reconstruction for the single Fourier coefficient $G_m = (A_m, B_m)$ is the contour $\mathbf{g}^{(m)}$, with coordinates

$$g_k^{(m)} = G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} \quad (26.40)$$

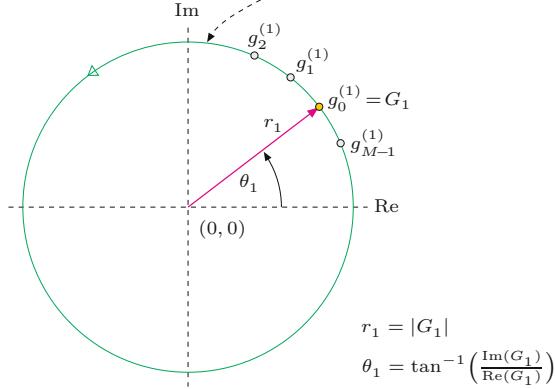
$$= [A_m + i \cdot B_m] \cdot [\cos(2\pi m \frac{k}{M}) + i \cdot \sin(2\pi m \frac{k}{M})], \quad (26.41)$$

which again describe a circle with radius $r_m = |G_m|$, phase $\theta_m = \arg(G_m) = \tan^{-1}(B_m/A_m)$, and starting point $g_0^{(m)} = G_m$. In this case, however, the angular velocity is scaled by m , that is, the resulting circle revolves m times faster than the circle for G_1 . In other

	G_{-j}		G_{-2}	G_{-1}	G_0	G_1	G_2		G_j		\mathbf{G}
--	----------	--	----------	----------	-------	-------	-------	--	-------	--	--------------

Fig. 26.6

A single DFT coefficient corresponds to a circle. The partial reconstruction from the single DFT coefficient G_m yields a sequence of M points $g_0^{(m)}, \dots, g_{M-1}^{(m)}$ on a circle centered at the coordinate origin, with radius r_m and starting angle (phase) θ_m .



words, while the contour is traversed once, this circle performs m full revolutions.

Note that G_0 (see Sec. 26.3.1) does not really constitute a special case at all. Formally, it also describes a circle but one that oscillates with zero frequency, that is, all points have the same (constant) position

$$g_k^{(0)} = G_0 \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = G_0 \cdot e^{i \cdot 2\pi 0 \cdot \frac{k}{M}} = G_0 \cdot e^0 = G_0, \quad (26.42)$$

for $k = 0, \dots, M-1$, which is equivalent to the curve's centroid $G_0 = (\bar{x}, \bar{y})$, as shown in Eqns. (26.27)–(26.29). Since the corresponding frequency is zero, the point never moves away from G_0 .

26.3.4 Negative Frequencies

The DFT spectrum is periodic and defined for all frequencies $m \in \mathbb{Z}$, including negative frequencies. From Eqn. (26.21) we know that for any DFT coefficient with negative index G_{-m} there is an equivalent coefficient G_n whose index n is in the range $0, \dots, M-1$. The partial reconstruction of the spectrum with the single coefficient G_{-m} is

$$g_k^{(-m)} = G_{-m} \cdot e^{-i \cdot 2\pi m \cdot \frac{k}{M}} = G_n \cdot e^{-i \cdot 2\pi m \cdot \frac{k}{M}}, \quad (26.43)$$

with $n = -m \bmod M$, which is again a sequence of points on the circle with radius $r_{-m} = r_n = |G_n|$ and phase $\theta_{-m} = \theta_n = \arg(G_n)$. The absolute rotation frequency is m , but this circle spins in the opposite, that is, *clockwise* direction, since angles become increasingly negative with growing k .

26.3.5 Fourier Descriptor Pairs Correspond to Ellipses

It follows therefore that the space-domain circles for the Fourier coefficients G_m and G_{-m} rotate with the same absolute frequency m but with different phase angles θ_m, θ_{-m} and in opposite directions. We denote the tuple

$$\text{FP}_m = (G_{-m}, G_{+m})$$

the “Fourier descriptor pair” (or “FD pair”) for the frequency index m . If we perform a partial reconstruction from only the two Fourier coefficients G_{-m}, G_{+m} of this FD pair, we obtain the spatial points

$$\begin{aligned} g_k^{(\pm m)} &= g_k^{(-m)} + g_k^{(+m)} \\ &= G_{-m} \cdot e^{-i \cdot 2\pi m \cdot \frac{k}{M}} + G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} \\ &= G_{-m} \cdot e^{-i \cdot \omega_m \cdot \frac{k}{M}} + G_m \cdot e^{i \cdot \omega_m \cdot \frac{k}{M}}. \end{aligned} \quad (26.44)$$

By Eqn. (26.15) we can expand the result from Eqn. (26.44) to cartesian x/y coordinates as¹²

$$\begin{aligned} x_k^{(\pm m)} &= A_{-m} \cdot \cos(-\omega_m \cdot \frac{k}{M}) - B_{-m} \cdot \sin(-\omega_m \cdot \frac{k}{M}) + \\ &\quad A_m \cdot \cos(\omega_m \cdot \frac{k}{M}) - B_m \cdot \sin(\omega_m \cdot \frac{k}{M}) \\ &= (A_{-m} + A_m) \cdot \cos(\omega_m \cdot \frac{k}{M}) + (B_{-m} - B_m) \cdot \sin(\omega_m \cdot \frac{k}{M}), \end{aligned} \quad (26.45)$$

$$\begin{aligned} y_k^{(\pm m)} &= B_{-m} \cdot \cos(-\omega_m \cdot \frac{k}{M}) + A_{-m} \cdot \sin(-\omega_m \cdot \frac{k}{M}) + \\ &\quad B_m \cdot \cos(\omega_m \cdot \frac{k}{M}) + A_m \cdot \sin(\omega_m \cdot \frac{k}{M}) \\ &= (B_{-m} + B_m) \cdot \cos(\omega_m \cdot \frac{k}{M}) - (A_{-m} - A_m) \cdot \sin(\omega_m \cdot \frac{k}{M}), \end{aligned} \quad (26.46)$$

for $k = 0, \dots, M-1$. The 2D point sequence $\mathbf{g}^{(\pm m)} = (g_0^{(\pm m)}, \dots, g_{M-1}^{(\pm m)})$, obtained with Eqns. (26.45) and (26.46), describes an oriented *ellipse* that is centered at the origin (see Fig. 26.7). The parametric equation for this ellipse is

$$\begin{aligned} x_t^{(\pm m)} &= (A_{-m} + A_m) \cdot \cos(\omega_m \cdot t) + (B_{-m} - B_m) \cdot \sin(\omega_m \cdot t), \\ &= (A_{-m} + A_m) \cdot \cos(2\pi m t) + (B_{-m} - B_m) \cdot \sin(2\pi m t), \end{aligned} \quad (26.47)$$

$$\begin{aligned} y_t^{(\pm m)} &= (B_{-m} + B_m) \cdot \cos(\omega_m \cdot t) - (A_{-m} - A_m) \cdot \sin(\omega_m \cdot t) \\ &= (B_{-m} + B_m) \cdot \cos(2\pi m t) - (A_{-m} - A_m) \cdot \sin(2\pi m t), \end{aligned} \quad (26.48)$$

for $t = 0, \dots, 1$.

Ellipse parameters

In general, the parametric equation of an ellipse with radii a, b , centered at (x_c, y_c) and oriented at an angle α is

$$\begin{aligned} x(\psi) &= x_c + a \cdot \cos(\psi) \cdot \cos(\alpha) - b \cdot \sin(\psi) \cdot \sin(\alpha), \\ y(\psi) &= y_c + a \cdot \cos(\psi) \cdot \sin(\alpha) + b \cdot \sin(\psi) \cdot \cos(\alpha), \end{aligned} \quad (26.49)$$

with $\psi = 0, \dots, 2\pi$. From Eqns. (26.45) and (26.46) we see that the parameters a_m, b_m, α_m of the ellipse for a single Fourier descriptor pair $\text{FP}_m = (G_{-m}, G_{+m})$ are

$$a_m = r_{-m} + r_{+m} = |G_{-m}| + |G_{+m}|, \quad (26.50)$$

$$b_m = |r_{-m} - r_{+m}| = ||G_{-m}| - |G_{+m}|||, \quad (26.51)$$

$$\begin{aligned} \alpha_m &= \frac{1}{2} \cdot \left(\underbrace{\Im G_{-m}}_{\theta_{-m}} + \underbrace{\Im G_{+m}}_{\theta_{+m}} \right) \\ &= \frac{1}{2} \cdot \left[\tan^{-1} \left(\frac{B_{-m}}{A_{-m}} \right) + \tan^{-1} \left(\frac{B_{+m}}{A_{+m}} \right) \right]. \end{aligned} \quad (26.52)$$

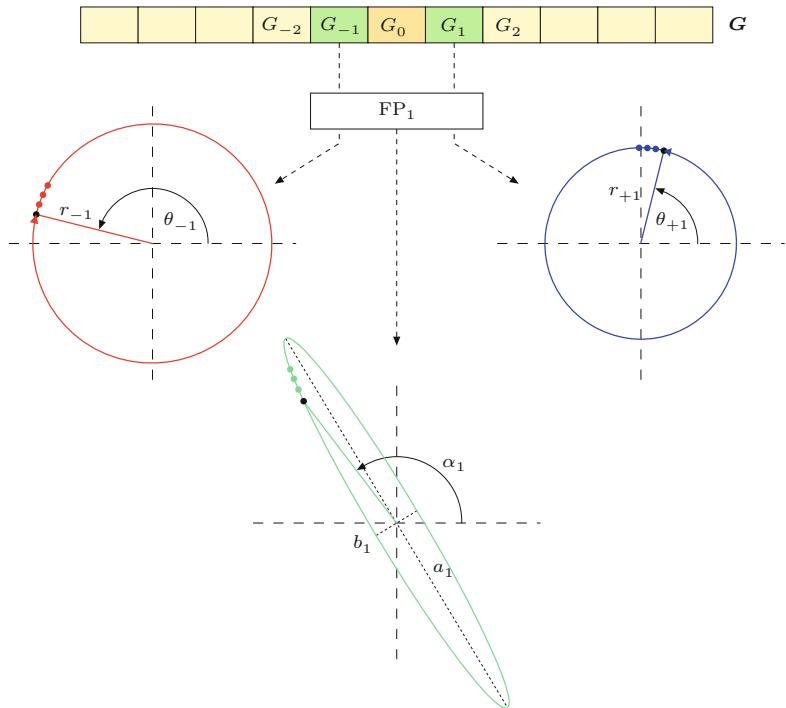
¹² Using the relations $\sin(-a) = -\sin(a)$ and $\cos(-a) = \cos(a)$.

26 FOURIER SHAPE DESCRIPTORS

Fig. 26.7

DFT coefficients G_{-m}, G_{+m} form a Fourier descriptor pair FP_m . Each of the two descriptors corresponds to M points on a circle of radius r_{-m}, r_{+m} and phase θ_{-m}, θ_{+m} , respectively, revolving with the same frequency m but in opposite directions.

The sum of each point pair is located on an ellipse with radii a_m, b_m and orientation α_m . The orientation α_m of the ellipse's major axis is centered between the starting angles of the circles defined by G_{-m} and G_{+m} ; its radii are $a_m = r_{-m} + r_{+m}$ for the major axis and $b_m = |r_{-m} - r_{+m}|$ for the minor axis. The figure shows the situation for $m = 1$.



Like its constituting circles, this ellipse is centered at $(x_c, y_c) = (0, 0)$ and performs m revolutions for one traversal of the contour. G_{-m} specifies the circle

$$z_{-m}(\varphi) = G_{-m} \cdot e^{i \cdot (-\varphi)} = r_{-m} \cdot e^{i \cdot (\theta_{-m} - \varphi)}, \quad (26.53)$$

for $\varphi \in [0, 2\pi]$, with starting angle θ_{-m} and radius r_{-m} , rotating in a clockwise direction. Similarly, G_{+m} specifies the circle

$$z_{+m}(\varphi) = G_{+m} \cdot e^{i \cdot (\varphi)} = r_{+m} \cdot e^{i \cdot (\theta_{+m} + \varphi)}, \quad (26.54)$$

with starting angle θ_{+m} and radius r_{+m} , rotating in a counter-clockwise direction. Both circles thus rotate at the same angular velocity but in opposite directions, as mentioned before. The corresponding (complex-valued) ellipse points are

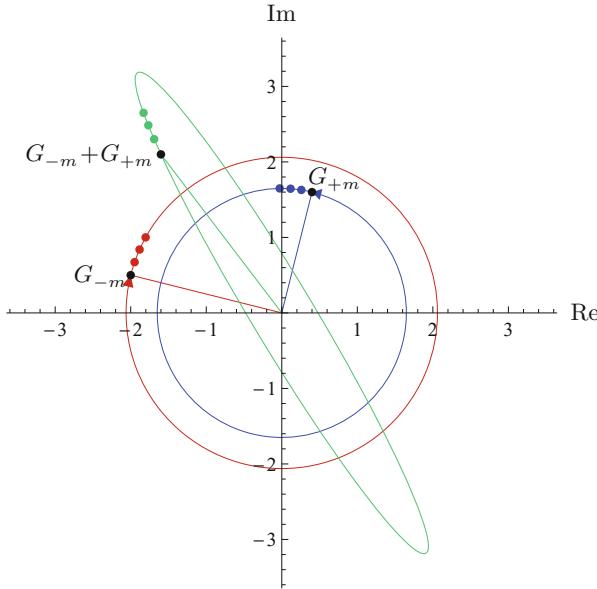
$$z_m(\varphi) = z_{-m}(\varphi) + z_{+m}(\varphi). \quad (26.55)$$

The ellipse radius $|z_m(\varphi)|$ is a *maximum* at position $\varphi = \varphi_{\max}$, where the angles on both circles are identical (i.e., the corresponding vectors have the same direction). This occurs when

$$\theta_{-m} - \varphi_{\max} = \theta_{+m} + \varphi_{\max} \quad \text{or} \quad \varphi_{\max} = \frac{1}{2} \cdot (\theta_{-m} - \theta_{+m}),$$

that is, at mid-angle between the two starting angles θ_{-m} and θ_{+m} . Therefore, the orientation of the ellipse's major axis is

$$\alpha_m = \theta_{+m} + \frac{\theta_{-m} - \theta_{+m}}{2} = \frac{1}{2} \cdot (\theta_{-m} + \theta_{+m}), \quad (26.56)$$


Fig. 26.8

Ellipse created by partial reconstruction from a single Fourier descriptor pair $\text{FP}_m = (G_{-m}, G_{+m})$. The two complex-valued Fourier coefficients $G_{-m} = (-2, 0.5)$ and $G_{+m} = (0.4, 1.6)$ represent circles with starting points G_{-m} and G_{+m} , respectively. The circle for G_{-m} (red) rotates in clockwise direction, the circle for G_{+m} (blue) rotates in counter-clockwise direction. The ellipse (green) is the result of point-wise addition of the two circles, as shown for four successive points, starting with point $G_{-m} + G_{+m}$.

as already stated in Eqn. (26.52). At $\varphi = \varphi_{\max}$ the two radial vectors align, and thus the radius of the ellipse's major axis a_m is the sum of the two circle radii, that is,

$$a_m = r_{-m} + r_{+m} \quad (26.57)$$

(cf. Eqn. (26.50)). Analogously, the ellipse radius is *minimized* at position $\varphi = \varphi_{\min}$, where the $z_{-m}(\varphi_{\min})$ and $z_{+m}(\varphi_{\min})$ lie on opposite sides of the circle. This occurs at angle

$$\varphi_{\min} = \varphi_{\max} + \frac{\pi}{2} = \frac{\pi + \theta_{-m} - \theta_{+m}}{2} \quad (26.58)$$

and the corresponding radius for the ellipse's minor axis is (cf. Eqn. (26.51))

$$b_m = r_{+m} - r_{-m}. \quad (26.59)$$

Figure 26.8 illustrates this situation for a specific Fourier descriptor pair $\text{FP}_m = (G_{-m}, G_{+m}) = (-2 + i \cdot 0.5, 0.4 + i \cdot 1.6)$. Note that the ellipse parameters a_m, b_m, α_m (see Eqns. (26.50)–(26.52)) are not explicitly required for reconstructing (drawing) the contour, since the ellipse can also be generated by simply adding the x/y -coordinates of the two counter-revolving circles for the participating Fourier descriptors, as given in Eqn. (26.55). Another example is shown in Fig. 26.9.

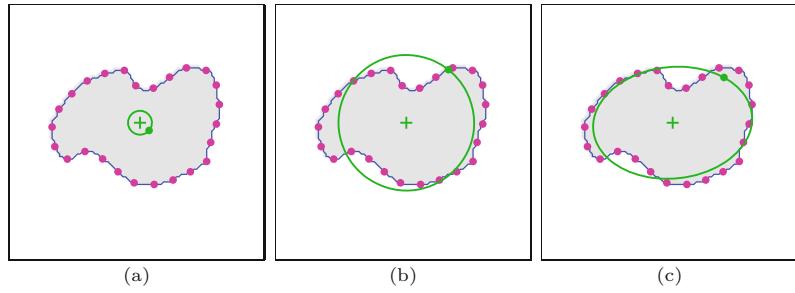
26.3.6 Shape Reconstruction from Truncated Fourier Descriptors

Due to the periodicity of the DFT spectrum, the complete reconstruction of the contour points g_k from the Fourier coefficients G_m (see Eqn. (26.11)) could also be written with a different summation range, as long as all spectral coefficients are included, that is,

26 FOURIER SHAPE DESCRIPTORS

Fig. 26.9

Partial reconstruction from single coefficients and an FD descriptor pair. The two circles reconstructed from DFT coefficient G_{-1} (a) and coefficient G_{+1} (b) are positioned at the centroid of the contour (G_0). The combined reconstruction for (G_{-1}, G_{+1}) produces the ellipse in (c). The dots on the green curves show the path position for $t = 0$.



$$g_k = \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = \sum_{m=m_0}^{m_0+M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}}, \quad (26.60)$$

for any start index $m_0 \in \mathbb{Z}$. As a special (though important) case we can perform the summation symmetrically around the zero index and write

$$g_k = \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = \sum_{\substack{m=-(M-1)/2 \\ m=1}}^{M/2} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}}. \quad (26.61)$$

To understand the reconstruction in terms of Fourier descriptor pairs, it is helpful to distinguish if M (the number of contour points and Fourier coefficients) is *even* or *odd*.

Odd number of contour points

If M is *odd*, then the spectrum consists of G_0 (representing the contour's centroid) plus exactly $M \div 2$ Fourier descriptor pairs FP_m , with $m = 1, \dots, M \div 2$.¹³ We can thus rewrite Eqn. (26.60) as

$$\begin{aligned} g_k &= \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = \underbrace{G_0}_{g_k^{(0)}} + \sum_{m=1}^{M/2} \underbrace{[G_{-m} \cdot e^{-i \cdot 2\pi m \cdot \frac{k}{M}} + G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}}]}_{g_k^{(\pm m)} = g_k^{(-m)} + g_k^{(m)}} \\ &= g_k^{(0)} + \sum_{m=1}^{M/2} g_k^{(\pm m)} = g_k^{(0)} + g_k^{(\pm 1)} + g_k^{(\pm 2)} + \dots + g_k^{(\pm M/2)}, \end{aligned} \quad (26.62)$$

where $g_k^{(\pm m)}$ denotes the partial reconstruction from the single Fourier descriptor pair FP_m (see Eqn. (26.44)).

As we already know, the partial reconstruction $g_k^{(\pm m)}$ of an individual Fourier descriptor pair FP_m is a set of points on an ellipse that is centered at the origin $(0, 0)$. The partial reconstruction of the *three* DFT coefficients G_0, G_{-m}, G_{+m} (i.e., FP_m plus the single coefficient G_0) is the point sequence

$$g_k^{(-m, 0, m)} = g_k^{(0)} + g_k^{(\pm m)}, \quad (26.63)$$

which is the ellipse for $g_k^{(\pm m)}$ shifted to $g_k^{(0)} = (\bar{x}, \bar{y})$, the centroid of the original contour. For example, the partial reconstruction from the coefficients G_{-1}, G_0, G_{+1} ,

¹³ If M is odd, then $M = 2 \cdot (M \div 2) + 1$.

$$g_k^{(-1,0,1)} = g_k^{(-1,\dots,1)} = g_k^{(0)} + g_k^{(\pm 1)}, \quad (26.64)$$

yields an ellipse with frequency $m = 1$ that revolves around the (fixed) centroid of the original contour. If we add another Fourier descriptor pair FP_2 , the resulting reconstruction is

$$g_k^{(-2,\dots,2)} = \underbrace{g_k^{(0)} + g_k^{(\pm 1)}}_{\text{ellipse 1}} + \underbrace{g_k^{(\pm 2)}}_{\text{ellipse 2}}. \quad (26.65)$$

The resulting ellipse $g_k^{(\pm 2)}$ has the frequency $m = 2$, but note that it is centered at a moving point on the “slower” ellipse (with frequency $m = 1$), that is, ellipse 2 effectively “rides” on ellipse 1. If we add FP_3 , its ellipse is again centered at a point on ellipse 2, and so on. For an illustration, see the examples in Figs. 26.11 and 26.12. In general, the ellipse for descriptor pair FP_j revolves around the (moving) center obtained as the superposition of $j - 1$ “slower” ellipses,

$$g_k^{(0)} + \sum_{m=1}^{j-1} g_k^{(\pm m)}. \quad (26.66)$$

Consequently, the curve obtained by the partial reconstruction from descriptor pairs $\text{FP}_1, \dots, \text{FP}_j$ (for $j \leq M \div 2$) is the point sequence

$$g_k^{(-j,\dots,j)} = g_k^{(0)} + \sum_{m=1}^j g_k^{(\pm m)}, \quad (26.67)$$

for $k = 0, \dots, M - 1$. The fully reconstructed shape is the sum of the centroid (defined by G_0) and $M \div 2$ ellipses, one for each Fourier descriptor pair $\text{FP}_1, \dots, \text{FP}_{M \div 2}$.

Even number of contour points

If M is even,¹⁴ then the reconstructed shape is a superposition of the centroid (defined by G_0), $(M - 1) \div 2$ ellipses from the Fourier descriptor pairs $\text{FP}_1, \dots, \text{FP}_{(M-1)\div 2}$, plus one additional *circle* specified by the single (highest frequency) Fourier coefficient $G_{M \div 2}$. The complete reconstruction from an even-length Fourier descriptor can thus be written as

$$g_k = \sum_{m=0}^{M-1} G_m \cdot e^{i \cdot 2\pi m \cdot \frac{k}{M}} = \underbrace{g_k^{(0)}}_{\text{center}} + \underbrace{\sum_{m=1}^{(M-1)\div 2} g_k^{(\pm m)}}_{(M-1)\div 2 \text{ ellipses}} + \underbrace{g_k^{(M \div 2)}}_{1 \text{ circle}}. \quad (26.68)$$

The single high-frequency circle associated with $g_k^{(M \div 2)}$ has its (moving) center at the sum of all lower-frequency ellipses that correspond to the Fourier coefficients G_{-m}, \dots, G_{+m} , with $m < (M \div 2)$.

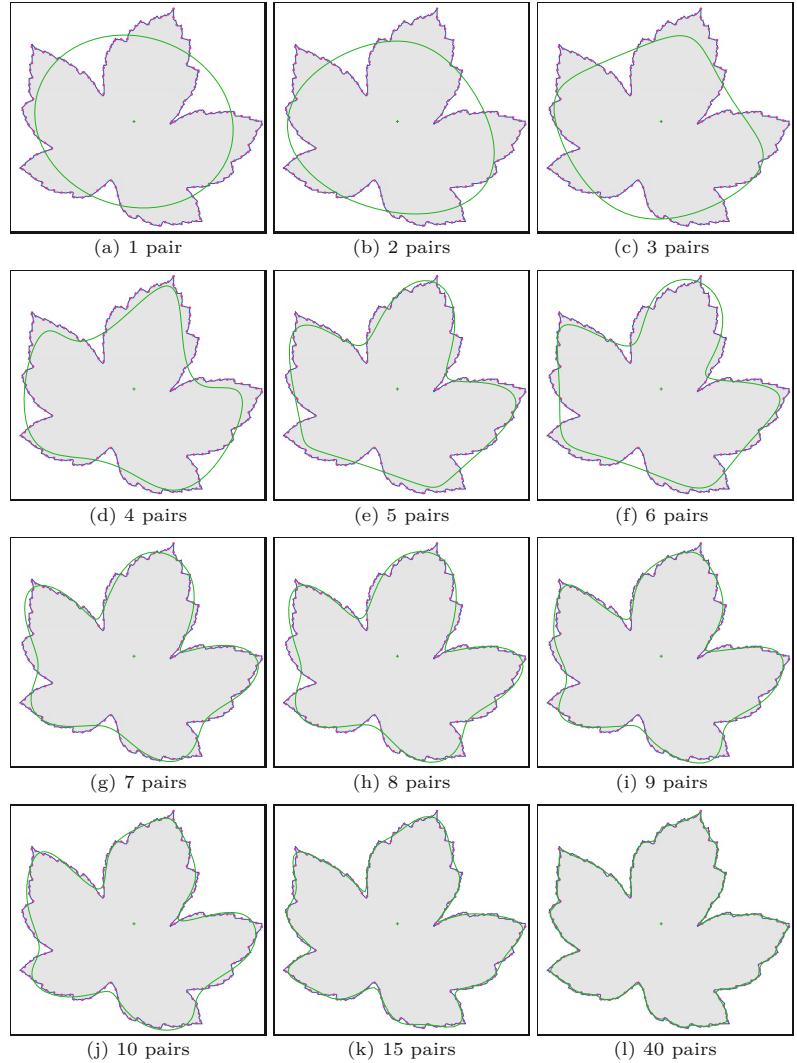
Reconstruction algorithm

Algorithm 26.4 describes the reconstruction of shapes from a Fourier descriptor using only a specified number (M_p) of Fourier descriptor pairs. The number of points on the reconstructed contour (N) can be freely chosen.

¹⁴ In this case, $M = 2 \cdot (M \div 2) = (M - 1) \div 2 + 1 + M \div 2$.

Fig. 26.10

Partial shape reconstruction from a limited set of Fourier descriptor pairs. The full descriptor contains 125 coefficients (G_0 plus 62 FD pairs).



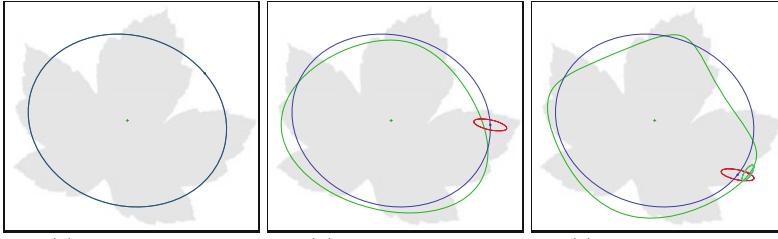
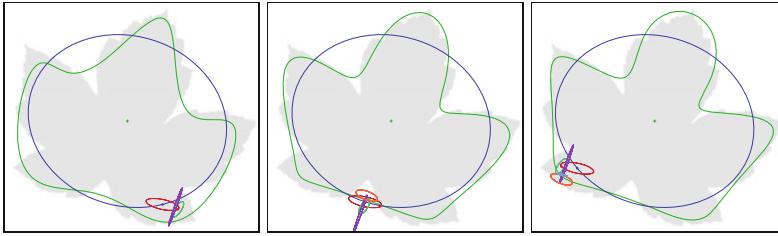
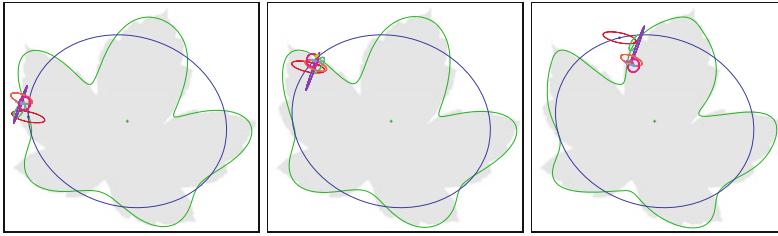
26.3.7 Fourier Descriptors from Unsampled Polygons

The requirement to distribute sample points uniformly along the contour path stems from classical signal processing and Fourier theory, where uniform sampling is a common assumption. However, as shown in [143] (see also [183, 262]), the Fourier descriptors for a polygonal shape can be calculated directly from the original polygon vertices without sub-sampling the contour. This “trigonometric” approach, described in the following, works for arbitrary (convex and non-convex) polygons.

We assume that the shape is specified as a sequence of P points $V = (\mathbf{v}_0, \dots, \mathbf{v}_{P-1})$, with $V(i) = \mathbf{v}_i = (x_i, y_i)$ representing the 2D vertices of a closed polygon. We define the quantities

$$\mathbf{d}(i) = \mathbf{v}_{(i+1) \bmod P} - \mathbf{v}_i \quad \text{and} \quad \lambda(i) = \|\mathbf{d}(i)\|, \quad (26.69)$$

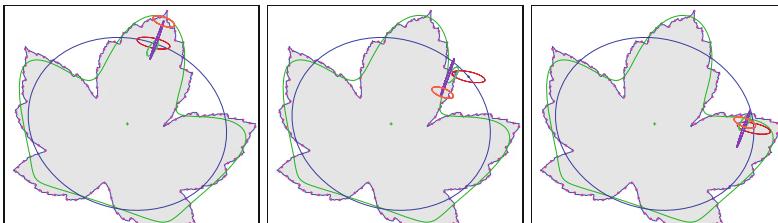
for $i = 0, \dots, P-1$, where $\mathbf{d}(i)$ is the vector representing the polygon segment between the vertices $\mathbf{v}_i, \mathbf{v}_{i+1}$, and $\lambda(i)$ is the length of that

(a) 1 pair, $t = 0.1$ (b) 2 pairs, $t = 0.2$ (c) 3 pairs, $t = 0.3$ (d) 4 pairs, $t = 0.4$ (e) 5 pairs, $t = 0.5$ (f) 6 pairs, $t = 0.6$ (g) 7 pairs, $t = 0.7$ (h) 8 pairs, $t = 0.8$ (i) 9 pairs, $t = 0.9$

26.3 GEOMETRIC INTERPRETATION OF FOURIER COEFFICIENTS

Fig. 26.11

Partial reconstruction by ellipse superposition (details). The green curve shows the partial reconstruction from $1, \dots, 9$ FD pairs. This curve performs one full revolution as the path parameter t runs from 0 to 1. Subfigures (a–i) depict the situation for $1, \dots, 9$ FD pairs and different path positions $t = 0.1, 0.2, \dots, 0.9$. Each Fourier descriptor pair corresponds to an ellipse that is centered at the current position t on the previous ellipse. The individual Fourier descriptor pair FP_1 in (a) corresponds to a single ellipse. In (b), the point for $t = 0.2$ on the blue ellipse (for FP_1) is the center of the red ellipse (for FP_2). In (c), the green ellipse (for FP_3) is centered at the point marked on the previous ellipse, and so on. The reconstructed shape is obtained by superposition of all ellipses. See Fig. 26.12 for a detailed view.

(a) $t = 0.0$ (b) $t = 0.1$ (c) $t = 0.2$ **Fig. 26.12**

Partial reconstruction by ellipse superposition (details). The green curve shows the partial reconstruction from 5 FD pairs $\text{FP}_1, \dots, \text{FP}_5$. This curve performs one full revolution as the path parameter t runs from 0 to 1. Subfigures (a–c) show the composition of the contour by superposition of the 5 ellipses, each corresponding to one FD pair, at selected positions $t = 0.0, 0.1, 0.2$. The blue ellipse corresponds to FP_1 and revolves once for $t = 0, \dots, 1$. The blue dot on this ellipse marks the position t , which serves as the center of the next (red) ellipse corresponding to FP_2 . This ellipse makes 2 revolutions for $t = 0, \dots, 1$ and the red dot for position t is again the center of green ellipse (for FP_3), and so on. Position t on the orange ellipse (for FP_1) coincides with the final reconstruction (green curve). The original contour was sampled at 125 equidistant points.

segment. We also define

$$L(i) = \sum_{j=0}^{i-1} \lambda(j), \quad (26.70)$$

for $i = 0, \dots, P$, which is the cumulative length of the polygon path from the start vertex v_0 to vertex v_i , such that $L(0)$ is zero and $L(P)$ is the closed path length of the polygon V .

26 FOURIER SHAPE DESCRIPTORS

Alg. 26.4

Partial shape reconstruction from a truncated Fourier descriptor \mathbf{G} . The shape is reconstructed by considering up to M_p Fourier descriptor pairs. The resulting sequence of contour points may be of arbitrary length (N). See Figs. 26.10–26.12 for examples.

```

1: GetPartialReconstruction( $\mathbf{G}, M_p, N$ )
   Input:  $\mathbf{G} = (G_0, \dots, G_{M-1})$ , Fourier descriptor with  $M$  coefficients;  $M_p$ , number of Fourier descriptor pairs to consider;  $N$ , number of points on the reconstructed shape. Returns the reconstructed contour as a sequence of  $N$  complex values.
2: Create map  $\mathbf{g}: [0, N-1] \rightarrow \mathbb{C}$ 
3:  $M \leftarrow |\mathbf{G}|$                                  $\triangleright$  total number of Fourier coefficients
4:  $M_p \leftarrow \min(M_p, (M-1) \div 2)$   $\triangleright$  available Fourier coefficient pairs
5: for  $k \leftarrow 0, \dots, N-1$  do
6:    $t \leftarrow k/N$                                  $\triangleright$  continuous path position  $t \in [0, 1]$ 
7:    $\mathbf{g}(k) \leftarrow \text{GetSinglePoint}(\mathbf{G}, -M_p, M_p, t)$        $\triangleright$  see below
8: return  $\mathbf{g}$ .
9: GetSinglePoint( $\mathbf{G}, m_-, m_+, t$ )
   Returns a single point (as a complex value) on the reconstructed shape for the continuous path position  $t \in [0, 1]$ , based on the Fourier coefficients  $\mathbf{G}(m_-), \dots, \mathbf{G}(m_+)$ .
10:  $M \leftarrow |\mathbf{G}|$ 
11:  $x \leftarrow 0, y \leftarrow 0$ 
12: for  $m \leftarrow m_-, \dots, m_+$  do
13:    $\phi \leftarrow 2 \cdot \pi \cdot m \cdot t$ 
14:    $G \leftarrow \mathbf{G}(m \bmod M)$ 
15:    $A \leftarrow \text{Re}(G), B \leftarrow \text{Im}(G)$ 
16:    $x \leftarrow x + A \cdot \cos(\phi) - B \cdot \sin(\phi)$ 
17:    $y \leftarrow y + A \cdot \sin(\phi) + B \cdot \cos(\phi)$ 
18: return  $(x + i \cdot y)$ .

```

For a (freely chosen) number of Fourier descriptor pairs (M_p), the corresponding Fourier descriptor $\mathbf{G} = (G_{-M_p}, \dots, G_0, \dots, G_{+M_p})$, has $2M_p + 1$ complex-valued coefficients G_m , where

$$G_0 = a_0 + i \cdot c_0 \quad (26.71)$$

and the remaining coefficients are calculated as

$$G_{+m} = (a_m + d_m) + i \cdot (c_m - b_m), \quad (26.72)$$

$$G_{-m} = (a_m - d_m) + i \cdot (c_m + b_m), \quad (26.73)$$

from the “trigonometric coefficients” a_m, b_m, c_m, d_m . As described in [143], these coefficients are obtained directly from the P polygon vertices \mathbf{v}_i as

$$\begin{pmatrix} a_0 \\ c_0 \end{pmatrix} = \mathbf{v}_0 + \frac{\sum_{i=0}^{P-1} \left[\frac{L^2(i+1) - L^2(i)}{2\lambda(i)} \cdot \mathbf{d}(i) + \lambda(i) \cdot \sum_{j=0}^{i-1} \mathbf{d}(j) - \mathbf{d}(i) \cdot \sum_{j=0}^{i-1} \lambda(j) \right]}{L(P)} \quad (26.74)$$

(representing the shape’s center), with \mathbf{d}, λ, L as defined in Eqns. (26.69) and (26.70). This can be simplified to

$$\begin{pmatrix} a_0 \\ c_0 \end{pmatrix} = \mathbf{v}_0 + \frac{\sum_{i=0}^{P-1} \left[\left(\frac{L^2(i+1) - L^2(i)}{2\lambda(i)} - L(i) \right) \cdot \mathbf{d}(i) + \lambda(i) \cdot (\mathbf{v}_i - \mathbf{v}_0) \right]}{L(P)}. \quad (26.75)$$

```

1: FourierDescriptorFromPolygon( $V, M_p$ )
   Input:  $V = (v_0, \dots, v_{P-1})$ , a sequence of  $P$  points representing
          the vertices of a closed 2D polygon;  $M_p$ , the desired number of
          FD pairs. Returns a new Fourier descriptor of length  $2M_p+1$ .
2:  $P \leftarrow |V|$                                  $\triangleright$  number of polygon vertices in  $V$ 
3:  $M \leftarrow 2 \cdot M_p + 1$                      $\triangleright$  number of Fourier coefficients in  $G$ 
4: Create maps  $\mathbf{d}: [0, P-1] \rightarrow \mathbb{R}^2$ ,  $\lambda: [0, P-1] \rightarrow \mathbb{R}$ ,
5:  $L: [0, P] \rightarrow \mathbb{R}$ ,  $G: [0, M-1] \rightarrow \mathbb{C}$ 
6:  $L(0) \leftarrow 0$ 
7: for  $i \leftarrow 0, \dots, P-1$  do
8:    $\mathbf{d}(i) \leftarrow V((i+1) \bmod P) - V(i)$             $\triangleright$  Eq. 26.69
9:    $\lambda(i) \leftarrow \|\mathbf{d}(i)\|$ 
10:   $L(i+1) \leftarrow L(i) + \lambda(i)$ 
11:   $\begin{pmatrix} a \\ c \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}$             $\triangleright a = a_0, c = c_0$ 
12:  for  $i \leftarrow 0, \dots, P-1$  do
13:     $s \leftarrow \frac{L^2(i+1) - L^2(i)}{2 \cdot \lambda(i)} - L(i)$ 
14:     $\begin{pmatrix} a \\ c \end{pmatrix} \leftarrow \begin{pmatrix} a \\ c \end{pmatrix} + s \cdot \mathbf{d}(i) + \lambda(i) \cdot (V(i) - V(0))$     $\triangleright$  Eq. 26.75
15:     $G(0) \leftarrow v_0 + \frac{1}{L(P)} \cdot \begin{pmatrix} a \\ c \end{pmatrix}$             $\triangleright$  Eq. 26.71
16:  for  $m \leftarrow 1, \dots, M_p$  do            $\triangleright$  for FD-pairs  $G_{\pm 1}, \dots, G_{\pm M_p}$ 
17:     $\begin{pmatrix} a \\ c \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} b \\ d \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}$             $\triangleright a_m, b_m, c_m, d_m$ 
18:    for  $i \leftarrow 0, \dots, P-1$  do
19:       $\omega_0 \leftarrow 2\pi m \cdot \frac{L(i)}{L(P)}$ 
20:       $\omega_1 \leftarrow 2\pi m \cdot \frac{L((i+1) \bmod P)}{L(P)}$ 
21:       $\begin{pmatrix} a \\ c \end{pmatrix} \leftarrow \begin{pmatrix} a \\ c \end{pmatrix} + \frac{\cos(\omega_1) - \cos(\omega_0)}{\lambda(i)} \cdot \mathbf{d}(i)$             $\triangleright$  Eq. 26.76
22:       $\begin{pmatrix} b \\ d \end{pmatrix} \leftarrow \begin{pmatrix} b \\ d \end{pmatrix} + \frac{\sin(\omega_1) - \sin(\omega_0)}{\lambda(i)} \cdot \mathbf{d}(i)$             $\triangleright$  Eq. 26.77
23:     $G(m) \leftarrow \frac{L(P)}{(2\pi m)^2} \cdot \begin{pmatrix} a+d \\ c-b \end{pmatrix}$             $\triangleright$  Eq. 26.72
24:     $G(-m \bmod M) \leftarrow \frac{L(P)}{(2\pi m)^2} \cdot \begin{pmatrix} a-d \\ c+b \end{pmatrix}$             $\triangleright$  Eq. 26.73
25:  return  $G$ .

```

26.3 GEOMETRIC INTERPRETATION OF FOURIER COEFFICIENTS

Alg. 26.5

Fourier descriptor from trigonometric data (arbitrary polygons). Parameter M_p specifies the number of Fourier coefficient pairs.

The remaining coefficients a_m, b_m, c_m, d_m ($m = 1, \dots, M_p$) are calculated as

$$\begin{pmatrix} a_m \\ c_m \end{pmatrix} = \frac{L(P)}{(2\pi m)^2} \cdot \sum_{i=0}^{P-1} \left[\frac{\cos(2\pi m \frac{L(i+1)}{L(P)}) - \cos(2\pi m \frac{L(i)}{L(P)})}{\lambda(i)} \cdot \mathbf{d}(i) \right], \quad (26.76)$$

$$\begin{pmatrix} b_m \\ d_m \end{pmatrix} = \frac{L(P)}{(2\pi m)^2} \cdot \sum_{i=0}^{P-1} \left[\frac{\sin(2\pi m \frac{L(i+1)}{L(P)}) - \sin(2\pi m \frac{L(i)}{L(P)})}{\lambda(i)} \cdot \mathbf{d}(i) \right], \quad (26.77)$$

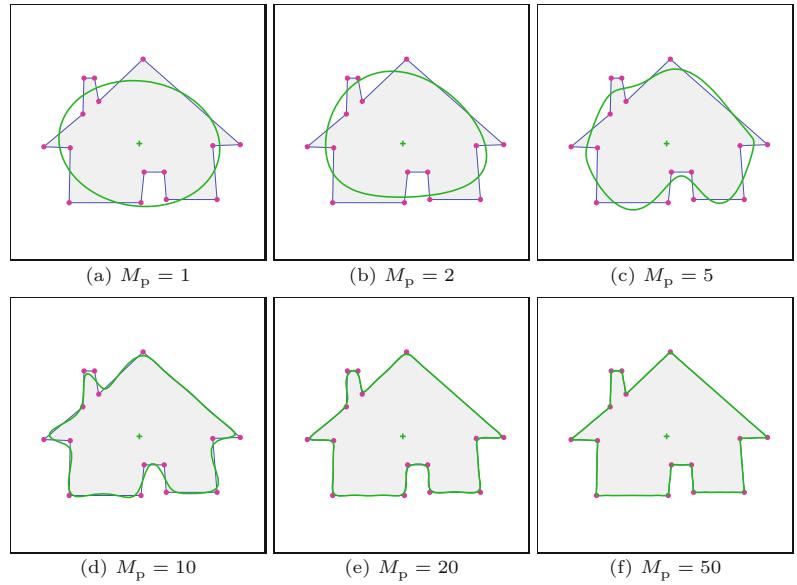
respectively. The complete calculation of a Fourier descriptor from trigonometric coordinates (i.e., from arbitrary polygons) is summarized in Alg. 26.5.

An approximate reconstruction of the original shape can be obtained directly from the trigonometric coefficients a_m, b_m, c_m, d_m de-

26 FOURIER SHAPE DESCRIPTORS

Fig. 26.13

Fourier descriptors calculated from trigonometric data (arbitrary polygons). Shape reconstructions with different numbers of Fourier descriptor pairs (M_p).



fined in Eqns. (26.75) and (26.76) as¹⁵

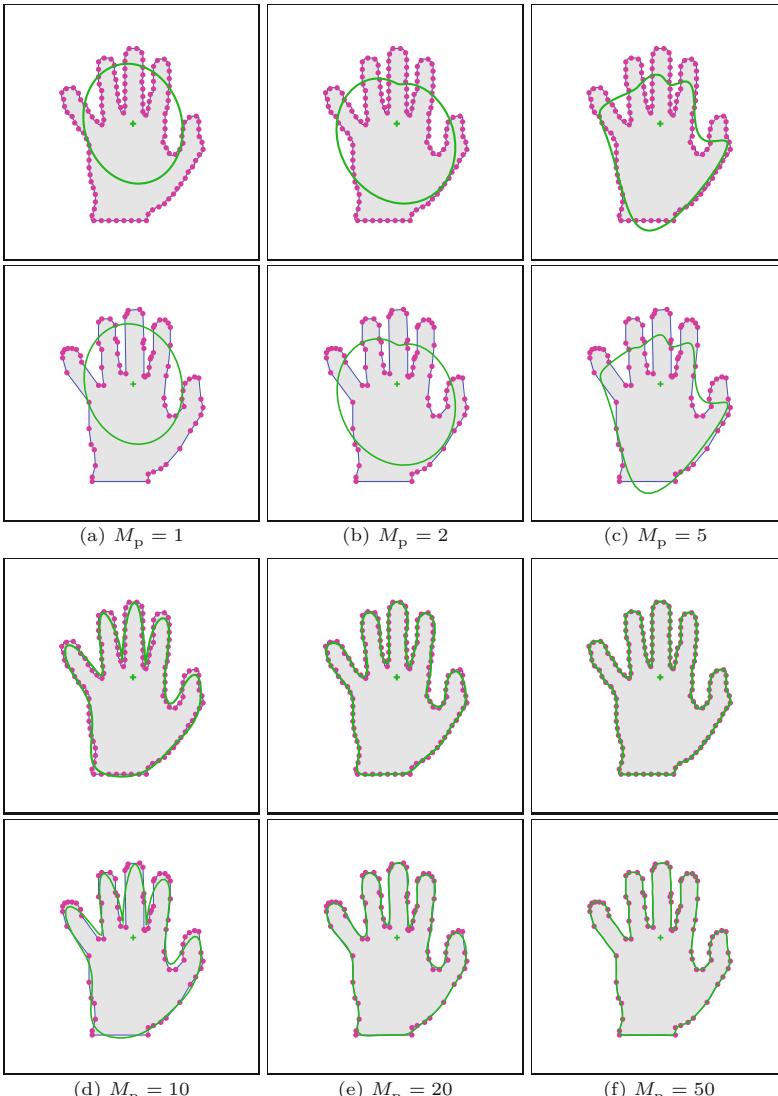
$$\mathbf{x}(t) = \begin{pmatrix} a_0 \\ c_0 \end{pmatrix} + \sum_{m=1}^{M_p} \left[\begin{pmatrix} a_m \\ c_m \end{pmatrix} \cdot \cos(2\pi m t) + \begin{pmatrix} b_m \\ d_m \end{pmatrix} \cdot \sin(2\pi m t) \right], \quad (26.78)$$

for $t = 0, \dots, 1$. Of course, this reconstruction can also be calculated from the actual DFT coefficients \mathbf{G} , as described in Eqn. (26.20). Again the reconstruction error is reduced by increasing the number of Fourier descriptor pairs (M_p), as demonstrated in Fig. 26.13.¹⁶ The reconstruction is theoretically perfect as M_p goes to infinity.

Working with the trigonometric technique is an advantage, in particular, if the boundary curvature along the outline varies strongly. For example, the silhouette of a human hand typically exhibits high curvature along the fingertips while other contour sections are almost straight. Capturing the high-curvature parts requires a significantly higher density of samples than in the smooth sections, as illustrated in Fig. 26.14. This figure compares the partial shape reconstructions obtained from Fourier descriptors calculated with uniform and non-uniform contour sampling, using identical numbers of Fourier descriptor pairs (M_p). Note that the coefficients (and thus the reconstructions) are very similar, although considerably fewer samples were used for the trigonometric approach.

¹⁵ Note the analogy to the elliptical reconstruction in Eqns. (26.47) and (26.48).

¹⁶ Most test images used in this chapter were taken from the Kimia dataset [134]. A selected subset of modified images taken from this dataset is available on the book's website.



26.4 EFFECTS OF GEOMETRIC TRANSFORMATIONS

Fig. 26.14
Fourier descriptors from uniformly sampled vs. non-uniformly sampled (trigonometric) contours. Partial constructions from Fourier descriptors obtained from uniformly sampled contours (rows 1, 3) and non-uniformly sampled contours (rows 2, 4), for different numbers of Fourier descriptor pairs (M_p).

26.4 Effects of Geometric Transformations

To be useful for comparing shapes, a representation should be invariant against a certain set of geometric transformations. Typically, a minimal requirement for robust 2D shape matching is invariance to translation, scale changes, and rotation. Fourier shape descriptors in their basic form are *not* invariant under any of these transformations but they can be modified to satisfy these requirements. In this section, we discuss the effects of such transformations upon the corresponding Fourier descriptors. The steps involved for making Fourier descriptors invariant are discussed subsequently in Sec. 26.5.

26.4.1 Translation

As described in Sec. 26.3.1, the coefficient G_0 of a Fourier descriptor \mathbf{G} corresponds to the centroid of the encoded contour. Moving the

points g_k of a shape \mathbf{g} in the complex plane by some constant $z \in \mathbb{C}$,

$$g'_k = g_k + z, \quad (26.79)$$

for $k = 0, \dots, M-1$, only affects Fourier coefficient G_0 , that is,

$$G'_m = \begin{cases} G_m + z & \text{for } m = 0, \\ G_m & \text{for } m \neq 0. \end{cases} \quad (26.80)$$

To make an FD invariant against translation, it is thus sufficient to zero its G_0 coefficient, thereby shifting the shape's center to the origin of the coordinate system. Alternatively, translation invariant matching of Fourier descriptors is achieved by simply ignoring coefficient G_0 .

26.4.2 Scale Change

Since the Fourier transform is a linear operation, scaling a 2D shape \mathbf{g} uniformly by a real-valued factor s ,

$$g'_k = s \cdot g_k, \quad (26.81)$$

also scales the corresponding Fourier spectrum by the same factor, that is,

$$G'_m = s \cdot G_m, \quad (26.82)$$

for $m = 1, \dots, M-1$. Note that scaling by $s = -1$ (or any other negative factor) corresponds to *reversing* the ordering of the samples along the contour (see also Sec. 26.4.6). Given the fact that the DFT coefficient G_1 represents a circle whose radius $r_1 = |G_1|$ is proportional to the size of the original shape (see Sec. 26.3.2), the Fourier descriptor \mathbf{G} could be normalized for scale by setting

$$G_m^S = \frac{1}{|G_1|} \cdot G_m, \quad (26.83)$$

for $m = 1, \dots, M-1$, such that $|G_1^S| = 1$. Although it is common to use only G_1 for scale normalization, this coefficient may be relatively small (and thus unreliable) for certain shapes. We therefore prefer to normalize the complete Fourier coefficient vector to achieve scale invariance (see Sec. 26.5.1).

26.4.3 Rotation

If a given shape is rotated about the origin by some angle β , then each contour point $\mathbf{v}_k = (x_k, y_k)$ moves to a new position

$$\mathbf{v}'_k = \begin{pmatrix} x'_k \\ y'_k \end{pmatrix} = \begin{pmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{pmatrix} \cdot \begin{pmatrix} x_k \\ y_k \end{pmatrix}. \quad (26.84)$$

If the 2D contour samples are represented as complex values $g_k = x_k + i \cdot y_k$, this rotation can be expressed as a multiplication

$$g'_k = e^{i\beta} \cdot g_k, \quad (26.85)$$

with the complex factor $e^{i\beta} = \cos(\beta) + i \cdot \sin(\beta)$. As in Eqn. (26.82), we can use the linearity of the DFT to predict the effects of rotating the shape \mathbf{g} by angle β as

$$G'_m = e^{i\beta} \cdot G_m, \quad (26.86)$$

for $m = 0, \dots, M-1$. Thus, the spatial rotation in Eqn. (26.85) multiplies each DFT coefficient G_m by the *same* complex factor $e^{i\beta}$, which has unit magnitude. Since

$$e^{i\beta} \cdot G_m = e^{i(\theta_m + \beta)} \cdot |G_m|, \quad (26.87)$$

this only rotates the *phase* $\theta_m = \angle G_m$ of each coefficient by the *same* angle β , without changing its *magnitude* $|G_m|$.

26.4.4 Shifting the Sampling Start Position

Despite the implicit periodicity of the boundary sequence and the corresponding DFT spectrum, Fourier descriptors are generally not the same if sampling starts at different positions along the contour. Given a periodic sequence of M discrete contour samples $\mathbf{g} = (g_0, g_1, \dots, g_{M-1})$, we select another sequence $\mathbf{g}' = (g'_0, g'_1, \dots) = (g_{k_s}, g_{k_s+1}, \dots)$, again of length M , from the same set of samples but starting at point k_s , that is,

$$g'_k = g_{(k+k_s) \bmod M}. \quad (26.88)$$

This is equivalent to *shifting* the original signal \mathbf{g} circularly by $-k_s$ positions. The well-known “shift property” of the Fourier transform¹⁷ states that such a change to the “signal” \mathbf{g} modifies the corresponding DFT coefficients G_m (for the original contour sequence) to

$$G'_m = e^{i \cdot m \cdot \frac{2\pi k_s}{M}} \cdot G_m = e^{i \cdot m \cdot \varphi_s} \cdot G_m, \quad (26.89)$$

where $\varphi_s = \frac{2\pi k_s}{M}$ is a constant phase angle that is obviously proportional to the chosen start position k_s . Note that, in Eqn. (26.89), each DFT coefficient G_m is multiplied by a *different* complex quantity $e^{i \cdot m \cdot \varphi_s}$, which is of unit magnitude and varies with the frequency index m . In other words, the *magnitude* of any DFT coefficient G_m is again preserved but its *phase* changes individually. The coefficients of any Fourier descriptor pair $\text{FP}_m = (G_{-m}, G_{+m})$ thus become

$$G'_{-m} = e^{-i \cdot m \cdot \varphi_s} \cdot G_{-m} \quad \text{and} \quad G'_{+m} = e^{i \cdot m \cdot \varphi_s} \cdot G_{+m}, \quad (26.90)$$

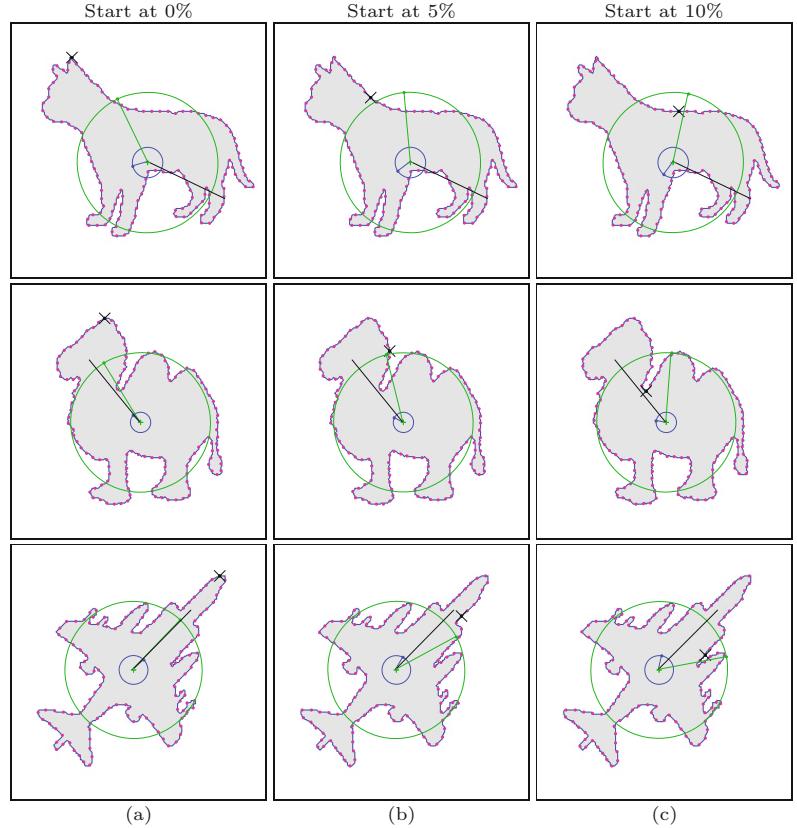
that is, coefficient G_{-m} is rotated by the angle $-m \cdot \varphi_s$ and G_{+m} is rotated by $m \cdot \varphi_s$. In other words, a circular shift of the signal by $-k_s$ samples rotates the coefficients G_{-m}, G_{+m} by the same angle $m \cdot \varphi_s$ but in *opposite* directions. Therefore, the sum of both angles stays the same, that is,

$$\angle G'_{-m} + \angle G'_{+m} \equiv \angle G_{-m} + \angle G_{+m}. \quad (26.91)$$

¹⁷ See Chapter 18, Sec. 18.1.6.

Fig. 26.15

Effects of choosing different start points for contour sampling. The start point (marked \times on the contour) is set to 0%, 5%, 10% of the contour path length. The blue and green circles represent the partial reconstruction from single DFT coefficients G_{-1} and G_{+1} , respectively. The dot on each circle and the associated radial line shows the phase of the corresponding coefficient. The black line indicates the average orientation $(\angle G_{-1} + \angle G_{+1})/2$. It can be seen that the phase difference of G_{-1} and G_{+1} is directly related to the start position, but the average orientation (black line) remains unchanged.



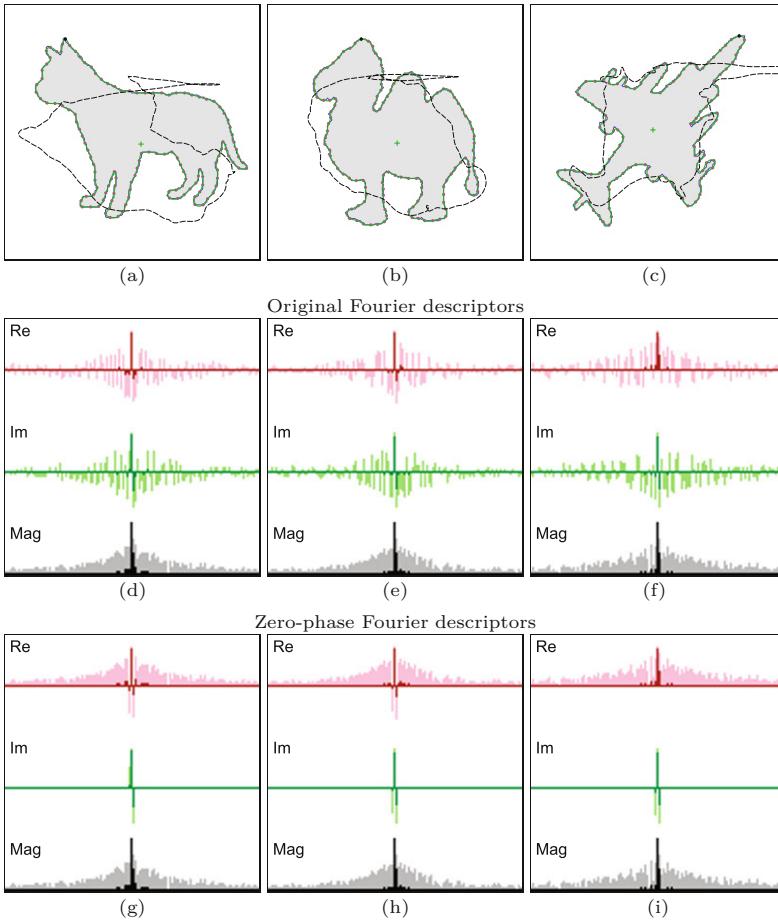
In particular, we see from Eqn. (26.90) that shifting the start position modifies the coefficients of the *first* descriptor pair $FP_1 = (G_{-1}, G_{+1})$ to

$$G'_{-1} = e^{-i\cdot\varphi_s} \cdot G_{-1} \quad \text{and} \quad G'_{+1} = e^{i\cdot\varphi_s} \cdot G_{+1}. \quad (26.92)$$

The resulting *absolute* phase change of the coefficients G_{-1}, G_{+1} is $-\varphi_s, +\varphi_s$, respectively, and thus the change in phase *difference* is $2 \cdot \varphi_s$, that is, the phase difference between the coefficients G_{-1}, G_{+1} is proportional to the chosen start position k_s (see Fig. 26.15).

26.4.5 Effects of Phase Removal

As described in the two previous sections, shape rotation (Sec. 26.4.3) and shift of start point (Sec. 26.4.4) both affect the phase of the Fourier coefficients but not their magnitude. The fact that magnitude is preserved suggests a simple solution for rotation invariant shape matching by simply ignoring the phase of the coefficients and comparing only their magnitude (see Sec. 26.6). Although this comes at the price of losing shape descriptiveness, magnitude-only descriptors are often used for shape matching. Clearly, the original shape cannot be reconstructed from a magnitude-only Fourier descriptor, as demonstrated in Fig. 26.16. It shows the reconstruction of shapes from Fourier descriptors with the phase of all coefficients set to zero, except for G_{-1}, G_0 and G_{+1} (to preserve the shape's center and main orientation).



26.4 EFFECTS OF GEOMETRIC TRANSFORMATIONS

Fig. 26.16

Effects of removing phase information. Original shapes and reconstruction after phase removal (a–c). Original Fourier coefficients (d–f) and zero-phase coefficients (g–i). The red and green plots in (d–i) show the real and imaginary components, respectively; gray plots show the coefficient magnitude. Dark-shaded bars correspond to the actual values, light-shaded bars are logarithmic values. The magnitude of the coefficients in (d–f) is the same as in (g–i).

26.4.6 Direction of Contour Traversal

If the traversal direction of the contour samples is reversed, the coefficients of all Fourier descriptor pairs are exchanged, that is,

$$G'_m = G_{-m \bmod M}. \quad (26.93)$$

This is equivalent to scaling the original shape by $s = -1$, as pointed out in Section 26.4.2. However, this is typically of no relevance in matching, since we can specify all contours to be sampled in either clockwise or counter-clockwise direction.

26.4.7 Reflection (Symmetry)

Mirroring or reflecting a contour about the x -axis is equivalent to replacing each complex-valued point $g_k = x_k + i \cdot y_k$ by its *complex conjugate* g_k^* , that is,

$$g'_k = g_k^* = x_k - i \cdot y_k. \quad (26.94)$$

This change to the “signal” results in a modified DFT spectrum with coefficients

$$G'_m = G_{-m \bmod M}^*, \quad (26.95)$$

26 FOURIER SHAPE DESCRIPTORS

Table 26.1
Effects of spatial transformations upon the corresponding DFT spectrum. The original contour samples are denoted g_k , the DFT coefficients are G_m .

Operation	Contour samples	DFT coefficients
Forward transformation	g_k , for $k = 0, \dots, M-1$	$G_m = \frac{1}{M} \sum_{k=0}^{M-1} g_k \cdot e^{-i2\pi m \frac{k}{M}}$
Inverse transformation	$g_k = \sum_{m=0}^{M-1} G_m \cdot e^{i2\pi m \frac{k}{M}}$	G_m , for $m = 0, \dots, M-1$
Translation (by $z \in \mathbb{C}$)	$g'_k = g_k + z$	$G'_m = \begin{cases} G_m + z & \text{for } m = 0 \\ G_m & \text{otherwise} \end{cases}$
Uniform scaling (by $s \in \mathbb{R}$)	$g'_k = s \cdot g_k$	$G'_m = s \cdot G_m$
Rotation about the origin (by β)	$g'_k = e^{i \cdot \beta} \cdot g_k$	$G'_m = e^{i \cdot \beta} \cdot G_m$
Shift of start position (by k_s)	$g'_k = g_{(k+k_s) \bmod M}$	$G'_m = e^{i \cdot m \cdot \frac{2\pi k_s}{M}} \cdot G_m$
Direction of contour traversal	$g'_k = g_{-k \bmod M}$	$G'_m = G_{-m \bmod M}$
Reflection about the x -axis	$g'_k = g_k^*$	$G'_m = G_{-m \bmod M}^*$

where G^* denotes the complex conjugate of the original DFT coefficients. Reflections about arbitrary axes can be described in the same way with additional rotations. Fourier descriptors can be made invariant against reflections, such that symmetric contours map to equivalent descriptors [245]. Note, however, that invariance to symmetry is not always desirable, for example, for distinguishing the silhouettes of left and right hands.

The relations between 2D point coordinates and the Fourier spectrum, as well as the effects of the aforementioned geometric shape transformations upon the DFT coefficients are compactly summarized in [Table 26.1](#).

26.5 Transformation-Invariant Fourier Descriptors

As mentioned already, making a Fourier descriptor invariant to *translation* or absolute shape position is easy because the only affected spectral coefficient is G_0 . Thus, setting coefficient G_0 to zero implicitly moves the center of the corresponding shape to the coordinate origin and thus creates a descriptor that is invariant to shape translation.

Invariance against a change in *scale* is also a simple issue because it only multiplies the magnitude of all Fourier coefficients by the same real-valued scale factor, which can be easily normalized.

A more challenging task is to make Fourier descriptors invariant against shape *rotation* and shift of the contour *starting point*, because they jointly affect the phase of the Fourier coefficients. If matching is to be based on the complex-valued Fourier descriptors (not on coefficient magnitude only) to achieve better shape discrimination, the phase changes introduced by shape rotation and start point shifts must be eliminated first. However, due to noise and possible ambiguities, this is not a trivial problem (see also [183, 184, 189, 245]).

26.5.1 Scale Invariance

As mentioned in Section 26.4.2, the magnitude G_{+1} is often used as a reference to normalize for scale, since G_{+1} is typically (though not always) the Fourier coefficient with the largest magnitude. Alternatively, one could use the size of the fundamental ellipse, defined by the Fourier descriptor pair FP_1 , to measure the overall scale, for example, by normalizing to

$$G_m^S \leftarrow \frac{1}{|G_{-1}| + |G_{+1}|} \cdot G_m, \quad (26.96)$$

which normalizes the *length* of the major axis $a_1 = |G_{-1}| + |G_{+1}|$ (see Eqn. (26.57)) of the fundamental ellipse to unity. Another alternative is

$$G_m^S \leftarrow \frac{1}{(|G_{-1}| \cdot |G_{+1}|)^{1/2}} \cdot G_m, \quad (26.97)$$

which normalizes the *area* of the fundamental ellipse. Since all variants in Eqns. (26.83), (26.96) and (26.97) scale the coefficients G_m by a fixed (real-valued) factor, the shape information contained in the Fourier descriptor remains unchanged.

There are shapes, however, where coefficients G_{+1} and/or G_{-1} are small or almost vanish to zero, such that they are not always a reliable reference for scale. An obvious solution is to include the complete set of Fourier coefficients by standardizing the *norm* of the coefficient vector \mathbf{G} to unity in the form

$$G_m^S \leftarrow \frac{1}{\|\mathbf{G}\|} \cdot G_m, \quad (26.98)$$

(assuming that $G_0 = 0$). In general, the L_2 norm of a complex-valued vector $Z = (z_0, z_1, \dots, z_{M-1})$, $z_i \in \mathbb{C}$, is defined as

$$\|Z\| = \left(\sum_{i=1}^{M-1} |z_i|^2 \right)^{1/2} = \left(\sum_{i=1}^{M-1} \operatorname{Re}(z_i)^2 + \operatorname{Im}(z_i)^2 \right)^{1/2}. \quad (26.99)$$

Scaling the vector Z by the reciprocal of its norm yields a vector with unit norm, that is,

$$\left\| \frac{1}{\|Z\|} \cdot Z \right\| = 1. \quad (26.100)$$

To normalize a given Fourier descriptor \mathbf{G} , we use all elements except G_0 (which relates to the absolute position of the shape and is not relevant for its shape). The following substitution makes \mathbf{G} scale invariant by normalizing the remaining sub-vector $(G_1, G_2, \dots, G_{M-1})$ to

$$G_m^S \leftarrow \begin{cases} G_m & \text{for } m = 0, \\ \frac{1}{\sqrt{\nu}} \cdot G_m & \text{for } 1 \leq m < M, \end{cases} \quad \text{with } \nu = \sum_{m=1}^{M-1} |G_m|^2. \quad (26.101)$$

See procedure `MakeScaleInvariant(\mathbf{G})` in Alg. 26.6 (lines 7–15) for a summary of this step.

26.5.2 Start Point Invariance

As discussed in Sections 26.4.3 and 26.4.4, respectively, shape rotation and shift of start point both affect the phase of the Fourier coefficients in a combined manner, without altering their magnitude. In particular, if the shape is rotated by some angle β (see Eqn. (26.89)) and the start position is shifted by k_s samples (see Eqn. (26.86)), then each Fourier coefficient G_m is modified to

$$G'_m = e^{i \cdot \beta} \cdot e^{i \cdot m \cdot \varphi_s} \cdot G_m = e^{i \cdot (\beta + m \cdot \varphi_s)} \cdot G_m, \quad (26.102)$$

where $\varphi_s = 2\pi k_s / M$ is the corresponding *start point phase*. Thus, the incurred phase shift is not only different for each coefficient but simultaneously depends on the rotation angle β and the start point phase φ_s . Normalization in this case means to remove these phase shifts, which would be straightforward if β and φ_s were known. We derive these two parameters one after the other, starting with the calculation of the start point phase φ_s , which we describe in this section, followed by the estimation of the rotation β , shown subsequently in Section 26.5.3.

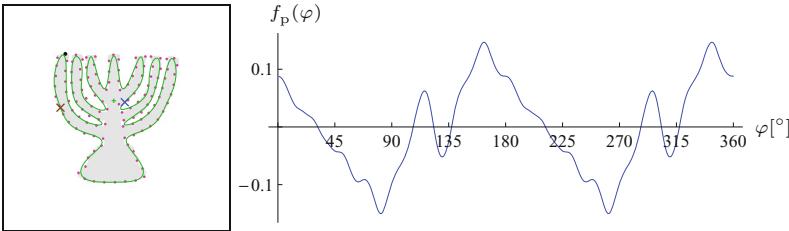
To normalize the Fourier descriptor of a particular shape to a “canonical” start point, we need a quantity that can be calculated from the Fourier spectrum and only depends on the start point phase φ_s but is independent of the rotation β . From Eqn. (26.90) and Fig. 26.15 we see that the phase *difference* within any Fourier descriptor pair (G_{-m}, G_{+m}) is proportional to the start point phase φ_s and independent to shape rotation β , since the latter rotates all coefficients by the same angle. Thus, we look for a quantity that depends only on the phase *differences* within Fourier descriptor pairs. This is accomplished, for example, by the function

$$f_p(\varphi) = \sum_{m=1}^{M_p} [e^{-i \cdot m \cdot \varphi} \cdot G_{-m}] \otimes [e^{i \cdot m \cdot \varphi} \cdot G_m], \quad (26.103)$$

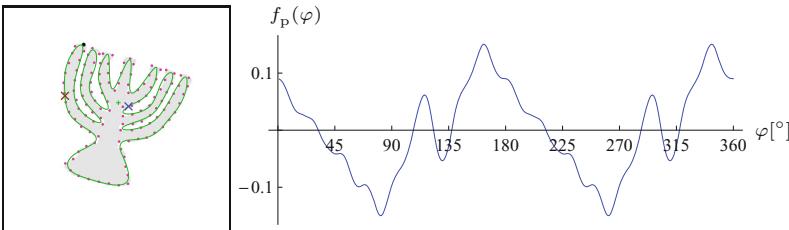
where parameter φ is an arbitrary start point phase, M_p is the number of coefficient pairs, and \otimes denotes the “cross product” between two Fourier coefficients.¹⁸ Given a particular start point phase φ , the function in Eqn. (26.103) yields the sum of the cross products of each coefficient pair (G_{-m}, G_m) , for $m = 1, \dots, M_p$. If each of the complex-valued coefficients is interpreted as a vector in the 2D plane, the magnitude of their cross product is proportional to the *area* of the enclosed parallelogram. The enclosed area is potentially large only if *both* vectors are of significant length, which means that the corresponding ellipse has a distinct eccentricity and orientation. Note that the sign of the cross product may be positive or negative and depends on the relative orientation or “handedness” of the two vectors.

Since the function $f_p(\varphi)$ is based only on the *relative* orientation (phase) of the involved coefficients, it is invariant to a shape rotation

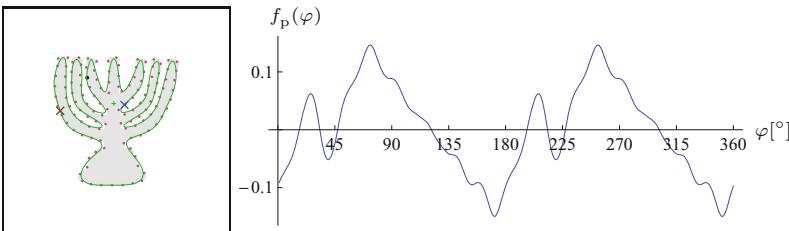
¹⁸ In analogy to 2D vector notation, we define the “cross product” of two complex quantities $z_1 = (a_1, b_1)$ and $z_2 = (a_2, b_2)$ as $z_1 \otimes z_2 = a_1 \cdot b_2 - b_1 \cdot a_2 = |z_1| \cdot |z_2| \cdot \sin(\theta_2 - \theta_1)$. See also Sec. B.3.3 in the Appendix.



(a) rotation $\theta = 0^\circ$, start point phase $\varphi_s = 0^\circ$



(b) rotation $\theta = 15^\circ$, start point phase $\varphi_s = 0^\circ$



(c) rotation $\theta = 0^\circ$, start point phase $\varphi_s = 90^\circ$

26.5 TRANSFORMATION-INVARIANT FOURIER DESCRIPTORS

Table 26.2

Plot of the function $f_p(\varphi)$ used for start point normalization. In the figures on the left, the real start point is marked by a black dot. The normalized start points φ_A and $\varphi_B = \varphi_A + \pi$ are marked by a blue and a brown cross, respectively. They correspond to the two peak positions of the function $f_p(\varphi)$, as defined in Eqn. (26.103), separated by a fixed phase shift of $\pi = 180^\circ$ (right). The function is invariant under shape rotation, as demonstrated in (b), where the shape is rotated by 15° but sampled from the same start point as in (a). However, the phase of $f_p(\varphi)$ is proportional to the start point shift, as shown in (c), where the start point is chosen at 25% ($\varphi_s = 90^\circ$) of the boundary path length. The functions were calculated after scale normalization, using $M_p = 25$ Fourier coefficient pairs.

β , which shifts all coefficients by the same angle (see Eqn. (26.86)). As shown in Fig. 26.2, $f_p(\varphi)$ is periodic with π and its phase is proportional to the actual start point shift. We choose the angle φ that *maximizes* $f_p(\varphi)$ as the “canonical” start point phase φ_A , that is,

$$\varphi_A = \operatorname{argmax}_{0 \leq \varphi < \pi} f_p(\varphi). \quad (26.104)$$

However, since $f_p(\varphi) = f_p(\varphi + \pi)$, there is also a *second* candidate phase

$$\varphi_B = \varphi_A + \pi, \quad (26.105)$$

displaced by $\pi = 180^\circ$. The two “canonical” start points corresponding to φ_A and φ_B , respectively, are marked on the reconstructed shapes in Fig. 26.2. Although it might seem easy at first to resolve this 180° ambiguity of the start point phase, this turns out to be difficult to achieve in general from the Fourier coefficients alone. Several functions have been proposed for this purpose that work well for certain shapes but fail on others, including the “positive real energy” function suggested in [245]. In particular, any decision based on the magnitude or phase of a *single* coefficient (or a single coefficient pair) must eventually fail, since none of the coefficients is guaranteed to have a significant magnitude. With vanishing coefficient magnitude,

phase measurements become unreliable and may be very susceptible to noise.

The complete process of start point normalization is summarized in Alg. 26.7. The start point phase φ_A is found numerically by evaluating the function $f_p(\varphi)$ at 400 discrete steps for $\varphi = 0, \dots, \pi$ (lines 6–16). For practical use, this exhaustive method should be substituted by a more efficient and accurate optimization technique (for example, using Brent’s method [190, Ch. 10]).¹⁹ Given the estimated start point phase φ_A for the Fourier descriptor \mathbf{G} , two normalized versions $\mathbf{G}^A, \mathbf{G}^B$ are calculated as

$$\begin{aligned}\mathbf{G}^A: G_m^A &\leftarrow G_m \cdot e^{i \cdot m \cdot \varphi_A}, \\ \mathbf{G}^B: G_m^B &\leftarrow G_m \cdot e^{i \cdot m \cdot (\varphi_A + \pi)},\end{aligned}\quad (26.106)$$

for $m = -M_p, \dots, M_p, m \neq 0$. Note that start point normalization does not require the Fourier descriptor \mathbf{G} to be normalized for translation and scale (see Sec. 26.5.1).

26.5.3 Rotation Invariance

After normalizing for starting point, the orientation of the fundamental ellipse (formed by the descriptor pair (G_{-1}, G_{+1})) could be assumed to be a reliable reference for global shape rotation. However, for certain shapes (e.g., regular polyhedra with an even number of faces), G_{-1} may vanish. Therefore, we recover the overall shape orientation from the vector obtained as the weighted sum of *all* Fourier coefficients, that is,

$$z = \sum_{m=1}^{M_p} \frac{1}{m} \cdot (G_{-m} + G_{+m}), \quad (26.107)$$

where the $1/m$ serves as a weighting factor, giving stronger emphasis to the low-frequency coefficients and attenuating the influence of the high-frequency coefficients. The resulting shape orientation estimate is

$$\beta = \text{arg} z = \tan^{-1} \left(\frac{\text{Im}(z)}{\text{Re}(z)} \right). \quad (26.108)$$

To normalize $\mathbf{G}^A, \mathbf{G}^B$ (obtained in Eqn. (26.106)) for shape orientation, we rotate each coefficient (except G_0) by $-\beta$, that is,

$$\begin{aligned}\mathbf{G}^A: G_m^A &\leftarrow G_m^A \cdot e^{-i \cdot \beta}, \\ \mathbf{G}^B: G_m^B &\leftarrow G_m^B \cdot e^{-i \cdot \beta},\end{aligned}\quad (26.109)$$

for $m = -M_p, \dots, M_p, m \neq 0$. For a summary of these steps, see procedure `MakeRotationInvariant(\mathbf{G})` in Alg. 26.6 (lines 16–24).

¹⁹ The accompanying Java implementation uses the class `BrentOptimizer` from the *Apache Commons Math library* [4] for this purpose.

```

1: Makelnvariant( $G$ )
   Input:  $G$ , Fourier descriptor with  $M_p$  coefficient pairs.
   Returns a pair of normalized Fourier descriptors  $G^A$ ,  $G^B$ , with
   a start point phase offset by  $180^\circ$ .
2: MakeScaleInvariant( $G$ )                                 $\triangleright$  see below
3:  $(G^A, G^B) \leftarrow \text{MakeStartPointInvariant}(G)$      $\triangleright$  see Alg. 26.7
4: MakeRotationInvariant( $G^A$ )                             $\triangleright$  see below
5: MakeRotationInvariant( $G^B$ )
6: return  $(G^A, G^B)$ .

```

```

7: MakeScaleInvariant( $G$ )
   Modifies  $G$  by unifying its norm and returns the scale factor  $\nu$ .
8:  $s \leftarrow 0$                                           $\triangleright s \in \mathbb{R}$ 
9: for  $m \leftarrow 1, \dots, M_p$  do
10:    $s \leftarrow s + |G(-m)|^2 + |G(m)|^2$ 
11:    $\nu \leftarrow 1/\sqrt{s}$ 
12:   for  $m \leftarrow 1, \dots, M_p$  do
13:      $G(-m) \leftarrow \nu \cdot G(-m)$ 
14:      $G(m) \leftarrow \nu \cdot G(m)$ 
15:   return  $\nu$ .

```

```

16: MakeRotationInvariant( $G$ )
   Modifies  $G$  and returns the estimated rotation angle  $\beta$ .
17:  $z \leftarrow 0 + i \cdot 0$                                   $\triangleright z \in \mathbb{C}$ 
18: for  $m \leftarrow 1, \dots, M_p$  do
19:    $z \leftarrow z + \frac{1}{m} \cdot (G(-m) + G(m))$        $\triangleright$  complex addition!
20:    $\beta \leftarrow \arg z$ 
21:   for  $m \leftarrow 1, \dots, M_p$  do                       $\triangleright$  rotate all coefficients by  $-\beta$ 
22:      $G(-m) \leftarrow e^{-i \cdot \beta} \cdot G(-m)$ 
23:      $G(m) \leftarrow e^{-i \cdot \beta} \cdot G(m)$ 
24:   return  $\beta$ .

```

26.5 TRANSFORMATION-INVARIANT FOURIER DESCRIPTORS

Alg. 26.6

Making Fourier descriptors invariant against scale, shift of start point, and shape rotation. For a given Fourier descriptor G , procedure $\text{MakeStartPointInvariant}(G)$ returns a pair of normalized Fourier descriptors (G^A, G^B) , one for each normalized start point phase φ_A and $\varphi_B = \varphi_A + \pi$.

26.5.4 Other Approaches

The aforementioned normalization for making Fourier descriptors invariant to geometric transformations deviates from the published “classic” techniques in certain ways, but also adopts some common elements. As representative examples, we briefly discuss two of these techniques (already referenced earlier) in the following.

Persoon and Fu [183,184] proposed (in what they call the “suboptimal” approach) to choose the parameters s (common scale factor), β (shape rotation), and φ_s (start point phase) such that the modified coefficients G'_{-1}, G'_{+1} are both imaginary and $|G_{-1} + G_{+1}| = 1$. As argued in [245], this method leaves a $\pm 180^\circ$ ambiguity for the shape orientation. Also, it requires that both G_{-1}, G_{+1} have significant magnitude, which may not be true for G_{-1} in case of shapes that are circularly symmetric (e.g., equilateral triangles, squares, pentagons etc.).

Wallace and Wintz [245] use $|G_{+1}|$ as the common scale factor, because the coefficient G_{+1} typically has the largest magnitude. The phase of G_{+1} , denoted $\phi_1 = \arg G_{+1}$, and the phase of another coefficient G_k ($k > 0$) with the second-largest magnitude and phase $\phi_k = \arg G_k$ are used to compensate for rotation and starting point. Coefficients are phase shifted such that both G'_{+1} and G'_k have zero

Alg. 26.7

Making Fourier descriptors invariant to the shift of start point. Since the result is ambiguous by 180° , two normalized descriptors ($\mathbf{G}^A, \mathbf{G}^B$) are returned, with the start point phase set to φ_A and $\varphi_A + \pi$, respectively.

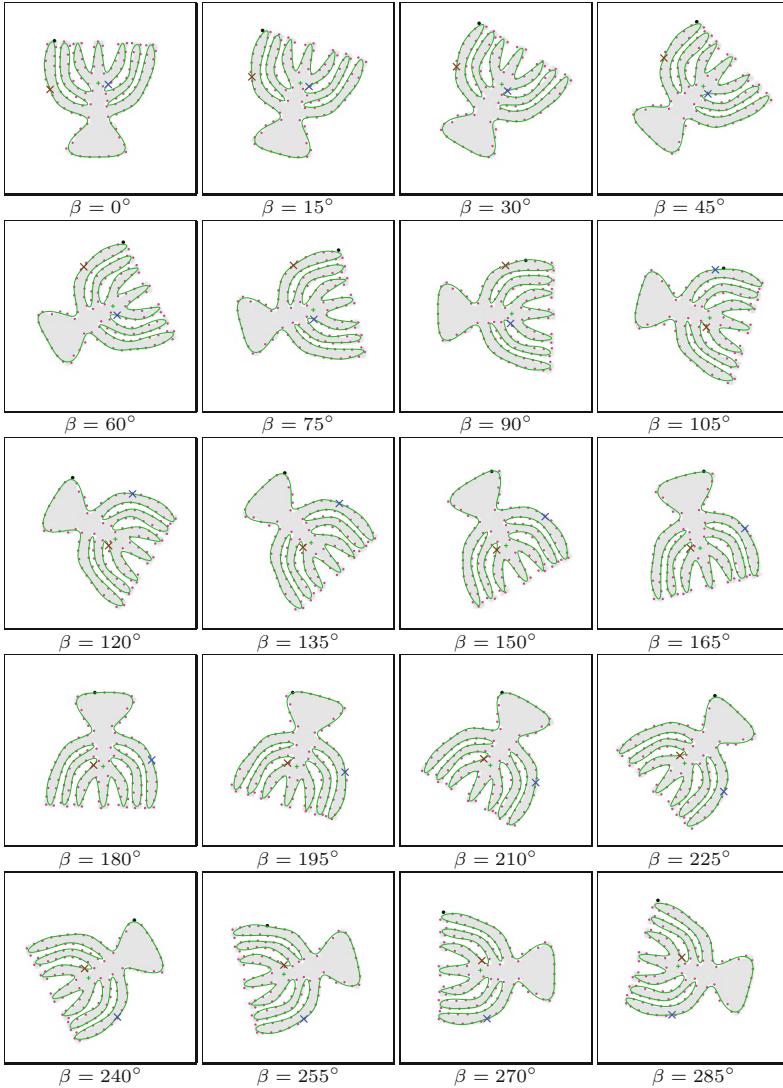
<pre> 1: MakeStartPointInvariant(\mathbf{G}) 2: $\varphi_A \leftarrow \text{GetStartPointPhase}(\mathbf{G})$ \triangleright see below 3: $\mathbf{G}^A \leftarrow \text{ShiftStartPointPhase}(\mathbf{G}, \varphi_A)$ \triangleright see below 4: $\mathbf{G}^B \leftarrow \text{ShiftStartPointPhase}(\mathbf{G}, \varphi_A + \pi)$ 5: return ($\mathbf{G}^A, \mathbf{G}^B$). </pre>	<p>Input: \mathbf{G}, Fourier descriptor with M_p coefficient pairs. Returns a pair of new Fourier descriptors $\mathbf{G}^A, \mathbf{G}^B$, normalized to the start point phase φ_A and $\varphi_A + \pi$, respectively.</p>
<pre> 6: GetStartPointPhase(\mathbf{G}) 7: $c_{\max} \leftarrow -\infty$ 8: $\varphi_{\max} \leftarrow 0$ 9: $K \leftarrow 400$ \triangleright do K search steps over $0, \dots, \pi$ 10: for $k \leftarrow 0, \dots, K-1$ do \triangleright find φ maximizing $f_p(\mathbf{G}, \varphi)$ 11: $\varphi \leftarrow \pi \cdot \frac{k}{K}$ 12: $c \leftarrow f_p(\mathbf{G}, \varphi)$ 13: if $c > c_{\max}$ then 14: $c_{\max} \leftarrow c$ 15: $\varphi_{\max} \leftarrow \varphi$ 16: return φ_{\max}. </pre>	<p>Returns φ maximizing $f_p(\mathbf{G}, \varphi)$, with $\varphi \in [0, \pi]$. The maximum is found by simple brute-force search (for illustration only).</p>
<pre> 17: $f_p(\mathbf{G}, \varphi)$ \triangleright see Eq. 26.103 18: $s \leftarrow 0$ 19: for $m \leftarrow 1, \dots, M_p$ do 20: $z_1 \leftarrow \mathbf{G}(-m) \cdot e^{-i \cdot m \cdot \varphi}$ 21: $z_2 \leftarrow \mathbf{G}(m) \cdot e^{i \cdot m \cdot \varphi}$ 22: $s \leftarrow s + \text{Re}(z_1) \cdot \text{Im}(z_2) - \text{Im}(z_1) \cdot \text{Re}(z_2)$ $\triangleright = s + (z_1 \otimes z_2)$ 23: return s. </pre>	
<pre> 24: ShiftStartPointPhase(\mathbf{G}, φ) \triangleright start-point normalize \mathbf{G} by φ 25: $\mathbf{G}' \leftarrow \text{Duplicate}(\mathbf{G})$ 26: for $m \leftarrow 1, \dots, M_p$ do 27: $\mathbf{G}'(-m) \leftarrow \mathbf{G}(-m) \cdot e^{-i \cdot m \cdot \varphi}$ 28: $\mathbf{G}'(m) \leftarrow \mathbf{G}(m) \cdot e^{i \cdot m \cdot \varphi}$ 29: return \mathbf{G}'. </pre>	

phase. This is accomplished by multiplying all coefficients in the form

$$G'_m = G_m \cdot e^{i \cdot [(m-k) \cdot \phi_1 + (1-m) \cdot \phi_k] \cdot (k-1)}, \quad (26.110)$$

for $-\frac{M}{2} + 1 \leq m \leq \frac{M}{2}$ (also used in [189]). Depending on the index k of the second-largest coefficient, there exist $|k-1|$ different orientation/start point combinations to obtain zero-phase in G'_{+1} and G'_k . If $k=2$, then $|k-1|=1$, thus the solution is unique and Eqn. (26.110) simplifies to

$$G'_m = G_m \cdot e^{i \cdot [(m-2) \cdot \phi_1 + (1-m) \cdot \phi_2]}, \quad (26.111)$$



26.5 TRANSFORMATION-INVARIANT FOURIER DESCRIPTORS

Fig. 26.17

Start point normalization under varying shape rotation (β). The real start point (which varies with shape rotation) is marked by a black dot. The two normalized start points φ_A and $\varphi_B = \varphi_A + \pi$ (calculated with the procedure in Alg. 26.7) are marked by a blue and a brown \times , respectively. Twenty-five Fourier coefficient pairs are used for the normalization and shape reconstruction. Inaccuracies are due to shape variations caused by the use of nearest-neighbor interpolation for the image rotation.

with $\phi_2 = \Im G_2$.²⁰ Otherwise, the ambiguity is resolved by calculating an “ambiguity-resolving” criterion for each of the $|k-1|$ solutions, for example, the amount of “positive real energy”,

$$\sum_{m=1}^{N-1} \operatorname{Re}(G'_m) \cdot |\operatorname{Re}(G'_m)|,$$

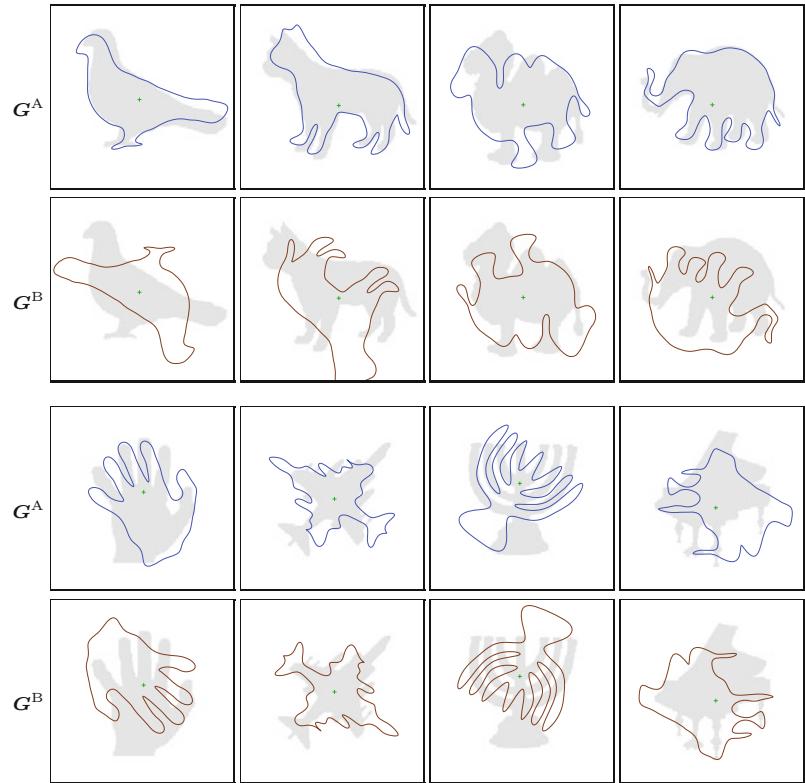
as defined in [245] (other functions were suggested in [189]). This leaves the problem that, for matching, the normalization of the investigated shape descriptor must be based on the same set of dominant coefficients as the reference descriptor. Alternatively, one could memorize the relevant coefficient indexes for every reference descrip-

²⁰ Unfortunately, the general use of coefficient G_2 as a phase reference is critical, because the magnitude of G_2 may be small or even zero for certain symmetrical shapes (including all regular polygons with an even number of faces).

26 FOURIER SHAPE DESCRIPTORS

Fig. 26.18

Reconstruction of various shapes from Fourier descriptors normalized for start point shift and shape rotation. The blue shapes (rows 1, 3) correspond to the normalized Fourier descriptors \mathbf{G}^A with start point phase φ_A . The brown shapes (rows 2, 4) correspond to the normalized Fourier descriptors \mathbf{G}^B with start point phase $\varphi_B = \varphi_A + \pi$. No scale normalization was applied for better visualization.



tor, but then different normalizations must be applied for matching against multiple models in a database.

26.6 Shape Matching with Fourier Descriptors

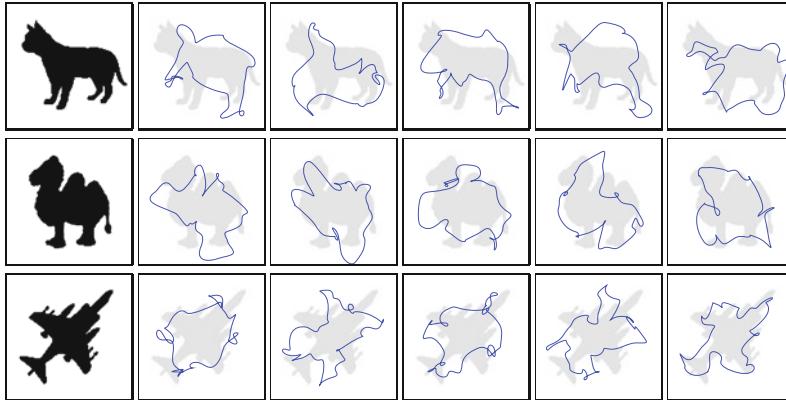
A typical use of Fourier descriptors is to see if a given shape is identical or similar to an exemplar contained in a database of reference shapes. For this purpose, we need to define a distance measure that quantifies the difference between two Fourier shape descriptors \mathbf{G}_1 and \mathbf{G}_2 . In the following, we assume that the Fourier descriptors $\mathbf{G}_1, \mathbf{G}_2$ are at least scale-normalized (as described in Alg. 26.6) and of identical length, each with M_p coefficient pairs.

26.6.1 Magnitude-Only Matching

In the simplest case, we only use the *magnitude* of the Fourier coefficients for comparison and entirely ignore their phase, using the distance function

$$\begin{aligned} \text{dist}_M(\mathbf{G}_1, \mathbf{G}_2) &= \left[\sum_{\substack{m=-M_p, \\ m \neq 0}}^{M_p} (|\mathbf{G}_1(m)| - |\mathbf{G}_2(m)|)^2 \right]^{1/2} \\ &= \left[\sum_{m=1}^{M_p} (|\mathbf{G}_1(-m)| - |\mathbf{G}_2(-m)|)^2 + (|\mathbf{G}_1(m)| - |\mathbf{G}_2(m)|)^2 \right]^{1/2}, \end{aligned} \quad (26.112)$$

where M_p denotes the number of FD pairs used for matching. Note that Eqn. (26.112) is simply the L_2 norm of the magnitude difference vector, and of course other norms (such as L_1 or L_∞) could be used as well. The advantage of the magnitude-only approach is that no normalization (except for scale) is required. Its drawback is that even highly dissimilar shapes might be mistakenly matched, since the removal of phase naturally eliminates shape information that is possibly essential for discrimination. As demonstrated in Fig. 26.19, a given Fourier magnitude vector may correspond to a great diversity of shapes, and thus the subspace of “equivalent” shapes defined by the magnitude-only distance dist_M is quite large.



26.6 SHAPE MATCHING WITH FOURIER DESCRIPTORS

Fig. 26.19
Magnitude-only reconstruction (randomized phase). Reconstruction of shapes from Fourier descriptors with the phase of all coefficients (except G_{-1} , G_0 , and G_{+1}) individually randomized. Note that the magnitude of the coefficients is exactly the same for each shape category, so all blue shapes would be considered “equivalent” to the original shape (first column) by a magnitude-only matcher.

Nevertheless, magnitude-only matching may be sufficient in situations where the reference shapes are not too similar. In a sense, the operation of reducing the complex-valued Fourier descriptors to their magnitude vectors can be viewed as a *hash* function. While potentially many different shapes may produce (i.e., “hash to”) similar Fourier magnitude vectors, the chance of two real shapes mapping to the same vector (and thus being confused) may be relatively small. Thus, particularly considering its simplicity (only scale-normalization of descriptors is required), magnitude-based matching can be quite effective in practice.

Figure 26.20 shows the pair-wise magnitude-only distances (blue cells, values are $10 \times \text{dist}_M$) between various sample shapes. The corresponding intra-class distances, given in Fig. 26.21, are typically more than one order of magnitude smaller, indicating that shape discrimination based on this measure should be fairly reliable.

26.6.2 Complex (Phase-Preserving) Matching

Assuming that the Fourier descriptors \mathbf{G}_1 and \mathbf{G}_2 have been normalized for scale, start point shift, and shape rotation (see Alg. 26.6), we can use the following function to measure their mutual distance:

26 FOURIER SHAPE DESCRIPTORS

Fig. 26.20

Inter-class Fourier descriptor distances (magnitude-only and complex-valued). Numbers inside the green fields (lower-left half of the matrix) are the magnitude-only distances dist_M (see Eqn. (26.112)). Numbers in blue fields (upper-right half of the matrix) are the complex-valued distances dist_C (see Eqn. (26.114)). Shapes were sampled uniformly at 125 contour positions, with 25 coefficient pairs. Fourier descriptors were normalized for scale, start point and rotation. All distance values are multiplied by 10.

	bird	cat	camel	elephant	hand	harrier	menora	piano	creature
	0.000	4.529	4.482	5.007	5.525	4.314	7.554	5.174	7.076
	3.156	0.000	5.788	4.708	5.711	5.701	7.181	5.543	7.677
	2.648	3.005	0.000	4.429	5.573	3.726	7.014	4.013	8.480
	3.487	1.933	2.549	0.000	6.100	4.618	5.338	4.369	8.743
	4.627	3.146	3.132	2.372	0.000	6.079	8.540	5.580	7.136
	3.712	3.707	2.687	3.553	4.294	0.000	6.818	4.958	8.284
	5.835	4.893	4.563	4.162	3.788	5.775	0.000	6.826	11.072
	4.037	2.426	2.610	1.876	1.848	3.405	4.315	0.000	7.666
	6.030	6.261	5.554	5.492	5.955	5.914	5.190	6.049	0.000

$\text{dist}_M(\mathbf{G}_1, \mathbf{G}_2)$

$\text{dist}_C(\mathbf{G}_1, \mathbf{G}_2)$

$$\text{dist}_C(\mathbf{G}_1, \mathbf{G}_2) = \left(\sum_{\substack{m=-M_p, \\ m \neq 0}}^{M_p} |\mathbf{G}_1(m) - \mathbf{G}_2(m)|^2 \right)^{1/2} \quad (26.113)$$

$$= \left(\sum_{m=1}^{M_p} |\mathbf{G}_1(-m) - \mathbf{G}_2(-m)|^2 + |\mathbf{G}_1(m) - \mathbf{G}_2(m)|^2 \right)^{1/2} \quad (26.114)$$

$$= \left(\sum_{\substack{m=-M_p, \\ m \neq 0}}^{M_p} [\text{Re}(\mathbf{G}_1(m)) - \text{Re}(\mathbf{G}_2(m))]^2 + [\text{Im}(\mathbf{G}_1(m)) - \text{Im}(\mathbf{G}_2(m))]^2 \right)^{1/2}. \quad (26.115)$$

Again, this is simply the L_2 norm of the complex-valued difference vector $\mathbf{G}_1 - \mathbf{G}_2$ (ignoring the coefficients at $m = 0$), which could be substituted by some other norm. Since the phase of the involved coefficients is fully preserved, a zero distance between two Fourier descriptors means that they represent the very same shape. Thus the set of equivalent shapes defined by the distance function in Eqn. (26.114) is much smaller than the one defined by the magnitude-only distance in Eqn. (26.112). Consequently, the probability of two different shapes being confused for the same is also significantly smaller with this distance measure.

$\alpha =$	0°	17°	34°	51°	68°	85°	102°	119°	136°	153°	170°	187°	204°
dist _M	0.000	0.070	0.126	0.151	0.103	0.058	0.143	0.107	0.195	0.190	0.105	0.078	0.053
dist _C	0.000	0.141	0.222	0.299	0.198	0.111	0.274	0.159	0.313	0.400	0.142	0.162	0.092
dist _M	0.000	0.134	0.144	0.176	0.167	0.055	0.104	0.206	0.227	0.135	0.164	0.083	0.174
dist _C	0.000	0.222	0.214	0.252	0.244	0.081	0.141	0.310	0.339	0.197	0.231	0.157	0.281
dist _M	0.000	0.117	0.346	0.147	0.142	0.141	0.109	0.100	0.125	0.163	0.099	0.147	0.106
dist _C	0.000	0.229	0.728	0.367	0.310	0.386	0.161	0.186	0.202	0.252	0.141	0.191	0.271
dist _M	0.000	0.121	0.195	0.272	0.170	0.057	0.135	0.175	0.216	0.176	0.092	0.112	0.160
dist _C	0.000	0.180	0.317	0.392	0.278	0.080	0.218	0.257	0.307	0.266	0.160	0.198	0.248
dist _M	0.000	0.127	0.138	0.179	0.130	0.048	0.131	0.115	0.329	0.173	0.202	0.109	0.132
dist _C	0.000	0.179	0.186	0.361	0.180	0.085	0.234	0.188	0.496	0.263	0.313	0.182	0.195
dist _M	0.000	0.234	0.171	0.224	0.095	0.090	0.106	0.189	0.228	0.170	0.079	0.121	0.213
dist _C	0.000	0.433	0.290	0.317	0.147	0.129	0.197	0.276	0.344	0.251	0.146	0.197	0.308
dist _M	0.000	0.163	0.148	0.131	0.213	0.116	0.228	0.322	0.334	0.205	0.253	0.108	0.122
dist _C	0.000	0.570	0.330	0.395	0.456	0.169	0.271	0.401	0.465	0.295	0.440	0.149	0.251
dist _M	0.000	0.164	0.186	0.161	0.186	0.101	0.112	0.252	0.159	0.150	0.169	0.104	0.201
dist _C	0.000	0.264	0.362	0.311	0.255	0.175	0.148	0.576	0.230	0.267	0.232	0.142	0.284
dist _M	0.000	0.154	0.190	0.167	0.103	0.084	0.180	0.390	0.210	0.123	0.194	0.084	0.131
dist _C	0.000	0.203	0.260	0.248	0.141	0.108	0.232	0.447	0.308	0.171	0.234	0.120	0.160

26.6 SHAPE MATCHING WITH FOURIER DESCRIPTORS

Fig. 26.21

Intra-class Fourier descriptor distances (magnitude-only and complex-valued). The reference images (0° column) were rotated by angle α (multiples of 17°), using no (i.e., nearest-neighbor) interpolation. Numbers inside the blue fields are the magnitude-only distances dist_M (see Eqn. (26.112)). Numbers inside the green fields are the complex-valued distances dist_C (see Eqn. (26.114)). Shapes were sampled uniformly at 125 contour positions, with 25 coefficient pairs. Fourier descriptors were normalized for scale, start point shift and shape rotation. All distance values are multiplied by 10. Note that all intra-class distances are roughly one order of magnitude smaller than the inter-class distances shown in Fig. 26.20.

Complex inter-class and intra-class distance values for the set of sample shapes are listed in Figs. 26.20 and 26.21. Notice that, with the normalization described in Alg. 26.6, the complex intra-class distance values in Fig. 26.21 (which should be as small as possible) are typically about twice as large as the corresponding magnitude-only distance values, but still an order of magnitude smaller than comparable inter-class values in Fig. 26.20, so reliable shape discrimination should be possible.

The price paid for the increased discriminative power is the extra work necessary for normalizing the Fourier descriptors for start point and shape rotation (in addition to scale), as described in Alg. 26.6. Note that this involves the comparison with *two* normalized descriptors to cope with the unresolved 180° ambiguity of the start point normalization (see Eqns. (26.104) and (26.105)). For example, assume we wish to compare two shapes V_1, V_2 with Fourier descriptors $\mathbf{G}_1, \mathbf{G}_2$, respectively. We first calculate the corresponding invariant descriptors (as described in Alg. 26.6),

$$\begin{aligned} (\mathbf{G}_1^A, \mathbf{G}_1^B) &\leftarrow \text{MakeInvariant}(\mathbf{G}_1), \\ (\mathbf{G}_2^A, \mathbf{G}_2^B) &\leftarrow \text{MakeInvariant}(\mathbf{G}_2). \end{aligned} \quad (26.116)$$

Now we use Eqn. (26.114) to calculate the complex-valued distance as

$$d_{\min} = \min(\text{dist}_C(\mathbf{G}_1^A, \mathbf{G}_2^A), \text{dist}_C(\mathbf{G}_1^A, \mathbf{G}_2^B)) \quad (26.117)$$

or, alternatively, as

$$d_{\min} = \min(\text{dist}_C(\mathbf{G}_1^A, \mathbf{G}_2^A), \text{dist}_C(\mathbf{G}_1^B, \mathbf{G}_2^A)). \quad (26.118)$$

Note that, in any case, the resulting distance d_{\min} will be small only if the two shapes V_1, V_2 are really similar. This also means that we only need to store *one* of the two normalized Fourier descriptors—for example, $\mathbf{G}_{\text{ref}}^A$ —for each reference shape V_{ref} and then (following Eqn. (26.117)) compare it to *both* normalized descriptors $\mathbf{G}_{\text{new}}^A$ and $\mathbf{G}_{\text{new}}^B$ of any new shape V_{new} .²¹

To illustrate this idea, Alg. 26.8 shows the construction of a simple Fourier descriptor database from a set of reference shapes and its subsequent use for classifying unknown shapes. First, procedure `MakeFdDataBase(V)` returns a map D holding a normalized Fourier descriptor for each of the reference shapes given in V . Matching a new shape V_{new} to the entries in the database D is accomplished by procedure `FindBestMatch(V_{\text{new}}, D, d_{\max})`, which returns the index of the best-fitting shape in D , or nil if the distance of the closest match exceeds the predefined threshold d_{\max} . As common in this situation, we use *squared* distance values (i.e., dist_C^2) for matching in Alg. 26.8 (lines 15–18), thereby avoiding the square root operations in Eqns. (26.112) and (26.114).

26.7 Java Implementation

The algorithms described in this chapter have been implemented as part of the open `imagingbook` library,²² which is available at the book’s accompanying website. As usual, most Java methods are named and structured identically to the procedures defined in the various algorithms for easy identification.

`FourierDescriptor` (class)

This is the main class of this package; it holds all data structures and implements the functionality common to all Fourier descriptors, including methods for shape reconstruction, invariance, and matching, as will be described here.

²¹ The justification for keeping only *one* of the two normalized descriptors $\mathbf{G}_{\text{ref}}^A, \mathbf{G}_{\text{ref}}^B$ of each reference shape V_{ref} is that if two candidate shapes V_1, V_2 are similar, then the normalization will produce pairs of Fourier descriptors $(\mathbf{G}_1^A, \mathbf{G}_1^B)$ and $(\mathbf{G}_2^A, \mathbf{G}_2^B)$ that are also similar but not necessarily in the same order. Therefore \mathbf{G}_1^A must only match with *either* \mathbf{G}_2^A *or* \mathbf{G}_2^B to detect the similarity of V_1 and V_2 .

²² Package `imagingbook.pub.fd`.

```

1: MakeFdDataBase( $V_{\text{ref}}, M'$ )
   Input:  $V_{\text{ref}} = (V_0, V_1, \dots, V_{N_R})$ , a sequence of reference shapes;
           $M'$ , the number of Fourier coefficients. Returns a sequence of
          model Fourier descriptors for the reference shapes in  $V_{\text{ref}}$ .
2:  $N_R \leftarrow |\mathcal{V}_{\text{ref}}|$ 
3:  $R \leftarrow$  new map of Fourier descriptors over  $[0, N_R - 1]$ 
4: for  $i \leftarrow 0, \dots, N_R - 1$  do
5:    $\mathbf{G} \leftarrow \text{FourierDescriptorUniform}(V_{\text{ref}}(i), M')$             $\triangleright$  Alg. 26.3
6:    $(\mathbf{G}^A, \mathbf{G}^B) \leftarrow \text{MakelnInvariant}(\mathbf{G})$             $\triangleright$  Alg. 26.6
7:    $R(i) \leftarrow \mathbf{G}^A$             $\triangleright$  store only one normalized descriptor ( $\mathbf{G}^A$ )
8: return R.

9: FindBestMatch( $V_{\text{new}}, M', R, d_{\text{max}}$ )
   Input:  $V_{\text{new}}$ , a new shape;  $M'$ , the number of Fourier coefficients;
          R, a sequence of reference Fourier descriptors;  $d_{\text{max}}$ , maximum
          squared distance acceptable for a positive match. Returns the
          best-matching shape index  $i_{\text{min}}$  or nil if no acceptable match was
          found.
10:   $\mathbf{G}_{\text{new}} \leftarrow \text{FourierDescriptorUniform}(V_{\text{new}}, M')$             $\triangleright$  Alg. 26.3
11:   $(\mathbf{G}_{\text{new}}^A, \mathbf{G}_{\text{new}}^B) \leftarrow \text{MakelnInvariant}(\mathbf{G}_{\text{new}})$             $\triangleright$  Alg. 26.6
12:   $d_{\text{min}} \leftarrow \infty, i_{\text{min}} \leftarrow -1$ 
13:  for  $i \leftarrow 0, \dots, |R| - 1$  do
14:     $\mathbf{G}_{\text{ref}}^A \leftarrow R(i)$ 
15:     $d_2 \leftarrow \min(D2(\mathbf{G}_{\text{new}}^A, \mathbf{G}_{\text{ref}}^A), D2(\mathbf{G}_{\text{new}}^B, \mathbf{G}_{\text{ref}}^A))$             $\triangleright$  Eq. 26.118
16:    if  $d_2 < d_{\text{min}}$  then
17:       $d_{\text{min}} \leftarrow d_2$ 
18:       $i_{\text{min}} \leftarrow i$ 
19:    if  $d_{\text{min}} \leq d_{\text{max}}$  then
20:      return  $i_{\text{min}}$             $\triangleright$  best match index is  $i_{\text{min}}$ 
21:    else
22:      return nil.            $\triangleright$  no matching shape found in R

23: D2( $\mathbf{G}_1, \mathbf{G}_2$ )
   Returns the squared complex distance  $\text{dist}_C^2(\mathbf{G}_1, \mathbf{G}_2)$  between the
   Fourier descriptors  $\mathbf{G}_1, \mathbf{G}_2$  (see Eq. 26.114).
24:  $d \leftarrow 0, M_p \leftarrow (\min(|\mathbf{G}_1|, |\mathbf{G}_2|) - 1) \div 2$ 
25: for  $m \leftarrow -M_p, \dots, M_p, m \neq 0$  do
26:    $d \leftarrow d + [\text{Re}(\mathbf{G}_1(m)) - \text{Re}(\mathbf{G}_2(m))]^2 +$ 
       $[\text{Im}(\mathbf{G}_1(m)) - \text{Im}(\mathbf{G}_2(m))]^2$ 
27: return d.            $\triangleright d \equiv (\text{dist}_C(\mathbf{G}_1, \mathbf{G}_2))^2$ 

```

Class `FourierDescriptor` is abstract and thus cannot be instantiated. To create Fourier descriptor objects, one of the concrete subclasses `FourierDescriptorUniform` or `FourierDescriptorFromPolygon` (discussed later in this section) may be used, which provide the appropriate constructors. `FourierDescriptor` provides the following methods for both types of Fourier descriptors.

Access to Fourier coefficients

```

Complex[] getCoefficients ()
   Returns the complete vector of complex-valued Fourier coeffi-
   cients.23

```

²³ The class `Complex` is defined in package `imagingbook.lib.math`.

26.7 JAVA IMPLEMENTATION

Alg. 26.8

Simple shape matching with a database of Fourier descriptors. `MakeFd DataBase(Vref, M')` creates and returns a new database (map) R from a sequence of reference shapes V_{ref}. R can then be passed to `FindBestMatch(Vnew, M', R, dmax)` for classifying a new shape V_{new}, where d_{max} is a predefined distance threshold.

```

Complex getCoefficient (int m)
    Returns the value of the Fourier coefficient  $\mathbf{G}(m \bmod M)$ , with
     $M = |\mathbf{G}|$  as above.

Complex setCoefficient (int m, Complex z)
    Replaces the Fourier coefficient  $\mathbf{G}(m \bmod M)$  by the complex
    value  $z$ , with  $M = |\mathbf{G}|$  as above.

Complex setCoefficient (int m, double a, double b)
    Replaces the Fourier coefficient  $\mathbf{G}(m \bmod M)$  by the complex
    value  $z = a + i \cdot b$ , with  $M = |\mathbf{G}|$  as above.

int size ()
    Returns the length ( $M$ ) of the Fourier descriptor.

int getMaxNegHarmonic ()
    Returns the max. negative harmonic  $m = -(M - 1) \div 2$  for
    this Fourier descriptor (of length  $M$ ).

int getMaxPosHarmonic ()
    Returns the max. positive harmonic  $m = M \div 2$  for this Fourier
    descriptor (of length  $M$ ).

int getMaxCoefficientPairs ()
    Returns the maximum number of coefficient pairs,  $(M - 1) \div 2$ ,
    for this Fourier descriptor (of length  $M$ ).

void truncate (int Mp)
    Truncates this Fourier descriptor to the  $M_p$  lowest-frequency
    coefficients (see Eqn. (26.23)).

```

Comparing Fourier descriptors

```

double distanceComplex (FourierDescriptor fd2)
    Returns the complex-valued distance ( $\text{dist}_C(\mathbf{G}_1, \mathbf{G}_2)$ , see Eqn.
    (26.114)) between this Fourier descriptor ( $\mathbf{G}_1$ ) and another
    Fourier descriptor  $\text{fd2}$  ( $\mathbf{G}_2$ ). The zero-coefficients are ignored.

double distanceComplex (FourierDescriptor fd2, int Mp)
    As above, but using only  $M_p$  coefficient pairs (see Eqn.
    (26.114)).

double distanceMagnitude (FourierDescriptor fd2)
    Returns the magnitude-only distance ( $\text{dist}_M(\mathbf{G}_1, \mathbf{G}_2)$ , see Eqn.
    (26.112)) between this Fourier descriptor ( $\mathbf{G}_1$ ) and another
    Fourier descriptor  $\text{fd2}$  ( $\mathbf{G}_2$ ). The zero-coefficients are ignored.

double distanceMagnitude (FourierDescriptor fd2,
int Mp)
    As above, but using only  $M_p$  coefficient pairs (see Eqn.
    (26.112)).

```

Shape reconstruction

```

Complex[] getReconstruction (int N)
    Returns the shape reconstructed from the complete Fourier de-
    scriptor as a sequence of  $N$  complex-valued contour points. The
    contour points are obtained by evaluating  $\text{getReconstruct-}
    \text{ionPoint}(t)$  at uniformly spaced positions  $t \in [0, 1]$ .

Complex[] getReconstruction (int N, int Mp)
    Returns a partial shape reconstruction from  $M_p$  Fourier coeffi-
    cient pairs as a sequence of  $N$  complex-valued contour points.

```

Complex getReconstructionPoint (double t)

Returns a single point (as a complex value) on the continuous contour for path parameter $t \in [0, 1]$, reconstructed from the complete Fourier descriptor (see Eqn. (26.20)).

Complex getReconstructionPoint (double t, int Mp)

Returns a single point (as a complex value) on the continuous contour for path parameter $t \in [0, 1]$, reconstructed from Mp Fourier coefficient pairs.

Normalization

FourierDescriptor[] makeInvariant ()

Returns a pair of Fourier descriptors ($\mathbf{G}^A, \mathbf{G}^B$) that are normalized for scale, start point shift and shape rotation (see Alg. 26.6).

double makeRotationInvariant ()

Normalizes the Fourier descriptor for shape rotation by phase-shifting all coefficients (see Alg. 26.6). Returns the estimated rotation angle β .

double makeScaleInvariant ()

Normalizes the Fourier descriptor for scale by multiplying with a common factor, such that the L_2 norm of the resulting vector is 1. Returns the scale factor that was applied for normalization.

FourierDescriptor[] makeStartPointInvariant ()

Returns a pair of normalized Fourier descriptors ($\mathbf{G}^A, \mathbf{G}^B$), one for each start point normalization angles φ_A and $\varphi_B = \varphi_A + \pi$, respectively (see Alg. 26.7).

void makeTranslationInvariant ()

Modifies this Fourier descriptor by setting the coefficient $\mathbf{G}(0)$ to zero. This method is rarely needed because $\mathbf{G}(0)$ is ignored for matching.

FourierDescriptorUniform (class)

This sub-class of **FourierDescriptor** represents Fourier descriptors obtained from uniformly sampled contours, as described in Alg. 26.2. It provides the constructor methods

FourierDescriptorUniform (Point2D[] V),

FourierDescriptorUniform (Point2D[] V, int Mp),

where V is a sequence of M contour points (**Point2D**), assumed to be uniformly sampled. The first constructor creates a full Fourier descriptor with M coefficients (see Alg. 26.2). The second constructor creates a Fourier descriptor with Mp coefficient pairs (i.e., $2 \cdot Mp + 1$ coefficients), as described in Alg. 26.3

FourierDescriptorFromPolygon (class)

This sub-class of **FourierDescriptor** represents Fourier descriptors obtained directly from polygons (without contour sampling, see Alg. 26.5). It provides the single constructor method

FourierDescriptorFromPolygon (Point2D[] V, int Mp),

where V is a sequence of polygon vertices and M_p specifies the number of Fourier coefficient pairs.

PolygonSampler (class)

Instances of this utility class can be used to produce uniformly sampled polygons.

```
Point2D[] samplePolygonUniformly(Point2D[] V, int M)
```

Samples the closed polygon path specified by the vertices in V at M equi-distant positions and returns the resulting point sequence (see Alg. 26.1).

Example

The code example in Prog. 26.1 demonstrates the use of the Fourier descriptor API. It assumes that the binary input image (ip) contains at least one connected foreground region. Region labeling and contour extraction is applied first, using methods provided by the `imagingbook.regions` and `imagingbook.contours` packages.²⁴ Subsequently, the longest region contour (C) is used to create a Fourier descriptor (fd) with $M_p = 15$ coefficient pairs. A partial reconstruction is calculated from the original Fourier descriptor with 100 sample points along the contour. The last lines show how a pair of invariant descriptors (G^A, G^B) is obtained by applying the `makeInvariant()` method. Note that the code fragment in Prog. 26.1 is not complete but would typically be part of the `run()` method in an ImageJ plugin. The full version and additional code examples can be found on the book's website.

26.8 Discussion and Further Reading

The use of Fourier descriptors for shape description and matching dates back to the early 1960's [55, 81], advanced by the work of Zahn and Roskies [262], Granlund [93], Richard and Hemami [196], and Persoon and Fu [183, 184] in the 1970s, particularly in the context of character recognition and aircraft identification. Making Fourier descriptors invariant against various geometric transformations was a key issue from the very beginning, and several relevant contributions were published in the 1980s, including [245], [57] [143], and [189]. Unfortunately, as illustrated in this chapter, to achieve robust invariance and uniqueness of representation in practice is not as easy as sometimes suggested in the literature, despite the simplicity and elegance of the underlying theory. In practice, normalization for descriptor invariance is quite difficult for arbitrary shapes because of possibly vanishing Fourier coefficients and the resulting sensitivity to noise.

Fourier descriptors have nevertheless become popular in a wide range of applications, including geology and, in particular, biological imaging, as documented by the work of Lestrel and others in [146].

²⁴ See also Chapter 10.

```

1 ...
2 import imagingbook.lib.math.Complex;
3 import imagingbook.pub.fd.*;
4 import imagingbook.pub.regions.*;
5
6 ByteProcessor ip ...; // assumed to contain a binary image
7
8 // segment ip and select the longest outer region contour:
9 RegionContourLabeling labeling =
10     new RegionContourLabeling(ip);
11 List<Contour> outerContours =
12     labeling.getAllOuterContours(true);
13 Contour contr = outerContours.get(0); // get the longest contour
14 Point2D[] V = contr.getPointArray();
15
16 // create the Fourier descriptor for V with 15 coefficient pairs:
17 FourierDescriptor fd = new FourierDescriptorUniform(V, 15);
18
19 // reconstruct the corresponding shape with 100 contour points:
20 Complex[] R = fd.getReconstruction(100);
21
22 // create a pair of invariant descriptors ( $G^A, G^B$ ):
23 FourierDescriptor[] fdAB = fd.makeInvariant();
24 FourierDescriptor fdA = fdAB[0]; // =  $G^A$ 
25 FourierDescriptor fdb = fdAB[1]; // =  $G^B$ 
26 ...

```

26.9 EXERCISES

Prog. 26.1

Fourier descriptor code example. The input image `ip` is assumed to contain a binary image (line 6). The class `RegionContourLabeling` is used to find connected regions (line 10). Then the list of outer contours is retrieved (line 12) and the longest contour is assigned to `V` as an array of type `Point2D` (lines 13–14). In line 17, the contour `V` is used to create a Fourier descriptor with 15 coefficient pairs. Alternatively, we could have created a Fourier descriptor of the same length (number of coefficients) as the contour and then truncated it (using the `truncate()` method) to the specified number of coefficient pairs. A partial reconstruction of the contour (with 100 sample points) is calculated from the Fourier descriptor `fd` in line 20. Finally, a pair of invariant descriptors (contained in the array `fdAB`) is calculated in line 23.

Fourier descriptors have been extended to accommodate affine transformations and applied to 3D object identification [5] and stereo matching [257].

Although Fourier descriptors have been investigated to handle open contours and partial shapes [148], they are naturally best suited to dealing with closed contours, as we have described. Of course, this is a limitation if shapes are only partially visible or occluded. The presentation in this chapter was limited to what are frequently called “elliptical” Fourier descriptors [93], since they are most popular and well known. Other types of Fourier descriptors have been proposed, which are not covered here but can be found elsewhere in the literature (see, e.g., [126, p. 534] and [174, Ch. 7]).

26.9 Exercises

Exercise 26.1. Verify that the DFT spectrum is periodic, that is, that $\mathbf{G}(-m) = \mathbf{G}(M-m)$ holds for arbitrary $m \in \mathbb{Z}$ (as claimed in Eqn. (26.22)).

Exercise 26.2. Algorithm 26.9 shows an alternative solution to uniform polygon sampling. Implement this algorithm and verify that it is equivalent to Alg. 26.1 (implemented as method `samplePolygonUniformly()` in class `PolygonSampler`, see Sec. 26.7).

Exercise 26.3. Assume that the complete outer contour of a binary region is given as a sequence of P boundary pixels with coordinates

26 FOURIER SHAPE DESCRIPTORS

Alg. 26.9

Uniform sampling of a polygon path (alternative to Alg. 26.1, proposed by J. Heinzelreiter).

1: **SamplePolygonUniformly**(V, M)

Input: $V = (\mathbf{v}_0, \dots, \mathbf{v}_{N-1})$, a sequence of N points representing the vertices of a closed 2D polygon; M , number of desired sample points. Returns a new sequence $\mathbf{g} = (g_0, \dots, g_{M-1})$ of complex values representing sample points sampled uniformly along the path of the input polygon V .

```

2:    $N \leftarrow |V|$ 
3:    $\Delta \leftarrow \frac{1}{M} \cdot \text{PathLength}(V)$        $\triangleright$  segment length  $\Delta$ , see Alg. 26.1
4:   Create map  $\mathbf{g}: [0, M-1] \rightarrow \mathbb{C}$        $\triangleright$  complex point sequence  $\mathbf{g}$ 
5:    $\mathbf{g}(0) \leftarrow \text{Complex}(V(0))$ 
6:    $i \leftarrow 0$                                  $\triangleright$  index of path segment  $\langle V_i, V_{i+1} \rangle$ 
7:    $k \leftarrow 1$                                  $\triangleright$  index of first unassigned point in  $\mathbf{g}$ 
8:    $d_p \leftarrow 0$                                  $\triangleright$  path distance between  $V(i)$  and  $V(k-1)$ 
9:   while ( $i < N$ )  $\wedge$  ( $k < M$ ) do
10:     $\mathbf{v}_A \leftarrow V(i)$ 
11:     $\mathbf{v}_B \leftarrow V((i+1) \bmod N)$ 
12:     $\delta \leftarrow \|\mathbf{v}_B - \mathbf{v}_A\|$                    $\triangleright$  Euclidean distance
13:    if ( $\Delta - d_p \leq \delta$ ) then
14:       $\mathbf{x} \leftarrow \mathbf{v}_A + \frac{\Delta - d_p}{\delta} \cdot (\mathbf{v}_B - \mathbf{v}_A)$      $\triangleright x_k$  by lin. interpolation
15:       $\mathbf{g}(k) \leftarrow \text{Complex}(\mathbf{x})$ 
16:       $d_p \leftarrow d_p - \Delta$ 
17:       $k \leftarrow k + 1$ 
18:    else
19:       $d_p \leftarrow d_p + \delta$ 
20:       $i \leftarrow i + 1$ 
21:   return  $\mathbf{g}$ .

```

$V = (\mathbf{p}_0, \dots, \mathbf{p}_{P-1})$. To produce a Fourier descriptor of length $M < P$ there are several options:

1. Sample the original contour V at M uniformly-spaced positions (see Alg. 26.1) and then calculate the Fourier descriptor of length M using Alg. 26.2.
2. Calculate a partial Fourier descriptor of length M' from the original contour V using Alg. 26.3.
3. Calculate the full Fourier descriptor (of length M) from the original contour V (using Alg. 26.2) and subsequently truncate²⁵ the Fourier descriptor to length M' , as described in Eqns. (26.23) and (26.24).
4. Treat the original boundary coordinates V as the vertices of a closed polygon and calculate a Fourier descriptor with $M_P = M \div 2$ coefficient pairs, using the trigonometric method described in Alg. 26.5.

Compare these approaches and discuss their individual merits or disadvantages in terms of efficiency and accuracy.

Exercise 26.4. Test the Fourier descriptor normalization described in Algs. 26.6 and 26.7 (implemented by method `makeInvariant()` in the Java API) for changes in scale, start point shift, and shape rotation on a suitable set of binary shapes (e.g., images from the

²⁵ See method `truncate(int Mp)` in Sec. 26.7.

KIMIA dataset [134]). See the examples for shape rotation and (implicit) start point shifts in Fig. 26.21. How reliably do the normalized Fourier descriptors of the modified shapes match to their corresponding originals?

26.9 EXERCISES

Exercise 26.5. Magnitude-only matching (see Sec. 26.6.1) is much simpler than complex-valued matching (see Sec. 26.6.2) of Fourier descriptors, since no normalization for phase (start point shift and shape rotation) is required. However, it can be assumed that different shapes are more likely to be confused if the phase information is ignored. Test this hypothesis on a large number and variety of different shapes. Compare the confusion probability for magnitude-only vs. complex-valued matching.

Appendix A

Mathematical Symbols and Notation

A.1 Symbols

The following symbols are used in the main text primarily with the denotations given here. While some symbols may be used for purposes other than the ones listed, the meaning should always be clear in the particular context.

(a_0, \dots, a_{n-1}) A *vector* or *list*, that is, an ordered sequence of n elements of the same type. Unlike a *set* (see below), a list may contain the same element more than once. If used to denote a *vector*, then (a_0, \dots, a_{n-1}) is usually a *row vector* and $(a_0, \dots, a_{n-1})^\top$ is the corresponding (transposed) *column vector*.¹ If used to represent a *list*,² () represents the *empty list* and (a) is a list with a single element a . $|A|$ is the *length* of the sequence A , that is, the number of contained elements. $A \cup B$ denotes the concatenation of A, B . $A(i)$ or a_i refers to the i -th element of A . $A(i) \leftarrow x$ means that the i -th element of A is set to (i.e., replaced by) the quantity x .

$\{a, b, c, d, \dots\}$ A *set*, that is, an unordered collection of distinct elements. A particular element x can be contained in a set at most once. {} denotes the empty set. $|\mathcal{A}|$ is the size (cardinality) of the set \mathcal{A} . $\mathcal{A} \cup \mathcal{B}$ is the union and $\mathcal{A} \cap \mathcal{B}$ is the intersection of two sets \mathcal{A}, \mathcal{B} . $x \in \mathcal{A}$ means that the element x is contained in \mathcal{A} .

$\langle A, B, C \rangle$ A *tuple*, that is, a fixed-size, ordered sequence of elements, each possibly of a different type.³

¹ In most programming environments, vectors are implemented as one-dimensional arrays, with elements being referred to by position (index).

² Lists are usually implemented with dynamic data structures, such as linked lists. Java's *Collections* framework provides numerous easy-to-use list implementations.

³ Tuples are typically implemented as *objects* (in Java or C++) or *structures* (in C) with elements being referred to by name.

Appendix A MATHEMATICAL SYMBOLS AND NOTATION	
$[a, b]$	Numeric interval; $x \in [a, b]$ means $a \leq x \leq b$. Similarly, $x \in [a, b)$ says that $a \leq x < b$.
$ A $	Length (number of elements) of a sequence (see above) or size (cardinality) of a set A , that is, $ A \equiv \text{card } A$.
$ \mathbf{A} $	Determinant of a matrix \mathbf{A} ($ \mathbf{A} \equiv \det(\mathbf{A})$).
$ x $	Absolute value (magnitude) of a scalar or complex quantity x .
$\ \mathbf{x}\ $	Euclidean (L_2) norm of a vector \mathbf{x} . $\ \mathbf{x}\ _n$ denotes the magnitude of \mathbf{x} using a particular norm L_n .
$\lceil x \rceil$	“Ceil” of x , the smallest integer $z \in \mathbb{Z}$ greater than $x \in \mathbb{R}$. For example, $\lceil 3.141 \rceil = 4$, $\lceil -1.2 \rceil = -1$.
$\lfloor x \rfloor$	“Floor” of x , the largest integer $z \in \mathbb{Z}$ smaller than $x \in \mathbb{R}$. For example, $\lfloor 3.141 \rfloor = 3$, $\lfloor -1.2 \rfloor = -2$.
\div	Integer division operator: $a \div b$ denotes the quotient of the two integers a, b . For example, $5 \div 3 = 1$ and $-13 \div 4 = -3$ (equivalent to Java’s “ $/$ ” operator in the case of integer operands).
$*$	Linear convolution operator (see Sec. 5.3.1).
\circledast	Linear correlation operator (see Sec. 23.1.1).
\otimes	Outer vector product (see Sec. B.3.2).
\times	Cross product (between vectors or complex quantities (see Sec. B.3.3).
\oplus	Morphological dilation operator (see Sec. 9.2.3).
\ominus	Morphological erosion operator (see Sec. 9.2.4).
\circ	Morphological opening operator (see Sec. 9.3.1).
\bullet	Morphological closing operator (see Sec. 9.3.2).
\smile	Concatenation operator. Given two sequences $A = (a, b, c)$ and $B = (d, e)$, $A \smile B$ denotes the concatenation of A and B , with the result (a, b, c, d, e) . Inserting a single element x at the end or front of the list A is written as $A \smile (x)$ or $(x) \smile A$, resulting in (a, b, c, x) or (x, a, b, c) , respectively.
\sim	“Similarity” relation used in the context of random variables and statistical distributions.
\approx	“Approximately equal” relation.
\equiv	Equivalence relation.
\leftarrow	Assignment operator: $a \leftarrow \text{expr}$ means that expression expr is evaluated and subsequently the result is assigned to the variable a .
\leftarrow^+	Incremental assignment operator: $a \leftarrow^+ b$ is equivalent to $a \leftarrow a + b$.
$:=$	Function definition operator (used in algorithms). For example, $f(x) := x^2 + 5$ defines a function $f()$ with the bound variable (formal function argument) x .
\cdots	“upto” (incrementing) iteration, used in loop constructs like for $q \leftarrow 1, \dots, K$ (with $q = 1, 2, \dots, K-1, K$).
\dots	“downto” (decrementing) iteration, for example, for $q \leftarrow K, \dots, 1$ (with $q = K, K-1, \dots, 2, 1$).

\wedge	Logical “and” operator.
\vee	Logical “or” operator.
∂	Partial derivative operator (see Sec. 6.2.1). For example, $\frac{\partial}{\partial x_i} f$ denotes the <i>first</i> derivative of the multi-dimensional function $f(x_1, x_2, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$ along variable x_i , $\frac{\partial^2}{\partial x_i^2} f$ is the <i>second</i> derivative (i.e., differentiating f twice along variable x_i), etc.
∇	Gradient operator. The gradient of a multi-dimensional function $f(x_1, x_2, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$, denoted ∇f (also ∇_f or $\text{grad } f$), is the vector of its first partial derivatives (see also Sec. C.2.2).
∇^2	Laplace operator (or <i>Laplacian</i>). The Laplacian of a multi-dimensional function $f(x_1, x_2, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$, denoted $\nabla^2 f$ (or ∇_f^2), is the sum of its second partial derivatives (see Sec. C.2.5).
$\mathbf{0}$	Zero vector, $\mathbf{0} = (0, \dots, 0)^\top$.
adj	Adjugate of a square matrix, denoted $\text{adj}(\mathbf{A})$; also called <i>adjoint</i> in older texts.
AND	Bitwise “and” operation. Example: $(0011_b \text{ AND } 1010_b) = 0010_b$ (binary) and $(3 \text{ AND } 6) = 2$ (decimal).
$\text{ArcTan}(x, y)$	Inverse tangent function. The result of $\text{ArcTan}(x, y)$ is equivalent to $\arctan(\frac{y}{x}) = \tan^{-1}(\frac{y}{x})$ but with two arguments and returning angles in the range $[-\pi, +\pi]$ (i.e., covering all four quadrants). $\text{ArcTan}(x, y)$ is equivalent to the $\text{ArcTan}[x, y]$ function in <i>Mathematica</i> and the $\text{Math.atan2}(y, x)$ method in Java (but note the reversed arguments!).
\mathbb{C}	The set of complex numbers.
card	Size (cardinality) of a set. $\text{card}(\mathcal{A}) = \mathcal{A} $ (see also Sec. 3.1).
det	Determinant of a matrix ($\det(\mathbf{A}) = \mathbf{A} $).
DFT	Discrete Fourier transform (see Sec. 18.3).
e	Euler’s constant.
\mathbf{e}	Unit vector. For example, $\mathbf{e}_x = (1, 0)^\top$ denotes the 2D unit vector in x -direction. $\mathbf{e}_\theta = (\cos \theta, \sin \theta)^\top$ is the 2D unit vector oriented at angle θ and $\mathbf{e}_i, \mathbf{e}_j, \mathbf{e}_k$ are the unit vectors along the coordinate axes in 3D.
exp	Exponential function: $\exp(x) = e^x$.
\mathcal{F}	Continuous Fourier transform (see Sec. 18.1.4).
false	Boolean constant ($\text{false} = \neg \text{true}$).
grad	Gradient operator (see ∇).
h	Histogram of an image (see Sec. 3.1).
H	Cumulative histogram (see Sec. 3.6).
\mathbf{H}	Hessian matrix (see Sec. C.2.6).
hom	Operator for converting Cartesian to homogeneous coordinates. $\text{hom}(\mathbf{x}) = \underline{\mathbf{x}}$ maps the Cartesian point \mathbf{x} to a corresponding homogeneous point $\underline{\mathbf{x}}$; the reverse mapping is denoted $\text{hom}^{-1}(\underline{\mathbf{x}}) = \mathbf{x}$ (see Sec. B.5).
i	Imaginary unit ($i^2 = -1$), see Sec. A.3.

Appendix A
MATHEMATICAL SYMBOLS
AND NOTATION

I	Image with scalar pixel values (e.g., an intensity or grayscale image). $I(u, v) \in \mathbb{R}$ is the pixel value at position (u, v)
\mathbf{I}	Vector-valued image, for example, a RGB color image with 3D color vectors $\mathbf{I}(u, v) \in \mathbb{R}^3$ at position (u, v) .
\mathbf{I}_n	Identity matrix of size $n \times n$. For example, $\mathbf{I}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ is the 2×2 identity matrix.
\mathbf{J}	Jacobian matrix (see Sec. C.2.1).
L_1, L_2, L_∞	Common distance measures or <i>norms</i> (see Eqns. (15.23)–(15.25)).
$M \times N$	Domain of pixel coordinates (u, v) for an image with M columns (width) and N rows (height); used as a shortcut notation for the set $\{0, \dots, M-1\} \times \{0, \dots, N-1\}$.
mod	Modulus operator: $(a \bmod b)$ is the remainder of the <i>integer</i> division $a \div b$ (see Sec. F.1.2).
μ	Arithmetic mean value.
\mathbb{N}	The set of natural numbers; $\mathbb{N} = \{1, 2, 3, \dots\}$, $\mathbb{N}_0 = \{0, 1, 2, \dots\}$.
nil	Null (“nothing”) constant, typically used in algorithms to denote an invalid quantity (similar to <code>null</code> in Java).
p	Discrete probability density function (see Sec. 4.6.1).
P	Discrete probability distribution function or cumulative probability density (see Sec. 4.6.1).
Q	Quadrilateral (see Sec. 21.1.4).
\mathbb{R}	The set of real numbers.
R, G, B	Red, green and blue color components.
rank	Rank of a matrix \mathbf{A} , denoted by $\text{rank}(\mathbf{A})$.
round	Rounding function: returns the integer closest to the scalar $x \in \mathbb{R}$. $\text{round}(x) \equiv \lfloor x + 0.5 \rfloor$.
σ	Standard deviation (square root of the <i>variance</i> σ^2).
S_1	Unit square (see Sec. 21.1.4).
sgn	“Sign” or “signum” function: $\text{sgn}(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$
τ	Interval in time or space.
t	Continuous time variable.
t	Threshold value.
\top	Transpose of a vector (\mathbf{a}^\top) or matrix (\mathbf{A}^\top).
trace	Trace (sum of the diagonal elements) of a matrix, e.g., $\text{trace}(\mathbf{A})$.
true	Boolean constant ($\text{true} = \neg\text{false}$).
$\mathbf{u} = (u, v)$	Discrete 2D coordinate variable with $u, v \in \mathbb{Z}$.
$\mathbf{x} = (x, y)$	Continuous 2D coordinate variable with $x, y \in \mathbb{R}$.
XOR	Bitwise “xor” (exclusive OR) operator. Example: $(0011_b \text{ XOR } 1010_b) = 1001_b$ (binary) and $(3 \text{ XOR } 6) = 5$ (decimal).
\mathbb{Z}	The set of integers.

$ \mathcal{A} $	The size of the set \mathcal{A} (equal to $\text{card}(\mathcal{A})$).
$\forall_x \dots$	“All” quantifier (for all x, \dots).
$\exists_x \dots$	“Exists” quantifier (there is some x for which \dots).
\cup	Set union (e.g., $\mathcal{A} \cup \mathcal{B}$).
\cap	Set intersection (e.g., $\mathcal{A} \cap \mathcal{B}$).
$\bigcup_i \mathcal{A}_i$	Union of multiple sets \mathcal{A}_i .
$\bigcap_i \mathcal{A}_i$	Intersection over multiple sets \mathcal{A}_i .
\setminus	Set difference: if $x \in \mathcal{A} \setminus \mathcal{B}$, then $x \in \mathcal{A}$ and $x \notin \mathcal{B}$.

A.3 Complex Numbers

Basic relations:

$$z = a + i \cdot b \quad (\text{with } z, i \in \mathbb{C}, a, b \in \mathbb{R}, i^2 = -1) \quad (\text{A.1})$$

$$s \cdot z = s \cdot a + i \cdot s \cdot b \quad (\text{for } s \in \mathbb{R}) \quad (\text{A.2})$$

$$|z| = \sqrt{a^2 + b^2} \quad (\text{A.3})$$

$$|s \cdot z| = s \cdot |z| \quad (\text{A.4})$$

$$z = a + i \cdot b = |z| \cdot (\cos \psi + i \cdot \sin \psi) \quad (\text{A.5})$$

$$= |z| \cdot e^{i \cdot \psi} \quad (\text{with } \psi = \text{ArcTan}(a, b)) \quad (\text{A.6})$$

$$\text{Re}(a + i \cdot b) = a \quad \text{Re}(e^{i \cdot \varphi}) = \cos \varphi \quad (\text{A.7})$$

$$\text{Im}(a + i \cdot b) = b \quad \text{Im}(e^{i \cdot \varphi}) = \sin \varphi \quad (\text{A.8})$$

$$e^{i \cdot \varphi} = \cos \varphi + i \cdot \sin \varphi \quad (\text{A.9})$$

$$e^{-i \cdot \varphi} = \cos \varphi - i \cdot \sin \varphi \quad (\text{A.10})$$

$$\cos(\varphi) = \frac{1}{2} \cdot (e^{i \cdot \varphi} + e^{-i \cdot \varphi}) \quad (\text{A.11})$$

$$\sin(\varphi) = \frac{1}{2i} \cdot (e^{i \cdot \varphi} - e^{-i \cdot \varphi}) \quad (\text{A.12})$$

$$z^* = a - i \cdot b \quad (\text{complex conjugate}) \quad (\text{A.13})$$

$$z \cdot z^* = z^* \cdot z = |z|^2 = a^2 + b^2 \quad (\text{A.14})$$

$$z^0 = (a + i \cdot b)^0 = (1 + i \cdot 0) = 1 \quad (\text{A.15})$$

Arithmetic operations:

$$z_1 = (a_1 + i \cdot b_1) = |z_1| e^{i \cdot \varphi_1} \quad (\text{A.16})$$

$$z_2 = (a_2 + i \cdot b_2) = |z_2| e^{i \cdot \varphi_2} \quad (\text{A.17})$$

$$z_1 + z_2 = (a_1 + a_2) + i \cdot (b_1 + b_2), \quad (\text{A.18})$$

$$z_1 \cdot z_2 = (a_1 \cdot a_2 - b_1 \cdot b_2) + i \cdot (a_1 \cdot b_2 + b_1 \cdot a_2) \quad (\text{A.19})$$

$$= |z_1| \cdot |z_2| \cdot e^{i \cdot (\varphi_1 + \varphi_2)} \quad (\text{A.20})$$

$$\frac{z_1}{z_2} = \frac{a_1 \cdot a_2 + b_1 \cdot b_2}{a_2^2 + b_2^2} + i \cdot \frac{a_2 \cdot b_1 - a_1 \cdot b_2}{a_2^2 + b_2^2} = \frac{|z_1|}{|z_2|} \cdot e^{i \cdot (\varphi_1 - \varphi_2)} \quad (\text{A.21})$$

Appendix B

Linear Algebra

This part contains a compact set of elementary tools and concepts from algebra and calculus that are referenced in the main text. Many good textbooks (probably including some of your school books) are available on this subject, for example, [35, 36, 145, 264]. For numerical aspects of linear algebra see [160, 190].

B.1 Vectors and Matrices

Here we describe the basic notation for vectors in two and three dimensions. Let

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \quad (\text{B.1})$$

denote vectors \mathbf{a}, \mathbf{b} in 2D, and analogously

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \quad (\text{B.2})$$

vectors in 3D (with $a_i, b_i \in \mathbb{R}$). Vectors are used to describe 2D or 3D points (relative to the origin of the coordinate system) or the displacement between two arbitrary points in the corresponding space.

We commonly use upper-case letters to denote a *matrix*, for example,

$$\mathbf{A} = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{pmatrix}. \quad (\text{B.3})$$

This matrix consists of 3 rows and 2 columns; in other words, \mathbf{A} is of size $(3, 2)$. Its individual elements are referenced as $A_{i,j}$, where i is the *row* index (vertical coordinate) and j is the *column* index (horizontal coordinate).¹

¹ Note that the usual notation for matrix coordinates is (unlike image coordinates) vertical-first!

The *transpose* of \mathbf{A} , denoted \mathbf{A}^\top , is obtained by exchanging rows and columns, that is,

$$\mathbf{A}^\top = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{pmatrix}^\top = \begin{pmatrix} A_{0,0} & A_{1,0} & A_{2,0} \\ A_{0,1} & A_{1,1} & A_{2,1} \end{pmatrix}. \quad (\text{B.4})$$

The *inverse* of a square matrix \mathbf{A} is denoted \mathbf{A}^{-1} , such that

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I} \quad \text{and} \quad \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I} \quad (\text{B.5})$$

(\mathbf{I} is the identity matrix). Note that not every square matrix has an inverse. Calculation of the inverse can be performed in closed form up to the size $(3, 3)$; for example, see Eqn. (21.29) and Eqn. (24.47). In general, the use of standard numerical methods is recommended (see Sec. B.6).

B.1.1 Column and Row Vectors

For practical purposes, a vector can be considered a special case of a matrix. In particular, a the m -dimensional *column* vector

$$\mathbf{a} = \begin{pmatrix} a_0 \\ \vdots \\ a_{m-1} \end{pmatrix} \quad (\text{B.6})$$

corresponds to a matrix of size $(m, 1)$, while its transpose \mathbf{a}^\top is a *row* vector and thus like a matrix of size $(1, m)$. By default, and unless otherwise noted, any vector is implicitly assumed to be a *column* vector.

B.1.2 Length (Norm) of a Vector

The *length* or *Euclidean norm* (L_2 norm) of a vector $\mathbf{a} = (a_1, \dots, a_{m-1})^\top$, denoted $\|\mathbf{a}\|$, is defined as

$$\|\mathbf{a}\| = \left(\sum_{i=0}^{m-1} a_i^2 \right)^{1/2}. \quad (\text{B.7})$$

For example, the length of the 3D vector $\mathbf{x} = (x, y, z)^\top$ is

$$\|\mathbf{x}\| = \sqrt{x^2 + y^2 + z^2}. \quad (\text{B.8})$$

B.2 Matrix Multiplication

B.2.1 Scalar Multiplication

The product of a real-valued matrix and a scalar value $s \in \mathbb{R}$ is defined as

$$s \cdot \mathbf{A} = \mathbf{A} \cdot s = [s \cdot A_{i,j}] = \begin{pmatrix} s \cdot A_{0,0} & \cdots & s \cdot A_{0,n-1} \\ \vdots & \ddots & \vdots \\ s \cdot A_{m-1,0} & \cdots & s \cdot A_{m-1,n-1} \end{pmatrix}. \quad (\text{B.9})$$

B.2.2 Product of Two Matrices

We say that a matrix is of size (m, n) if it consists of m rows and n columns. Given two matrices \mathbf{A}, \mathbf{B} of size (m, n) and (p, q) , respectively, the product $\mathbf{A} \cdot \mathbf{B}$ is only defined if $n = p$. Thus the number of columns (n) in \mathbf{A} must always match the number of rows (p) in \mathbf{B} . The result is a new matrix \mathbf{C} of size (m, q) , that is,

$$\begin{aligned}\mathbf{C} = \mathbf{A} \cdot \mathbf{B} &= \underbrace{\begin{pmatrix} A_{0,0} & \dots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \dots & A_{m-1,n-1} \end{pmatrix}}_{(m,n)} \cdot \underbrace{\begin{pmatrix} B_{0,0} & \dots & B_{0,q-1} \\ \vdots & \ddots & \vdots \\ B_{n-1,0} & \dots & B_{n-1,q-1} \end{pmatrix}}_{(n,q)} \\ &= \underbrace{\begin{pmatrix} C_{0,0} & \dots & C_{0,q-1} \\ \vdots & \ddots & \vdots \\ C_{m-1,0} & \dots & C_{m-1,q-1} \end{pmatrix}}_{(m,q)},\end{aligned}\quad (\text{B.10})$$

with the elements

$$C_{ij} = \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j}, \quad (\text{B.11})$$

for $i = 0, \dots, m-1$ and $j = 0, \dots, q-1$. Note that this product is not commutative, that is, $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$ in general.

B.2.3 Matrix-Vector Products

The product $\mathbf{A} \cdot \mathbf{x}$ between a matrix \mathbf{A} and a vector \mathbf{x} is only a special case of the matrix-matrix multiplication given in Eqn. (B.10). In particular, if $\mathbf{x} = (x_0, \dots, x_{n-1})^\top$ is a n -dimensional *column* vector (i.e., a matrix of size $(n, 1)$), then the multiplication

$$\underbrace{\mathbf{y}}_{(m,1)} = \underbrace{\mathbf{A}}_{(m,n)} \cdot \underbrace{\mathbf{x}}_{(n,1)} \quad (\text{B.12})$$

is only defined if the matrix \mathbf{A} is of size (m, n) , for arbitrary $m \geq 1$. The result \mathbf{y} is a *column* vector of length m (equivalent to a matrix of size $(m, 1)$). For example (with $m = 2, n = 3$),

$$\mathbf{A} \cdot \mathbf{x} = \underbrace{\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}}_{(2,3)} \cdot \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{(3,1)} = \underbrace{\begin{pmatrix} A \cdot x + B \cdot y + C \cdot z \\ D \cdot x + E \cdot y + F \cdot z \end{pmatrix}}_{(2,1)}. \quad (\text{B.13})$$

Here \mathbf{A} operates on the column vector \mathbf{x} “from the left”, that is, $\mathbf{A} \cdot \mathbf{x}$ is the *left-sided* matrix-vector product of \mathbf{A} and \mathbf{x} .

Similarly, a *right-sided* multiplication of a *row* vector \mathbf{x}^\top of length m with a matrix of size (m, n) is performed as

$$\underbrace{\mathbf{x}^\top}_{(1,m)} \cdot \underbrace{\mathbf{B}}_{(m,n)} = \underbrace{\mathbf{z}}_{(1,n)}, \quad (\text{B.14})$$

where the result \mathbf{z} is a n -dimensional *row* vector; for example (again with $m = 2, n = 3$),

$$\mathbf{x}^\top \cdot \mathbf{B} = \underbrace{(x, y)}_{(1,2)} \cdot \underbrace{\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}}_{(2,3)} = \underbrace{(x \cdot A + y \cdot D, x \cdot B + y \cdot E, x \cdot C + y \cdot F)}_{(1,3)}. \quad (\text{B.15})$$

In general, if $\mathbf{A} \cdot \mathbf{x}$ is defined, then

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{x}^\top \cdot \mathbf{A}^\top)^\top \quad \text{and} \quad (\mathbf{A} \cdot \mathbf{x})^\top = \mathbf{x}^\top \cdot \mathbf{A}^\top. \quad (\text{B.16})$$

Thus, any right-sided matrix-vector product $\mathbf{A} \cdot \mathbf{x}$ can also be calculated as a left-sided product $\mathbf{x}^\top \cdot \mathbf{A}^\top$ by transposing the corresponding matrix \mathbf{A} and vector \mathbf{x} .

B.3 Vector Products

Products between vectors are a common cause of confusion, mainly because the same symbol (\cdot) is used to denote widely different operators.

B.3.1 Dot (Scalar) Product

The *dot* product (also called *scalar* or *inner* product) of two vectors $\mathbf{a} = (a_0, \dots, a_{n-1})^\top$, $\mathbf{b} = (b_0, \dots, b_{n-1})^\top$ of the same length n is defined as

$$x = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i \cdot b_i. \quad (\text{B.17})$$

Thus the result x is a scalar value (hence the name of this product). If we write this as the product of a row and a column vector, as in Eqn. (B.14),

$$\underbrace{x}_{(1,1)} = \underbrace{\mathbf{a}^\top}_{(1,n)} \cdot \underbrace{\mathbf{b}}_{(n,1)}, \quad (\text{B.18})$$

we conclude that the result x is a matrix of size $(1, 1)$, that is, a single scalar value. The dot product can be viewed as the *projection* of one vector onto the other, with the relation

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \cdot \|\mathbf{b}\| \cdot \cos(\alpha), \quad (\text{B.19})$$

where α is angle enclosed by the vectors \mathbf{a} and \mathbf{b} . As a consequence, the dot product is *zero* if the two vectors are *orthogonal* to each other.

The dot product of a vector with *itself* gives the square of its length (see Eqn. (B.7)), that is,

$$\mathbf{a} \cdot \mathbf{a} = \sum_{i=0}^{n-1} a_i^2 = \|\mathbf{a}\|^2. \quad (\text{B.20})$$

B.3.2 Outer Product

The outer product of two vectors $\mathbf{a} = (a_0, \dots, a_{m-1})^\top$, $\mathbf{b} = (b_0, \dots, b_{n-1})^\top$ of length m and n , respectively, is defined as

$$\mathbf{M} = \mathbf{a} \otimes \mathbf{b} = \mathbf{a} \cdot \mathbf{b}^\top = \begin{pmatrix} a_0 b_0 & a_0 b_1 & \dots & a_0 b_{n-1} \\ a_1 b_0 & a_1 b_1 & \dots & a_1 b_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1} b_0 & a_{m-1} b_1 & \dots & a_{m-1} b_{n-1} \end{pmatrix}. \quad (\text{B.21})$$

Thus the result is a *matrix* \mathbf{M} with m rows and n columns and elements $M_{ij} = a_i \cdot b_j$, for $i = 0, \dots, m-1$ and $j = 1, \dots, n-1$. Note that $\mathbf{a} \cdot \mathbf{b}^\top$ in Eqn. (B.21) denotes the ordinary (matrix) product of the column vector \mathbf{a} (of size $m \times 1$) and the row vector \mathbf{b}^\top (of size $1 \times n$), as defined in Eqn. (B.10). The outer product is a special case of the *Kronecker* product (\otimes) which generally operates on pairs of matrices.

B.3.3 Cross Product

Although the cross product (\times) is generally defined for n -dimensional vectors, it is almost exclusively used in the 3D case, where the result is geometrically easy to understand. For a pair of 3D vectors, $\mathbf{a} = (a_0, a_1, a_2)^\top$ and $\mathbf{b} = (b_0, b_1, b_2)^\top$, the *cross product* is defined as

$$\mathbf{c} = \mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \times \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_2 - a_2 \cdot b_1 \\ a_2 \cdot b_0 - a_0 \cdot b_2 \\ a_0 \cdot b_1 - a_1 \cdot b_0 \end{pmatrix}. \quad (\text{B.22})$$

In the 3D case, the *cross product* is another 3D vector that is perpendicular to both of the original vectors.² The magnitude (length) of the vector \mathbf{c} relates to the angle θ between \mathbf{a} and \mathbf{b} as

$$\|\mathbf{c}\| = \|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \cdot \|\mathbf{b}\| \cdot \sin(\theta). \quad (\text{B.23})$$

The quantity $\|\mathbf{a} \times \mathbf{b}\|$ corresponds to the area of the parallelogram spanned by the vectors \mathbf{a} and \mathbf{b} .

B.4 Eigenvectors and Eigenvalues

This section gives an elementary introduction to eigenvectors and eigenvalues, which are mentioned at several places in the main text (see also [27, 64]). In general, the eigenvalue problem is to find solutions $\mathbf{x} \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$ for the linear equation

$$\mathbf{A} \cdot \mathbf{x} = \lambda \cdot \mathbf{x}, \quad (\text{B.24})$$

with the given square matrix \mathbf{A} of size (n, n) . Any non-trivial³ solution \mathbf{x} is an *eigenvector* of \mathbf{A} and the scalar λ (which may be

² For dimensions greater than three, the definition (and calculation) of the cross product is considerably more involved.

³ An obvious but trivial solution is $\mathbf{x} = \mathbf{0}$ (where $\mathbf{0}$ denotes the zero-vector).

B.4 EIGENVECTORS AND EIGENVALUES

complex-valued) is the associated *eigenvalue*. Eigenvalue and eigenvectors thus always come in pairs $\langle \lambda_j, \mathbf{x}_j \rangle$, usually called *eigenpairs*. Geometrically speaking, applying the matrix \mathbf{A} to an eigenvector only changes the vector's *magnitude* or *length* (by the associated eigenvalue λ), but not its orientation in space. Equation (B.24) can be rewritten as

$$\mathbf{A} \cdot \mathbf{x} - \lambda \cdot \mathbf{x} = \mathbf{0} \quad \text{or} \quad (\mathbf{A} - \lambda \cdot \mathbf{I}_n) \cdot \mathbf{x} = \mathbf{0}, \quad (\text{B.25})$$

where \mathbf{I}_n is the (n, n) identity matrix. This homogeneous linear equation has non-trivial solutions only if the matrix $(\mathbf{A} - \lambda \cdot \mathbf{I}_n)$ is *singular*, that is, its rank is *less* than n and thus its determinant $\det()$ is zero, that is,

$$\det(\mathbf{A} - \lambda \cdot \mathbf{I}_n) = 0. \quad (\text{B.26})$$

Equation (B.26) is called the “characteristic equation” of the matrix \mathbf{A} and can be expanded to a n -th order polynomial in λ . This polynomial has a maximum of n distinct roots, which are the eigenvalues of \mathbf{A} (that is, solutions to Eqn. (B.26)). A matrix of size (n, n) thus has up to n non-distinct eigenvectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, each with an associated eigenvalue $\lambda_1, \lambda_2, \dots, \lambda_n$.

If they exist, the *eigenvalues* of a matrix are *unique*, but the associated *eigenvectors* are not! This results from the fact that, if Eqn. (B.24) is satisfied for a vector \mathbf{x} (and the associated eigenvalue λ), it also applies to any *scaled* vector $s\mathbf{x}$, that is,

$$\mathbf{A} \cdot s\mathbf{x} = \lambda \cdot s\mathbf{x}, \quad (\text{B.27})$$

for arbitrary $s \in \mathbb{R}$ (and $s \neq 0$). Thus, if \mathbf{x} is an eigenvector of \mathbf{A} , then $s\mathbf{x}$ is also an (equivalent) eigenvector.

Note that the eigenvalues of a real-valued matrix may generally be complex. However, (as an important special case) if the matrix \mathbf{A} is *real* and *symmetric*, all its eigenvalues are guaranteed to be *real*.

Example

For the real-valued (non-symmetric) 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} 3 & -2 \\ -4 & 1 \end{pmatrix},$$

the two eigenvalues and their associated eigenvectors are

$$\lambda_1 = 5, \quad \mathbf{x}_1 = s \cdot \begin{pmatrix} 4 \\ -4 \end{pmatrix}, \quad \text{and} \quad \lambda_2 = -1, \quad \mathbf{x}_2 = s \cdot \begin{pmatrix} -2 \\ -4 \end{pmatrix},$$

for any nonzero $s \in \mathbb{R}$. The result can be easily verified by inserting pairs $\langle \lambda_1, \mathbf{x}_1 \rangle$ and $\langle \lambda_2, \mathbf{x}_2 \rangle$, respectively, into Eqn. (B.24).

B.4.1 Calculation of Eigenvalues

Special case: 2×2 matrix

For the special (but frequent) case of $n = 2$, the solution can be found in closed form (and without any software libraries). In this case, the characteristic equation (Eqn. (B.26)) reduces to

1: **RealEigenValues2x2** (A, B, C, D)

Input: $A, B, C, D \in \mathbb{R}$, the elements of a real-valued 2×2 matrix $\mathbf{A} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$. Returns an ordered sequence of real-valued eigenpairs $\langle \lambda_i, \mathbf{x}_i \rangle$ for \mathbf{A} , or nil if the matrix has no real-valued eigenvalues.

```

2:    $R \leftarrow \frac{A+D}{2}$ 
3:    $S \leftarrow \frac{A-D}{2}$ 
4:   if  $(S^2 + B \cdot C) < 0$  then
5:     return nil                                 $\triangleright \mathbf{A}$  has no real-valued eigenvalues
6:   else
7:      $T \leftarrow \sqrt{S^2 + B \cdot C}$ 
8:      $\lambda_1 \leftarrow R + T$                        $\triangleright$  eigenvalue  $\lambda_1$ 
9:      $\lambda_2 \leftarrow R - T$                        $\triangleright$  eigenvalue  $\lambda_2$ 
10:    if  $(A - D) \geq 0$  then
11:       $\mathbf{x}_1 \leftarrow (S + T, C)^\top$            $\triangleright$  eigenvector  $\mathbf{x}_1$ 
12:       $\mathbf{x}_2 \leftarrow (B, -S - T)^\top$            $\triangleright$  eigenvector  $\mathbf{x}_2$ 
13:    else
14:       $\mathbf{x}_1 \leftarrow (B, -S + T)^\top$            $\triangleright$  eigenvector  $\mathbf{x}_1$ 
15:       $\mathbf{x}_2 \leftarrow (S - T, C)^\top$            $\triangleright$  eigenvector  $\mathbf{x}_2$ 
16:    return  $(\langle \lambda_1, \mathbf{x}_1 \rangle, \langle \lambda_2, \mathbf{x}_2 \rangle)$            $\triangleright \lambda_1 \geq \lambda_2$ 
```

B.4 EIGENVECTORS AND EIGENVALUES

Alg. B.1

Calculating the real eigenvalues and eigenvectors for a 2×2 real-valued matrix \mathbf{A} . If the matrix has real eigenvalues, an ordered sequence of two “eigenpairs” $\langle \lambda_i, \mathbf{x}_i \rangle$, each containing the eigenvalue λ_i and the associated eigenvector \mathbf{x}_i , is returned ($i = 1, 2$). The resulting sequence is ordered by decreasing eigenvalues. nil is returned if \mathbf{A} has no real eigenvalues.

$$\det(\mathbf{A} - \lambda \cdot \mathbf{I}_2) = \left| \begin{pmatrix} A & B \\ C & D \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right| = \begin{vmatrix} A-\lambda & B \\ C & D-\lambda \end{vmatrix} \quad (\text{B.28})$$

$$= \lambda^2 - (A + D) \cdot \lambda + (AD - BC) = 0. \quad (\text{B.29})$$

The two possible solutions to this quadratic equation,

$$\begin{aligned} \lambda_{1,2} &= \frac{A + D}{2} \pm \left[\left(\frac{A + D}{2} \right)^2 - (AD - BC) \right]^{1/2} \\ &= \frac{A + D}{2} \pm \left[\left(\frac{A - D}{2} \right)^2 + BC \right]^{1/2} \\ &= R \pm \sqrt{S^2 + BC}, \end{aligned} \quad (\text{B.30})$$

are the eigenvalues of the matrix \mathbf{A} , with

$$\begin{aligned} \lambda_1 &= R + \sqrt{S^2 + B \cdot C}, \\ \lambda_2 &= R - \sqrt{S^2 + B \cdot C}. \end{aligned} \quad (\text{B.31})$$

Both λ_1, λ_2 are real-valued if the term under the square root is positive, that is, if

$$S^2 + B \cdot C = \left(\frac{A - D}{2} \right)^2 + B \cdot C \geq 0. \quad (\text{B.32})$$

In particular, if the matrix is *symmetric* (i.e., $B = C$), this condition is guaranteed (because $B \cdot C \geq 0$). In this case, $\lambda_1 \geq \lambda_2$. Algorithm B.1⁴ summarizes the closed-form computation of the eigenvalues and eigenvectors of a 2×2 matrix.

⁴ See [27] and its reprint in [28, Ch. 5].

General case: $n \times n$

In general, proven numerical software should be used for eigenvalue calculations. See the example using the Apache Commons Math library in Sec. B.6.5.

B.5 Homogeneous Coordinates

Homogeneous coordinates are an alternative representation of points in multi-dimensional space. They are commonly used in 2D and 3D geometry because they can greatly simplify the description of certain transformations. For example, affine and projective transformations become matrices with homogeneous coordinates and the composition of transformations can be performed by simple matrix multiplication.⁵

To convert a given n -dimensional *Cartesian* point $\mathbf{x} = (x_0, \dots, x_{n-1})^\top$ to *homogeneous* coordinates $\underline{\mathbf{x}}$, we use the notation⁶

$$\text{hom}(\mathbf{x}) = \underline{\mathbf{x}}. \quad (\text{B.33})$$

This operation increases the dimensionality of the original vector by one by inserting the additional element 1, that is,

$$\text{hom} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 1 \end{pmatrix} = \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \\ \underline{x}_n \end{pmatrix}. \quad (\text{B.34})$$

Note that the homogeneous representation of a Cartesian vector is not unique, but every multiple of the homogeneous vector is an equivalent representation of \mathbf{x} . Thus any scaled homogeneous vector $\underline{\mathbf{x}}' = s \cdot \underline{\mathbf{x}}$ (with $s \in \mathbb{R}$, $s \neq 0$) corresponds to the *same* Cartesian vector (see also Eqn. (B.39)).

To convert a given homogeneous point $\underline{\mathbf{x}} = (\underline{x}_0, \dots, \underline{x}_n)^\top$ back to Cartesian coordinates \mathbf{x} we simply write

$$\text{hom}^{-1}(\underline{\mathbf{x}}) = \mathbf{x}. \quad (\text{B.35})$$

This operation can be easily derived as

$$\text{hom}^{-1} \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \\ \underline{x}_n \end{pmatrix} = \frac{1}{\underline{x}_n} \cdot \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}, \quad (\text{B.36})$$

provided that $\underline{x}_n \neq 0$. Two homogeneous points $\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2$ are considered *equivalent* (\equiv), if they represent the same Cartesian point, that is,

$$\underline{\mathbf{x}}_1 \equiv \underline{\mathbf{x}}_2 \Leftrightarrow \text{hom}^{-1}(\underline{\mathbf{x}}_1) = \text{hom}^{-1}(\underline{\mathbf{x}}_2). \quad (\text{B.37})$$

It follows from Eqn. (B.36) that

⁵ See Chapter 21, Sec. 21.1.2.

⁶ The operator $\text{hom}()$ is introduced here for convenience and clarity.

$$\hom^{-1}(\underline{x}) = \hom^{-1}(s \cdot \underline{x}) \quad (\text{B.38})$$

for any nonzero factor $s \in \mathbb{R}$. Thus, as mentioned earlier, any scaled homogeneous point corresponds to the same Cartesian point, that is,

$$\underline{x} \equiv s \cdot \underline{x}. \quad (\text{B.39})$$

For example, for the Cartesian point $\underline{x} = (3, 7, 2)^\top$, the homogeneous coordinates

$$\hom(\underline{x}) = \begin{pmatrix} 3 \\ 7 \\ 2 \\ 1 \end{pmatrix} \equiv \begin{pmatrix} -3 \\ -7 \\ -2 \\ -1 \end{pmatrix} \equiv \begin{pmatrix} 9 \\ 31 \\ 6 \\ 3 \end{pmatrix} \equiv \begin{pmatrix} 30 \\ 70 \\ 20 \\ 10 \end{pmatrix} \dots \quad (\text{B.40})$$

are all equivalent. Homogeneous coordinates can be used for vector spaces of arbitrary dimension, including 2D coordinates.

B.6 Basic Matrix-Vector Operations with the Apache Commons Math Library

It is recommended to use proven standard software, such as the *Apache Commons Math*⁷ (ACM) library, for any non-trivial linear algebra calculation.

B.6.1 Vectors and Matrices

The basic data structures for representing vectors and matrices are `RealVector` and `RealMatrix`, respectively. The following ACM examples show the conversion from and to simple Java arrays of element-type `double`:

```
import org.apache.commons.math3.linear.MatrixUtils;
import org.apache.commons.math3.linear.RealMatrix;
import org.apache.commons.math3.linear.RealVector;

// Data given as simple arrays:
double[] xa = {1, 2, 3};
double[][] Aa = {{2, 0, 1}, {0, 2, 0}, {1, 0, 2}};

// Conversion to vectors and matrices:
RealVector x = MatrixUtils.createRealVector(xa);
RealMatrix A = MatrixUtils.createRealMatrix(Aa);

// Get a single matrix element Ai,j:
int i, j; // specify row (i) and column (j)
double aij = A.getEntry(i, j);

// Set a single matrix element to a new value:
double value;
A.setEntry(i, j, value);

// Extract data to arrays again:
double[] xb = x.toArray();
double[][] Ab = A.getData();
```

⁷ <http://commons.apache.org/math/>.

```
// Transpose the matrix A:  
RealMatrix At = A.transpose();
```

B.6.2 Matrix-Vector Multiplication

The following examples show how to implement the various matrix-vector products described in Sec. B.2.3.

```
RealMatrix A = ...; // matrix A of size (m, n)  
RealMatrix B = ...; // matrix B of size (p, q), with p = n  
RealVector x = ...; // vector x of length n  
  
// Scalar multiplication C ← s · A:  
double s = ...;  
RealMatrix C = A.scalarMultiply(s);  
  
// Product of two matrices: C ← A · B:  
RealMatrix C = A.multiply(B); // C is of size (m, q)  
  
// Left-sided matrix-vector product: y ← A · x:  
RealVector y = A.operate(x);  
  
// Right-sided matrix-vector product: y ← xT · A:  
RealVector y = A.preMultiply(x);
```

B.6.3 Vector Products

The following code segments show the use of the ACM library for calculating various vector products described in Sec. B.3.

```
RealVector a, b; // vectors a, b (both of length n)  
  
// Multiplication by a scalar c ← s · a:  
double s;  
RealVector c = a.mapMultiply(s);  
  
// Dot (scalar) product x ← a · b:  
double x = a.dotProduct(b);  
  
// Outer product M ← a ⊗ b:  
RealMatrix M = a.outerProduct(b);
```

B.6.4 Inverse of a Square Matrix

The following example shows the inversion of a square matrix:

```
RealMatrix A = ...; // a square matrix  
RealMatrix Ai = MatrixUtils.inverse(A);
```

B.6.5 Eigenvalues and Eigenvectors

The following code segment illustrates the calculation of eigenvalues and eigenvectors of a square matrix A using the class `EigenDecomposition` of the Apache Commons Math API. Note that the eigenval-

ues returned by `getRealEigenvalues()` are sorted in non-increasing order. The same ordering applies to the associated eigenvectors.

```
import org.apache.commons.math3.linear.EigenDecomposition;
...
RealMatrix A = MatrixUtils.createRealMatrix(new double[][] {
    {{2, 0, 1},
     {0, 2, 0},
     {1, 0, 2}});
EigenDecomposition ed = new EigenDecomposition(A);
if (ed.hasComplexEigenvalues()) {
    System.out.println("A has complex Eigenvalues!");
}
else {
    // get all real eigenvalues:
    double[] lambda = ed.getRealEigenvalues(); // = (3, 2, 1)
    // get the associated eigenvectors:
    for (int i = 0; i < lambda.length; i++) {
        RealVector x = ed.getEigenvector(i);
        ...
    }
}
```

B.7 Solving Systems of Linear Equations

This section describes standard methods for solving systems of linear equations. Such systems appear widely and frequently in all sorts of engineering problems. Identifying them and knowing about standard solution methods is thus quite important and may save much time in any development process. In addition, the solution techniques presented here are very mature and numerically stable. Note that this section is supposed to give only a brief summary of the topic and practical implementations using the Apache Commons Math library. Further details and the underlying theory can be found in most linear algebra textbooks (e.g., [145, 190]).

Systems of linear equations generally come in the form

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ A_{2,0} & A_{2,1} & \cdots & A_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0} & A_{m-1,1} & \cdots & A_{m-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{pmatrix}, \quad (\text{B.41})$$

or, in the standard notation,

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \quad (\text{B.42})$$

where the (known) matrix \mathbf{A} is of size (m, n) , the *unknown* vector \mathbf{x} is n -dimensional, and the (known) vector \mathbf{b} is m -dimensional. Thus n corresponds to the number of unknowns and m to the number

of equations. Each row i of the matrix \mathbf{A} thus represents a single equation

$$A_{i,0} \cdot x_0 + A_{i,1} \cdot x_1 + \dots + A_{i,n-1} \cdot x_{n-1} = b_i \quad (\text{B.43})$$

$$\text{or} \quad \sum_{j=0}^{n-1} A_{i,j} \cdot x_j = b_i, \quad (\text{B.44})$$

for $i = 0, \dots, m-1$. Depending on m and n , the following situations may occur:

- If $m = n$ (i.e., \mathbf{A} is square) the number of unknowns matches the number of equations and the system typically (but not always, of course) has a unique solution (see Sec. B.7.1 below).
- If $m < n$, we have more unknowns than equations. In this case no unique solution exists (but possibly infinitely many).
- With $m > n$ the system is said to be *over-determined* and thus not solvable in general. Nevertheless, this is a frequent case that is typically handled by calculating a minimum least squares solution (see Sec. B.7.2).

B.7.1 Exact Solutions

If the number of equations (m) is equal to the number of unknowns (n) and the resulting (square) matrix \mathbf{A} is non-singular and of full rank $m = n$, the system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ can be expected to have a unique solution for \mathbf{x} . For example, the system⁸

$$\begin{aligned} 2 \cdot x_0 + 3 \cdot x_1 - 2 \cdot x_2 &= 1, \\ -x_0 + 7 \cdot x_1 + 6 \cdot x_2 &= -2, \\ 4 \cdot x_0 - 3 \cdot x_1 - 5 \cdot x_2 &= 1, \end{aligned} \quad (\text{B.45})$$

with

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & -2 \\ -1 & 7 & 6 \\ 4 & -3 & -5 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}, \quad (\text{B.46})$$

has the unique solution $\mathbf{x} = (-0.3698, 0.1780, -0.6027)^T$. The following code segment shows how the previous example is solved using class `LUDecomposition` of the ACM library:

```
import org.apache...linear.DecompositionSolver;
import org.apache...linear.LUDecomposition;

RealMatrix A = MatrixUtils.createRealMatrix(new double[] []
    {{ 2, 3, -2},
     {-1, 7, 6},
     { 4, -3, -5}});
RealVector b = MatrixUtils.createRealVector(new double[]
    {1, -2, 1});
DecompositionSolver solver =
    new LUDecomposition(A).getSolver();
RealVector x = solver.solve(b);
```

An exception is thrown if the matrix \mathbf{A} is non-square or singular.

⁸ Example taken from the *Apache Commons Math User Guide* [4].

B.7.2 Over-Determined System (Least-Squares Solutions)

If a system of linear equations has more equations than unknowns (i.e., $m > n$) it is over-determined and thus has no exact solution. In other words, there is no vector \mathbf{x} that satisfies $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ or

$$\mathbf{A} \cdot \mathbf{x} - \mathbf{b} = \mathbf{0}. \quad (\text{B.47})$$

Instead, *any* \mathbf{x} plugged into Eqn. (B.47) yields some non-zero “residual” vector ϵ , such that

$$\mathbf{A} \cdot \mathbf{x} - \mathbf{b} = \epsilon. \quad (\text{B.48})$$

A “best” solution is commonly found by minimizing the squared norm of this residual, that is, by searching for \mathbf{x} such that

$$\|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}\|^2 = \|\epsilon\|^2 \rightarrow \min. \quad (\text{B.49})$$

Several matrix decompositions can be used for calculating the “least-squares solution” of an over-determined system of linear equations. As a simple example, we add a fourth line ($m = 4$) to the system in Eqns. (B.45) and (B.46) to

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & -2 \\ -1 & 7 & 6 \\ 4 & -3 & -5 \\ 2 & -2 & -1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ -2 \\ 1 \\ 0 \end{pmatrix}, \quad (\text{B.50})$$

without changing the number of unknowns ($n = 3$). The least-squares solution to this over-determined system is (approx.) $\mathbf{x} = (-0.2339, 0.1157, -0.4942)^T$. The following code segment shows the calculation using the `SingularValueDecomposition` class of the ACM library:

```
import org.apache...linear.DecompositionSolver;
import org.apache...linear.SingularValueDecomposition;

RealMatrix A = MatrixUtils.createRealMatrix(new double[][] [
    {2, 3, -2},
    {-1, 7, 6},
    {4, -3, -5},
    {2, -2, -1});
RealVector b = MatrixUtils.createRealVector(new double[]
    {1, -2, 1, 0});
DecompositionSolver solver =
    new SingularValueDecomposition(A).getSolver();
RealVector x = solver.solve(b);
```

Alternatively, an instance of `QRDecomposition` could be used for calculating the least-squares solution. If an *exact* solution exists (see Sec. B.7.1), it is the same as the least-squares solution (with zero residual $\epsilon = \mathbf{0}$).

Appendix C

Calculus

This part outlines selected topics from calculus that may serve as a useful supplement to Chapters 6, 16, 17, 24, and 25, in particular.

C.1 Parabolic Fitting

Given a single-variable (1D), discrete function $g: \mathbb{Z} \mapsto \mathbb{R}$, it is sometimes useful to locally fit a quadratic (parabolic) function, for example, for precisely locating a maximum or minimum position.

C.1.1 Fitting a Parabolic Function to Three Sample Points

For a quadratic function (second-order polynomial)

$$y = f(x) = a \cdot x^2 + b \cdot x + c \quad (\text{C.1})$$

with parameters a, b, c to pass through a given set of three sample points $\mathbf{p}_i = (x_i, y_i)$, $i = 1, 2, 3$, means that the following three equations must be satisfied:

$$\begin{aligned} y_1 &= a \cdot x_1^2 + b \cdot x_1 + c, \\ y_2 &= a \cdot x_2^2 + b \cdot x_2 + c, \\ y_3 &= a \cdot x_3^2 + b \cdot x_3 + c. \end{aligned} \quad (\text{C.2})$$

Written in the standard matrix form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, or

$$\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad (\text{C.3})$$

the unknown coefficient vector $\mathbf{x} = (a, b, c)^\top$ is directly found as

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} = \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad (\text{C.4})$$

assuming that the matrix \mathbf{A} has a non-zero determinant. Geometrically this means that the points \mathbf{p}_i must not be *collinear*.

Appendix C
CALCULUS
Example:

Fitting the sample points $\mathbf{p}_1 = (-2, 5)^\top$, $\mathbf{p}_2 = (-1, 6)^\top$, $\mathbf{p}_3 = (3, -10)^\top$ to a quadratic function, the equation to solve is (analogous to Eqn. (C.3))

$$\begin{pmatrix} 4 & -2 & 1 \\ 1 & -1 & 1 \\ 9 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix}, \quad (\text{C.5})$$

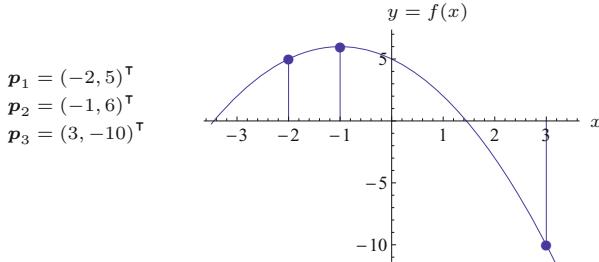
with the solution

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 4 & -2 & 1 \\ 1 & -1 & 1 \\ 9 & 3 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix} = \frac{1}{20} \cdot \begin{pmatrix} 4 & -5 & 1 \\ -8 & 5 & 3 \\ -12 & 30 & 2 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix} = \begin{pmatrix} -1 \\ -2 \\ 5 \end{pmatrix}.$$

Thus $a = -1$, $b = -2$, $c = 5$, and the equation of the quadratic fitting function is $y = -x^2 - 2x + 5$. The result for this example is shown graphically in Fig. C.1.

Fig. C.1

Fitting a quadratic function to three arbitrary sample points.



C.1.2 Locating Extrema by Quadratic Interpolation

A special situation is when the given points are positioned at $x_1 = -1$, $x_2 = 0$, and $x_3 = +1$. This is useful, for example, to estimate a continuous extremum position from successive discrete function values defined on a regular lattice. Again the objective is to fit a quadratic function (as in Eqn. (C.1)) to pass through the points $\mathbf{p}_1 = (-1, y_1)^\top$, $\mathbf{p}_2 = (0, y_2)^\top$, and $\mathbf{p}_3 = (1, y_3)^\top$. In this case, the simultaneous equations in Eqn. (C.2) simplify to

$$\begin{aligned} y_1 &= a - b + c, \\ y_2 &= \quad \quad \quad c, \\ y_3 &= a + b + c, \end{aligned} \quad (\text{C.6})$$

with the solution

$$a = \frac{y_1 - 2 \cdot y_2 + y_3}{2}, \quad b = \frac{y_3 - y_1}{2}, \quad c = y_2. \quad (\text{C.7})$$

To estimate a local extremum position, we take the first derivative of the quadratic fitting function (Eqn. (C.1)), which is the linear function $f'(x) = 2a \cdot x + b$, and find the position \check{x} of its (single) root by solving

$$2a \cdot x + b = 0. \quad (\text{C.8})$$

With a, b taken from Eqn. (C.7), the extremal position is thus found as

$$\check{x} = \frac{-b}{2a} = \frac{y_1 - y_3}{2 \cdot (y_1 - 2y_2 + y_3)} . \quad (\text{C.9})$$

The corresponding extremal *value* can then be found by evaluating the quadratic function $f()$ at position \check{x} , that is,

$$\check{y} = f(\check{x}) = a \cdot \check{x}^2 + b \cdot \check{x} + c, \quad (\text{C.10})$$

with a, b, c as defined in Eqn. (C.7). Figure C.2 shows an example with sample points $\mathbf{p}_1 = (-1, -2)^\top$, $\mathbf{p}_2 = (0, 7)^\top$, $\mathbf{p}_3 = (1, 6)^\top$. In this case, the interpolated maximum position is at $\check{x} = 0.4$ and the corresponding maximum value is $f(\check{x}) = 7.8$.

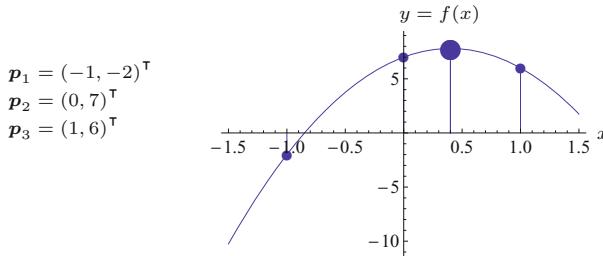


Fig. C.2

Fitting a quadratic function to three reference points at positions $x_1 = -1, x_2 = 0, x_3 = +1$. The interpolated, continuous curve has a maximum at the continuous position $\check{x} = 0.4$ (large circle).

Using the above scheme, we can interpolate any triplet of successive sample values centered around some position $u \in \mathbb{Z}$, that is, $\mathbf{p}_1 = (u-1, y_1)^\top$, $\mathbf{p}_2 = (u, y_2)^\top$, $\mathbf{p}_3 = (u+1, y_3)^\top$, with arbitrary values y_1, y_2, y_3 . In this case the estimated position of the extremum is simply (from Eqn. (C.9))

$$\check{x} = u + \frac{y_1 - y_3}{2 \cdot (y_1 - 2 \cdot y_2 + y_3)} . \quad (\text{C.11})$$

The application of quadratic interpolation to multi-variable functions is described in Sec. C.3.3.

C.2 Scalar and Vector Fields

An RGB color image $\mathbf{I}(u, v) = (I_R(u, v), I_G(u, v), I_B(u, v))$ can be considered a 2D function whose values are 3D vectors. Mathematically, this is a special case of a vector-valued function $\mathbf{f}: \mathbb{R}^n \mapsto \mathbb{R}^m$,

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(x_0, \dots, x_{n-1}) = \begin{pmatrix} f_0(\mathbf{x}) \\ \vdots \\ f_{m-1}(\mathbf{x}) \end{pmatrix}, \quad (\text{C.12})$$

which is composed of m scalar-valued functions $f_i: \mathbb{R}^n \mapsto \mathbb{R}$, each being defined on the domain of n -dimensional vectors.

A multi-variable, scalar-valued function $f: \mathbb{R}^n \mapsto \mathbb{R}$ is called a *scalar field*, while a vector-valued function $\mathbf{f}: \mathbb{R}^n \mapsto \mathbb{R}^m$ is referred to as a *vector field*.

C.2.1 The Jacobian Matrix

Assuming that the function $\mathbf{f}(\mathbf{x}) = (f_0(\mathbf{x}), \dots, f_{m-1}(\mathbf{x}))^\top$ is differentiable, the so-called *functional* or *Jacobian* matrix at a specific point $\dot{\mathbf{x}} = (\dot{x}_0, \dots, \dot{x}_{n-1})$ is defined as

$$\mathbf{J}_\mathbf{f}(\dot{\mathbf{x}}) = \begin{pmatrix} \frac{\partial}{\partial x_0} f_0(\dot{\mathbf{x}}) & \cdots & \frac{\partial}{\partial x_{n-1}} f_0(\dot{\mathbf{x}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_0} f_{m-1}(\dot{\mathbf{x}}) & \cdots & \frac{\partial}{\partial x_{n-1}} f_{m-1}(\dot{\mathbf{x}}) \end{pmatrix}. \quad (\text{C.13})$$

The Jacobian matrix is of size $m \times n$ and composed of the first derivatives of the m component functions f_0, \dots, f_{m-1} with respect to each of the n independent variables x_0, \dots, x_{n-1} . Thus each of its elements $\frac{\partial}{\partial x_j} f_i(\dot{\mathbf{x}})$ quantifies how much the value of the scalar-valued component function $f_i(\mathbf{x}) = f_i(x_0, \dots, x_{n-1})$ changes when only variable x_j is varied and all other variables remain fixed. Note that the matrix $\mathbf{J}_\mathbf{f}(\mathbf{x})$ is not constant for a given function \mathbf{f} but is different at each position $\dot{\mathbf{x}}$. In general, the Jacobian matrix is neither square (unless $m = n$) nor symmetric.

C.2.2 Gradients

Gradient of a scalar field

The gradient of a *scalar field* $f: \mathbb{R}^n \mapsto \mathbb{R}$, with $f(\mathbf{x}) = f(x_0, \dots, x_{n-1})$, at a given position $\dot{\mathbf{x}} \in \mathbb{R}^n$ is defined as

$$(\nabla f)(\dot{\mathbf{x}}) = (\text{grad } f)(\dot{\mathbf{x}}) = \begin{pmatrix} \frac{\partial}{\partial x_0} f(\dot{\mathbf{x}}) \\ \vdots \\ \frac{\partial}{\partial x_{n-1}} f(\dot{\mathbf{x}}) \end{pmatrix}. \quad (\text{C.14})$$

The resulting vector-valued function quantifies the amount of output change with respect to changing any of the input variables x_0, \dots, x_{n-1} at position $\dot{\mathbf{x}}$. Thus the gradient of a scalar field is a vector field.

The *directional* gradient of a scalar field describes how the (scalar) function value changes when the coordinates are modified along a particular direction, specified by the unit vector \mathbf{e} . We denote the directional gradient as $\nabla_{\mathbf{e}} f$ and define

$$(\nabla_{\mathbf{e}} f)(\dot{\mathbf{x}}) = (\nabla f)(\dot{\mathbf{x}}) \cdot \mathbf{e}, \quad (\text{C.15})$$

where \cdot is the scalar product (see Sec. B.3.1). The result is a scalar value that can be interpreted as the slope of the tangent on the n -dimensional surface of the scalar field at position $\dot{\mathbf{x}}$ along the direction specified by the n -dimensional unit vector $\mathbf{e} = (e_0, \dots, e_{n-1})^\top$.

Gradient of a vector field

To calculate the gradient of a *vector field* $\mathbf{f}: \mathbb{R}^n \mapsto \mathbb{R}^m$, we note that each row i in the $m \times n$ Jacobian matrix $\mathbf{J}_\mathbf{f}$ (Eqn. (C.13)) is the transposed gradient vector of the corresponding component function f_i , that is,

$$\mathbf{J}_f(\dot{\mathbf{x}}) = \begin{pmatrix} (\nabla f_0)(\dot{\mathbf{x}})^T \\ \vdots \\ (\nabla f_{m-1})(\dot{\mathbf{x}})^T \end{pmatrix}, \quad (\text{C.16})$$

and thus the Jacobian matrix is equivalent to the gradient of the vector field f ,

$$(\text{grad } f)(\dot{\mathbf{x}}) \equiv \mathbf{J}_f(\dot{\mathbf{x}}). \quad (\text{C.17})$$

Analogous to Eqn. (C.15), the *directional* gradient of the vector field is then defined as

$$(\text{grad}_{\mathbf{e}} f)(\dot{\mathbf{x}}) \equiv \mathbf{J}_f(\dot{\mathbf{x}}) \cdot \mathbf{e}, \quad (\text{C.18})$$

where \mathbf{e} is again a unit vector specifying the gradient direction and \cdot is the ordinary matrix-vector product. In this case the resulting gradient is a m -dimensional vector with one element for each component function in f .

C.2.3 Maximum Gradient Direction

In case of a scalar field $f(\mathbf{x})$, a resulting non-zero gradient vector $(\nabla f)(\dot{\mathbf{x}})$ (Eqn. (C.14)) is also the direction of the steepest ascent of $f(\mathbf{x})$ at position $\dot{\mathbf{x}}$.¹ In this case, the L_2 norm (see Sec. B.1.2) of the gradient vector, that is, $\|(\nabla f)(\dot{\mathbf{x}})\|$, corresponds to the maximum slope of f at point $\dot{\mathbf{x}}$.

In case of a vector field $f(\mathbf{x})$, the direction of maximum slope cannot be obtained directly, since the gradient is not a n -dimensional vector but its $m \times n$ Jacobian matrix. In this case, the direction of maximum change in the function f is found as the eigenvector \mathbf{x}_k of the square ($n \times n$) matrix

$$\mathbf{M} = \mathbf{J}_f^T(\dot{\mathbf{x}}) \cdot \mathbf{J}_f(\dot{\mathbf{x}}) \quad (\text{C.19})$$

that corresponds to its largest eigenvalue λ_k (see also Sec. B.4).

C.2.4 Divergence of a Vector Field

If the vector field maps to the same vector space (i.e., $f: \mathbb{R}^n \mapsto \mathbb{R}^n$), its *divergence* (div) is defined as

$$(\text{div } f)(\dot{\mathbf{x}}) = \frac{\partial}{\partial x_0} f_0(\dot{\mathbf{x}}) + \cdots + \frac{\partial}{\partial x_{n-1}} f_{n-1}(\dot{\mathbf{x}}) \quad (\text{C.20})$$

$$= \sum_{i=0}^{n-1} \frac{\partial}{\partial x_i} f_i(\dot{\mathbf{x}}) \in \mathbb{R}, \quad (\text{C.21})$$

for a given point $\dot{\mathbf{x}}$. The result is a scalar value and thus $(\text{div } f)(\dot{\mathbf{x}})$ yields a scalar field $\mathbb{R}^n \mapsto \mathbb{R}$. Note that, in this case, the Jacobian matrix \mathbf{J}_f in Eqn. (C.13) is square (of size $n \times n$) and $\text{div } f$ is equivalent to the trace of \mathbf{J}_f , that is,

$$(\text{div } f)(\dot{\mathbf{x}}) \equiv \text{trace}(\mathbf{J}_f(\dot{\mathbf{x}})). \quad (\text{C.22})$$

¹ If the gradient vector is zero, that is, if $(\nabla f)(\dot{\mathbf{x}}) = \mathbf{0}$, the direction of the gradient is undefined at position $\dot{\mathbf{x}}$.

C.2.5 Laplacian Operator

The *Laplacian* (or Laplace operator) of a scalar field $f: \mathbb{R}^n \mapsto \mathbb{R}$ is a linear differential operator, commonly denoted Δ or ∇^2 . The result of applying ∇^2 to the scalar field $f: \mathbb{R}^n \mapsto \mathbb{R}$ generates another scalar field that consists of the sum of all unmixed second-order partial derivatives of f (if existent), that is,

$$(\nabla^2 f)(\dot{\mathbf{x}}) = \frac{\partial^2}{\partial x_0^2} f(\dot{\mathbf{x}}) + \cdots + \frac{\partial^2}{\partial x_{n-1}^2} f(\dot{\mathbf{x}}) = \sum_{i=0}^{n-1} \frac{\partial^2}{\partial x_i^2} f(\dot{\mathbf{x}}). \quad (\text{C.23})$$

The result is a scalar value that is equivalent to the *divergence* (see Eqn. (C.21)) of the *gradient* (see Eqn. (C.14)) of the scalar field f , that is,

$$(\nabla^2 f)(\dot{\mathbf{x}}) = (\operatorname{div} \nabla f)s(\dot{\mathbf{x}}). \quad (\text{C.24})$$

The *Laplacian* is also found as the *trace* of the function's Hessian matrix \mathbf{H}_f (see Sec. C.2.6).

For a *vector*-valued function $\mathbf{f}: \mathbb{R}^n \mapsto \mathbb{R}^m$, the Laplacian at point $\dot{\mathbf{x}}$ is again a vector field $\mathbb{R}^n \mapsto \mathbb{R}^m$,

$$(\nabla^2 \mathbf{f})(\dot{\mathbf{x}}) = \begin{pmatrix} (\nabla^2 f_0)(\dot{\mathbf{x}}) \\ (\nabla^2 f_1)(\dot{\mathbf{x}}) \\ \vdots \\ (\nabla^2 f_{m-1})(\dot{\mathbf{x}}) \end{pmatrix} \in \mathbb{R}^m, \quad (\text{C.25})$$

that is obtained by applying the Laplacian to the individual (scalar-valued) component functions.

C.2.6 The Hessian Matrix

The Hessian matrix of a n -variable, real-valued function $f: \mathbb{R}^n \mapsto \mathbb{R}$ is the $n \times n$ square matrix composed of its second-order partial derivatives (assuming they all exist), that is,

$$\mathbf{H}_f = \begin{pmatrix} H_{0,0} & H_{0,1} & \cdots & H_{0,n-1} \\ H_{1,0} & H_{1,1} & \cdots & H_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{n-1,0} & H_{n-1,1} & \cdots & H_{n-1,n-1} \end{pmatrix} \quad (\text{C.26})$$

$$= \begin{pmatrix} \frac{\partial^2}{\partial x_0^2} f & \frac{\partial^2}{\partial x_0 \partial x_1} f & \cdots & \frac{\partial^2}{\partial x_0 \partial x_{n-1}} f \\ \frac{\partial^2}{\partial x_1 \partial x_0} f & \frac{\partial^2}{\partial x_1^2} f & \cdots & \frac{\partial^2}{\partial x_1 \partial x_{n-1}} f \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_{n-1} \partial x_0} f & \frac{\partial^2}{\partial x_{n-1} \partial x_1} f & \cdots & \frac{\partial^2}{\partial x_{n-1}^2} f \end{pmatrix}. \quad (\text{C.27})$$

Since the order of differentiation does not matter (i.e., $H_{i,j} = H_{j,i}$), \mathbf{H}_f is symmetric. Note that the Hessian is a matrix of *functions*. To evaluate the Hessian at a particular point $\dot{\mathbf{x}} \in \mathbb{R}^n$, we write

$$\mathbf{H}_f(\dot{\mathbf{x}}) = \begin{pmatrix} \frac{\partial^2}{\partial x_0^2} f(\dot{\mathbf{x}}) & \cdots & \frac{\partial^2}{\partial x_0 \partial x_{n-1}} f(\dot{\mathbf{x}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_{n-1} \partial x_0} f(\dot{\mathbf{x}}) & \cdots & \frac{\partial^2}{\partial x_{n-1}^2} f(\dot{\mathbf{x}}) \end{pmatrix}, \quad (\text{C.28})$$

C.3 OPERATIONS ON MULTI-VARIABLE, SCALAR FUNCTIONS (SCALAR FIELDS)

which is a scalar-valued matrix of size $n \times n$. As mentioned already, the *trace* of the Hessian matrix is the *Laplacian* ∇^2 of the function f , that is,

$$\nabla^2 f = \text{trace}(\mathbf{H}_f) = \sum_{i=0}^{n-1} \frac{\partial^2}{\partial x_i^2} f. \quad (\text{C.29})$$

Example

Given a 2D, continuous, grayscale image or scalar-valued intensity function $I(x, y)$, the corresponding Hessian matrix (of size 2×2) contains all second derivatives along the coordinates x, y , that is,

$$\mathbf{H}_I = \begin{pmatrix} \frac{\partial^2}{\partial x^2} I & \frac{\partial^2}{\partial x \partial y} I \\ \frac{\partial^2}{\partial y \partial x} I & \frac{\partial^2}{\partial y^2} I \end{pmatrix} = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix}, \quad (\text{C.30})$$

The elements of \mathbf{H}_I are 2D, scalar-valued functions over x, y and thus scalar fields again. Evaluating the Hessian matrix at a particular point $\dot{\mathbf{x}}$ yields the values of the second partial derivatives of I at this position,

$$\mathbf{H}_I(\dot{\mathbf{x}}) = \begin{pmatrix} \frac{\partial^2}{\partial x^2} I(\dot{\mathbf{x}}) & \frac{\partial^2}{\partial x \partial y} I(\dot{\mathbf{x}}) \\ \frac{\partial^2}{\partial y \partial x} I(\dot{\mathbf{x}}) & \frac{\partial^2}{\partial y^2} I(\dot{\mathbf{x}}) \end{pmatrix} = \begin{pmatrix} I_{xx}(\dot{\mathbf{x}}) & I_{xy}(\dot{\mathbf{x}}) \\ I_{yx}(\dot{\mathbf{x}}) & I_{yy}(\dot{\mathbf{x}}) \end{pmatrix}, \quad (\text{C.31})$$

that is, a matrix with scalar-valued elements.

C.3 Operations on Multi-Variable, Scalar Functions (Scalar Fields)

C.3.1 Estimating the Derivatives of a Discrete Function

Images are typically discrete functions (i.e., $I: \mathbb{N}^2 \mapsto \mathbb{R}$) and thus not differentiable. The derivatives can nevertheless be estimated by calculating finite differences from the pixel values in a 3×3 neighborhood, which can be expressed as a linear filter or convolution operation (*). In particular, the *first-order* derivatives $I_x = \partial I / \partial x$ and $I_y = \partial I / \partial y$ are usually estimated in the form

$$I_x \approx I * \begin{bmatrix} -0.5 & \mathbf{0} & 0.5 \end{bmatrix}, \quad I_y \approx I * \begin{bmatrix} -0.5 \\ \mathbf{0} \\ 0.5 \end{bmatrix}, \quad (\text{C.32})$$

the second-order derivatives $I_{xx} = \partial^2 I / \partial x^2$ and $I_{yy} = \partial^2 I / \partial y^2$ as

$$I_{xx} \approx I * \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}, \quad I_{yy} \approx I * \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}, \quad (\text{C.33})$$

and the mixed derivative

$$\begin{aligned} \frac{\partial^2 I}{\partial x \partial y} &= I_{xy} = I_{yx} \\ &\approx I * \begin{bmatrix} -0.5 & \mathbf{0} & 0.5 \end{bmatrix} * \begin{bmatrix} -0.5 \\ \mathbf{0} \\ 0.5 \end{bmatrix} = I * \begin{bmatrix} 0.25 & 0 & -0.25 \\ 0 & \mathbf{0} & 0 \\ -0.25 & 0 & 0.25 \end{bmatrix}. \end{aligned} \quad (\text{C.34})$$

C.3.2 Taylor Series Expansion of Functions

Single-variable functions

The Taylor series expansion (of degree d) of a single-variable function $f: \mathbb{R} \mapsto \mathbb{R}$ about a reference point a is

$$\begin{aligned} f(x) &= f(a) + f'(a) \cdot (x-a) + f''(a) \cdot \frac{(x-a)^2}{2} + \dots \\ &\quad \dots + f^{(d)}(a) \cdot \frac{(x-a)^d}{d!} + R_d \end{aligned} \quad (\text{C.35})$$

$$= f(a) + \sum_{i=1}^d f^{(i)}(a) \cdot \frac{(x-a)^i}{i!} + R_d \quad (\text{C.36})$$

$$= \sum_{i=0}^d f^{(i)}(a) \cdot \frac{(x-a)^i}{i!} + R_d, \quad (\text{C.37})$$

where R_d is the residual term.² This means that if the value $f(a)$ and the first d derivatives $f'(a), f''(a), \dots, f^{(d)}(a)$ exist and are known at some position a , the value of f at *another* point \dot{x} can be estimated (up to the residual R_d) only from the values at point a , without actually evaluating $f(\dot{x})$. Omitting the remainder R_d , the result is an *approximation* for $f(\dot{x})$, that is,

$$f(x) \approx \sum_{i=0}^d f^{(i)}(a) \cdot \frac{(x-a)^i}{i!}, \quad (\text{C.38})$$

whose accuracy depends upon d and the distance $x - a$.

Multi-variable functions

In general, for a real-valued function of n variables,

$$f(\mathbf{x}) = f(x_0, x_1, \dots, x_{n-1}) \in \mathbb{R},$$

the full Taylor series expansion about a reference point $\mathbf{a} = (a_0, \dots, a_{n-1})^\top$ is

$$\begin{aligned} f(x_0, \dots, x_{n-1}) &= f(\mathbf{a}) + \\ &\quad \sum_{i_0=1}^{\infty} \dots \sum_{i_{n-1}=1}^{\infty} \left[\frac{\partial^{i_0}}{\partial x_0^{i_0}} \dots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}} \right] f(\mathbf{a}) \cdot \frac{(x_0 - a_0)^{i_0} \dots (x_{n-1} - a_{n-1})^{i_{n-1}}}{i_1! \dots i_n!} \\ &= \sum_{i_1=0}^{\infty} \dots \sum_{i_n=0}^{\infty} \left[\frac{\partial^{i_0}}{\partial x_0^{i_0}} \dots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}} \right] f(\mathbf{a}) \cdot \frac{(x_0 - a_0)^{i_0} \dots (x_{n-1} - a_{n-1})^{i_{n-1}}}{i_0! \dots i_{n-1}!}. \end{aligned} \quad (\text{C.39})$$

² Note that $f^{(0)} = f$, $f^{(1)} = f'$, $f^{(2)} = f''$ etc., and $1! = 1$.

In Eqn. (C.39),³ the term

$$\left[\frac{\partial^{i_0}}{\partial x_0^{i_0}} \cdots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}} \right] f(\mathbf{a}) \quad (\text{C.40})$$

is the value of the function f , after applying a sequence of n partial derivatives, at the n -dimensional position \mathbf{a} . The operator $\frac{\partial^i}{\partial x_k^i}$ denotes the i -th partial derivative on the variable x_k .

To formulate Eqn. (C.39) in a more compact fashion, we define the index vector

$$\mathbf{i} = (i_0, i_1, \dots, i_{n-1}), \quad (\text{C.41})$$

(with $i_k \in \mathbb{N}_0$ and thus $\mathbf{i} \in \mathbb{N}_0^n$), and the associated operations

$$\begin{aligned} \mathbf{i}! &= i_0! \cdot i_1! \cdot \dots \cdot i_{n-1}!, \\ \mathbf{x}^{\mathbf{i}} &= x_1^{i_0} \cdot x_2^{i_1} \cdot \dots \cdot x_{n-1}^{i_{n-1}}, \\ \Sigma \mathbf{i} &= i_0 + i_1 + \dots + i_{n-1}. \end{aligned} \quad (\text{C.42})$$

As a shorthand notation for the combined partial derivative operator in Eqn. (C.40) we define

$$D^{\mathbf{i}} := \frac{\partial^{i_0}}{\partial x_0^{i_0}} \frac{\partial^{i_1}}{\partial x_1^{i_1}} \cdots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}} = \frac{\partial^{i_0+i_1+\dots+i_{n-1}}}{\partial x_0^{i_0} \partial x_1^{i_1} \cdots \partial x_{n-1}^{i_{n-1}}}. \quad (\text{C.43})$$

With these definitions, the full Taylor expansion of a multi-variable function about a point \mathbf{a} , as given in Eqn. (C.39), can be elegantly written in the form

$$f(\mathbf{x}) = \sum_{\mathbf{i} \in \mathbb{N}_0^n} D^{\mathbf{i}} f(\mathbf{a}) \cdot \frac{(\mathbf{x} - \mathbf{a})^{\mathbf{i}}}{\mathbf{i}!}. \quad (\text{C.44})$$

Note that $D^{\mathbf{i}} f$ is again a n -dimensional function $\mathbb{R}^n \mapsto \mathbb{R}$, and thus $[D^{\mathbf{i}} f](\mathbf{a})$ in Eqn. (C.44) is the scalar quantity obtained by evaluating the function $[D^{\mathbf{i}} f]$ at the n -dimensional point \mathbf{a} .

To obtain a Taylor *approximation* of order d , the sum of the indices i_1, \dots, i_n is limited to d , that is, the summation is constrained to index vectors \mathbf{i} , with $\Sigma \mathbf{i} \leq d$. The resulting formulation,

$$f(\mathbf{x}) \approx \sum_{\substack{\mathbf{i} \in \mathbb{N}_0^n \\ \Sigma \mathbf{i} \leq d}} D^{\mathbf{i}} f(\mathbf{a}) \cdot \frac{(\mathbf{x} - \mathbf{a})^{\mathbf{i}}}{\mathbf{i}!}, \quad (\text{C.45})$$

is obviously analogous to the 1D case in Eqn. (C.38).

Example: two-variable (2D) function

This example demonstrates the second-order ($d = 2$) Taylor expansion of a 2D ($n = 2$) function $f: \mathbb{R}^2 \mapsto \mathbb{R}$ around a point $\mathbf{a} = (x_a, y_a)$. By inserting into Eqn. (C.44), we get

³ Note that symbols x_0, \dots, x_{n-1} denote the individual variables, while $\dot{x}_0, \dots, \dot{x}_{n-1}$ are the coordinates of a specific point in n -dimensional space.

$$f(x, y) \approx \sum_{\substack{\mathbf{i} \in \mathbb{N}_0^2 \\ \Sigma \mathbf{i} \leq 2}} D^{\mathbf{i}} f(x_a, y_a) \cdot \frac{1}{\mathbf{i}!} \cdot \binom{x - x_a}{y - y_a}^{\mathbf{i}} \quad (\text{C.46})$$

$$= \sum_{\substack{0 \leq i, j \leq 2 \\ (i+j) \leq 2}} \frac{\partial^{i+j}}{\partial x^i \partial y^j} f(x_a, y_a) \cdot \frac{(x - x_a)^i \cdot (y - y_a)^j}{i! \cdot j!}. \quad (\text{C.47})$$

Since $d = 2$, the six permissible index vectors $\mathbf{i} = (i, j)$, with $\Sigma \mathbf{i} \leq 2$, are $(0, 0)$, $(1, 0)$, $(0, 1)$, $(1, 1)$, $(2, 0)$, and $(0, 2)$. Inserting into Eqn. (C.47), we obtain the corresponding Taylor approximation at position (\dot{x}, \dot{y}) as

$$\begin{aligned} f(x, y) &\approx \frac{\partial^0}{\partial x^0 \partial y^0} f(x_a, y_a) \cdot \frac{(x - x_a)^0 \cdot (y - y_a)^0}{1 \cdot 1} \\ &+ \frac{\partial^1}{\partial x^1 \partial y^0} f(x_a, y_a) \cdot \frac{(x - x_a)^1 \cdot (y - y_a)^0}{1 \cdot 1} \\ &+ \frac{\partial^1}{\partial x^0 \partial y^1} f(x_a, y_a) \cdot \frac{(x - x_a)^0 \cdot (y - y_a)^1}{1 \cdot 1} \\ &+ \frac{\partial^2}{\partial x^1 \partial y^1} f(x_a, y_a) \cdot \frac{(x - x_a)^1 \cdot (y - y_a)^1}{1 \cdot 1} \\ &+ \frac{\partial^2}{\partial x^2 \partial y^0} f(x_a, y_a) \cdot \frac{(x - x_a)^2 \cdot (y - y_a)^0}{2 \cdot 1} \\ &+ \frac{\partial^2}{\partial x^0 \partial y^2} f(x_a, y_a) \cdot \frac{(x - x_a)^0 \cdot (y - y_a)^2}{1 \cdot 2} \\ &= f(x_a, y_a) \end{aligned} \quad (\text{C.48})$$

$$\begin{aligned} &+ \frac{\partial}{\partial x} f(x_a, y_a) \cdot (x - x_a) + \frac{\partial}{\partial y} f(x_a, y_a) \cdot (y - y_a) \\ &+ \frac{\partial^2}{\partial x \partial y} f(x_a, y_a) \cdot (x - x_a) \cdot (y - y_a) \\ &+ \frac{1}{2} \cdot \frac{\partial^2}{\partial x^2} f(x_a, y_a) \cdot (x - x_a)^2 + \frac{1}{2} \cdot \frac{\partial^2}{\partial y^2} f(x_a, y_a) \cdot (y - y_a)^2. \end{aligned} \quad (\text{C.49})$$

It is assumed that the required derivatives of f exist, that is, f is differentiable at point (x_a, y_a) with respect to x and y up to the second order. By slightly rearranging Eqn. (C.49) to

$$\begin{aligned} f(x, y) &\approx f(x_a, y_a) + \frac{\partial}{\partial x} f(x_a, y_a) \cdot (x - x_a) + \frac{\partial}{\partial y} f(x_a, y_a) \cdot (y - y_a) \\ &+ \frac{1}{2} \cdot \left[\frac{\partial^2}{\partial x^2} f(x_a, y_a) \cdot (x - x_a)^2 + 2 \cdot \frac{\partial^2}{\partial x \partial y} f(x_a, y_a) \cdot (x - x_a) \cdot (y - y_a) \right. \\ &\quad \left. + \frac{\partial^2}{\partial y^2} f(x_a, y_a) \cdot (y - y_a)^2 \right] \end{aligned} \quad (\text{C.50})$$

we can now write the Taylor expansion in matrix-vector notation as

$$\begin{aligned} f(x, y) &\approx \tilde{f}(x, y) = f(x_a, y_a) + \left(\frac{\partial}{\partial x} f(x_a, y_a), \frac{\partial}{\partial y} f(x_a, y_a) \right) \cdot \binom{x - x_a}{y - y_a} \\ &+ \frac{1}{2} \cdot \left[(x - x_a, y - y_a) \cdot \begin{pmatrix} \frac{\partial^2}{\partial x^2} f(x_a, y_a) & \frac{\partial^2}{\partial x \partial y} f(x_a, y_a) \\ \frac{\partial^2}{\partial x \partial y} f(x_a, y_a) & \frac{\partial^2}{\partial y^2} f(x_a, y_a) \end{pmatrix} \cdot \binom{x - x_a}{y - y_a} \right] \end{aligned} \quad (\text{C.51})$$

or, even more compactly, in the form

$$\tilde{f}(\mathbf{x}) = f(\mathbf{a}) + \nabla_f^\top(\mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) + \frac{1}{2} \cdot (\mathbf{x} - \mathbf{a})^\top \cdot \mathbf{H}_f(\mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}). \quad (\text{C.52})$$

Here $\nabla_f^\top(\mathbf{a})$ denotes the (transposed) *gradient* vector of the function f at point \mathbf{a} (see Sec. C.2.2), and \mathbf{H}_f is the 2×2 *Hessian* matrix of f (see Sec. C.2.6),

$$\mathbf{H}_f(\mathbf{a}) = \begin{pmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2}{\partial x^2} f(\mathbf{a}) & \frac{\partial^2}{\partial x \partial y} f(\mathbf{a}) \\ \frac{\partial^2}{\partial x \partial y} f(\mathbf{a}) & \frac{\partial^2}{\partial y^2} f(\mathbf{a}) \end{pmatrix}. \quad (\text{C.53})$$

If the function f is *discrete*, for example, a scalar-valued image I , the required partial derivatives at some lattice point $\mathbf{a} = (u_a, v_a)^\top$ can be estimated from its 3×3 neighborhood, as described in Sec. C.3.1.

Example: three-variable (3D) function

For a 3D function $f: \mathbb{R}^3 \mapsto \mathbb{R}$, the second-order Taylor expansion ($d = 2$) is analogous to Eqns. (C.51–C.52) for the 2D case, except that now the positions $\mathbf{x} = (x, y, z)^\top$ and $\mathbf{a} = (x_a, y_a, z_a)^\top$ are 3D vectors. The associated (transposed) gradient vector is

$$\nabla_f^\top(\mathbf{a}) = \left(\frac{\partial}{\partial x} f(\mathbf{a}), \frac{\partial}{\partial y} f(\mathbf{a}), \frac{\partial}{\partial z} f(\mathbf{a}) \right), \quad (\text{C.54})$$

and the Hessian, composed of all second-order partial derivatives, is the 3×3 matrix

$$\mathbf{H}_f(\mathbf{a}) = \begin{pmatrix} \frac{\partial^2}{\partial x^2} f(\mathbf{a}) & \frac{\partial^2}{\partial x \partial y} f(\mathbf{a}) & \frac{\partial^2}{\partial x \partial z} f(\mathbf{a}) \\ \frac{\partial^2}{\partial y \partial x} f(\mathbf{a}) & \frac{\partial^2}{\partial y^2} f(\mathbf{a}) & \frac{\partial^2}{\partial y \partial z} f(\mathbf{a}) \\ \frac{\partial^2}{\partial z \partial x} f(\mathbf{a}) & \frac{\partial^2}{\partial z \partial y} f(\mathbf{a}) & \frac{\partial^2}{\partial z^2} f(\mathbf{a}) \end{pmatrix}. \quad (\text{C.55})$$

Note that the order of differentiation is not relevant since, for example, $\frac{\partial^2}{\partial x \partial y} = \frac{\partial^2}{\partial y \partial x}$, and therefore \mathbf{H}_f is always symmetric.

This can be easily generalized to the n -dimensional case, though things become considerably more involved for Taylor expansions of higher orders ($d > 2$).

C.3.3 Finding the Continuous Extremum of a Multi-Variable Discrete Function

In Sec. C.1.2 we described how the position of a local extremum can be determined by fitting a quadratic function to the neighboring samples of a *1D* function. This section shows how this technique can be extended to n -dimensional, scalar-valued functions $f: \mathbb{R}^n \mapsto \mathbb{R}$.

Without loss of generality we can assume that the Taylor expansion of the function $f(\mathbf{x})$ is carried out around the point $\mathbf{a} = \mathbf{0} = (0, \dots, 0)$, which clearly simplifies the remaining formulation. The Taylor approximation function (see Eqn. (C.52)) for this point can be written as

$$\tilde{f}(\mathbf{x}) = f(\mathbf{0}) + \nabla_f^\top(\mathbf{0}) \cdot \mathbf{x} + \frac{1}{2} \cdot \mathbf{x}^\top \cdot \mathbf{H}_f(\mathbf{0}) \cdot \mathbf{x}, \quad (\text{C.56})$$

with the gradient ∇_f and the Hessian matrix \mathbf{H}_f evaluated at position $\mathbf{0}$. The vector of the first derivative of this function is

$$\tilde{f}'(\mathbf{x}) = \nabla_f(\mathbf{0}) + \frac{1}{2} \cdot [(\mathbf{x}^\top \cdot \mathbf{H}_f(\mathbf{0}))^\top + \mathbf{H}_f(\mathbf{0}) \cdot \mathbf{x}]. \quad (\text{C.57})$$

Since $(\mathbf{x}^\top \cdot \mathbf{H}_f)^\top = (\mathbf{H}_f^\top \cdot \mathbf{x})$ and because the Hessian matrix \mathbf{H}_f is symmetric (i.e., $\mathbf{H}_f = \mathbf{H}_f^\top$), this simplifies to

$$\tilde{f}'(\mathbf{x}) = \nabla_f(\mathbf{0}) + \frac{1}{2} \cdot (\mathbf{H}_f(\mathbf{0}) \cdot \mathbf{x} + \mathbf{H}_f(\mathbf{0}) \cdot \mathbf{x}) \quad (\text{C.58})$$

$$= \nabla_f(\mathbf{0}) + \mathbf{H}_f(\mathbf{0}) \cdot \mathbf{x}. \quad (\text{C.59})$$

A local maximum or minimum is found where all first derivatives \tilde{f}' are zero, so we need to solve

$$\nabla_f(\mathbf{0}) + \mathbf{H}_f(\mathbf{0}) \cdot \check{\mathbf{x}} = \mathbf{0}, \quad (\text{C.60})$$

for the unknown position $\check{\mathbf{x}}$. By multiplying both sides with \mathbf{H}_f^{-1} (assuming that the inverse of $\mathbf{H}_f(\mathbf{0})$ exists), the solution is

$$\check{\mathbf{x}} = -\mathbf{H}_f^{-1}(\mathbf{0}) \cdot \nabla_f(\mathbf{0}), \quad (\text{C.61})$$

for the specific expansion point $\mathbf{a} = \mathbf{0}$ (Eqn. (C.63)). Analogously, for an arbitrary expansion point \mathbf{a} , the extremum position is

$$\check{\mathbf{x}} = \mathbf{a} - \mathbf{H}_f^{-1}(\mathbf{a}) \cdot \nabla_f(\mathbf{a}). \quad (\text{C.62})$$

Note that the inverse Hessian matrix \mathbf{H}_f^{-1} is again symmetric.

The estimated extremal *value* of the approximation function \tilde{f} is found by replacing \mathbf{x} in Eqn. (C.56) with the extremal position $\check{\mathbf{x}}$ (calculated in Eqn. (C.61)) as

$$\begin{aligned} \tilde{f}_{\text{extrm}} &= \tilde{f}(\check{\mathbf{x}}) = f(\mathbf{0}) + \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}} + \frac{1}{2} \cdot \check{\mathbf{x}}^\top \cdot \mathbf{H}_f(\mathbf{0}) \cdot \check{\mathbf{x}} \\ &= f(\mathbf{0}) + \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}} + \frac{1}{2} \cdot \check{\mathbf{x}}^\top \cdot \mathbf{H}_f(\mathbf{0}) \cdot (-\mathbf{H}_f^{-1}(\mathbf{0})) \cdot \nabla_f(\mathbf{0}) \\ &= f(\mathbf{0}) + \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}} - \frac{1}{2} \cdot \check{\mathbf{x}}^\top \cdot \mathbf{I} \cdot \nabla_f(\mathbf{0}) \quad (\text{C.63}) \\ &= f(\mathbf{0}) + \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}} - \frac{1}{2} \cdot \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}} \\ &= f(\mathbf{0}) + \frac{1}{2} \cdot \nabla_f^\top(\mathbf{0}) \cdot \check{\mathbf{x}}, \end{aligned}$$

again for the expansion point $\mathbf{a} = \mathbf{0}$.

$$\tilde{f}_{\text{extrm}} = \tilde{f}(\check{\mathbf{x}}) = f(\mathbf{a}) + \frac{1}{2} \cdot \nabla_f^\top(\mathbf{a}) \cdot (\check{\mathbf{x}} - \mathbf{a}). \quad (\text{C.64})$$

Note that \tilde{f}_{extrm} may be a local minimum or maximum, but could also be a *saddle point* where the first derivatives of the function are zero as well.

Local extrema in 2D

The aforementioned scheme can be applied to n -dimensional functions. In the special case of a 2D function $f: \mathbb{R}^2 \mapsto \mathbb{R}$ (e.g., a 2D image), the gradient vector and the Hessian matrix for the given expansion point $\mathbf{a} = (x_a, y_a)^\top$ can be noted as

$$\nabla_f(\mathbf{a}) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} \quad \text{and} \quad \mathbf{H}_f(\mathbf{a}) = \begin{pmatrix} H_{00} & H_{01} \\ H_{01} & H_{11} \end{pmatrix}, \quad (\text{C.65})$$

for a given expansion point $\mathbf{a} = (x_a, y_a)^\top$. In this case, the inverse of the Hessian matrix is

$$\mathbf{H}_f^{-1} = \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} -H_{11} & H_{01} \\ H_{01} & -H_{00} \end{pmatrix} \quad (\text{C.66})$$

and the resulting *position* of the extremal point is (see Eqn. (C.62))

$$\check{\mathbf{x}} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} -H_{11} & H_{01} \\ H_{01} & -H_{00} \end{pmatrix} \cdot \begin{pmatrix} d_x \\ d_y \end{pmatrix} \quad (\text{C.67})$$

$$= \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} H_{01} \cdot d_y - H_{11} \cdot d_x \\ H_{01} \cdot d_x - H_{00} \cdot d_y \end{pmatrix}. \quad (\text{C.68})$$

The extremal position is only defined if the denominator in Eqn. (C.68), $H_{01}^2 - H_{00} \cdot H_{11}$ (equivalent to the determinant of \mathbf{H}_f), is non-zero, indicating that the Hessian matrix \mathbf{H}_f is non-singular and thus has an inverse. The associated *value* of \tilde{f} at the estimated extremal position $\check{\mathbf{x}} = (\check{x}, \check{y})^\top$ can be now calculated using Eqn. (C.64) as

$$\begin{aligned} \tilde{f}(\check{x}, \check{y}) &= f(x_a, y_a) + \frac{1}{2} \cdot (d_x, d_y) \cdot \begin{pmatrix} \check{x} - x_a \\ \check{y} - y_a \end{pmatrix} \\ &= f(x_a, y_a) + \frac{d_x \cdot (\check{x} - x_a) + d_y \cdot (\check{y} - y_a)}{2}. \end{aligned} \quad (\text{C.69})$$

Numeric 2D example

The following example shows how a local extremum can be found in a discrete 2D image with sub-pixel accuracy using a second-order Taylor approximation. Assume we are given a grayscale image $I: \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{R}$ with the sample values

$$\begin{matrix} & u_a-1 & u_a & u_a+1 \\ v_a-1 & 8 & 11 & 7 \\ v_a & 15 & 16 & 9 \\ v_a+1 & 14 & 12 & 10 \end{matrix} \quad (\text{C.70})$$

in the 3×3 neighborhood of position $\mathbf{a} = (u_a, v_a)^\top$. Obviously, the discrete center value $f(\mathbf{a}) = 16$ is a local maximum but (as we shall see) the maximum of the *continuous* approximation function is *not* at the center. The gradient vector ∇_I and the Hessian Matrix \mathbf{H}_I at the expansion point \mathbf{a} are calculated from local finite differences (see Sec. C.3.1) as

$$\nabla_I(\mathbf{a}) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} = 0.5 \cdot \begin{pmatrix} 9-15 \\ 12-11 \end{pmatrix} = \begin{pmatrix} -3 \\ 0.5 \end{pmatrix} \quad \text{and} \quad (\text{C.71})$$

$$\begin{aligned} \mathbf{H}_I(\mathbf{a}) &= \begin{pmatrix} H_{11} & H_{12} \\ H_{12} & H_{22} \end{pmatrix} = \begin{pmatrix} 9-2 \cdot 16+15 & 0.25 \cdot (8-14-7+10) \\ 0.25 \cdot (8-14-7+10) & 11-2 \cdot 16+12 \end{pmatrix} \\ &= \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix}, \end{aligned} \quad (\text{C.72})$$

respectively. The resulting second-order Taylor expansion about the point \mathbf{a} is the continuous function (see Eqn. (C.52))

$$\begin{aligned} \tilde{f}(\mathbf{x}) &= f(\mathbf{a}) + \nabla_I^\top(\mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) + \frac{1}{2} \cdot (\mathbf{x} - \mathbf{a})^\top \cdot \mathbf{H}_I(\mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) \\ &= 16 + (-3, 0.5) \cdot \begin{pmatrix} x-u_a \\ y-v_a \end{pmatrix} \\ &\quad + \frac{1}{2} \cdot (x-u_a, y-v_a) \cdot \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix} \cdot \begin{pmatrix} x-u_a \\ y-v_a \end{pmatrix}. \end{aligned} \quad (\text{C.73})$$

We use the inverse of the 2×2 Hessian matrix at position \mathbf{a} (see Eqn. (C.66)),

$$\mathbf{H}_I^{-1}(\mathbf{a}) = \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix}^{-1} = \begin{pmatrix} -0.125984 & 0.010499 \\ 0.010499 & -0.111986 \end{pmatrix}, \quad (\text{C.74})$$

to calculate the *position* of the local extremum $\check{\mathbf{x}}$ (see Eqn. (C.68)) as

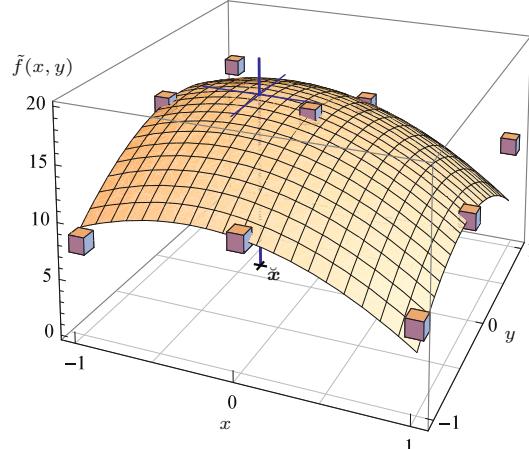
$$\begin{aligned} \check{\mathbf{x}} &= \mathbf{a} - \mathbf{H}_I^{-1}(\mathbf{a}) \cdot \nabla_I(\mathbf{a}) \\ &= \begin{pmatrix} u_a \\ v_a \end{pmatrix} - \begin{pmatrix} -0.125984 & 0.010499 \\ 0.010499 & -0.111986 \end{pmatrix} \cdot \begin{pmatrix} -3 \\ 0.5 \end{pmatrix} = \begin{pmatrix} u_a - 0.3832 \\ v_a + 0.0875 \end{pmatrix}. \end{aligned} \quad (\text{C.75})$$

Finally, the extremal *value* (see Eqn. (C.64)) is found as

$$\begin{aligned} \tilde{f}(\check{\mathbf{x}}) &= f(\mathbf{a}) + \frac{1}{2} \cdot \nabla_f^\top(\mathbf{a}) \cdot (\check{\mathbf{x}} - \mathbf{a}) \\ &= 16 + \frac{1}{2} \cdot (-3, 0.5) \cdot \begin{pmatrix} u_a - 0.3832 - u_a \\ v_a + 0.0875 - v_a \end{pmatrix} \\ &= 16 + \frac{1}{2} \cdot (3 \cdot 0.3832 + 0.5 \cdot 0.0875) = 16.5967. \end{aligned} \quad (\text{C.76})$$

Figure (C.3) illustrates the aforementioned example, with the expansion point set to $\mathbf{a} = (u_a, v_a)^\top = (0, 0)^\top$.

Fig. C.3
 Continuous Taylor approximation of a discrete 2D image function for determining the local extremum position with sub-pixel accuracy. The cubes represent the discrete image samples in 3×3 neighborhood around the reference coordinate $(0, 0)$, which is a local maximum of the discrete image function (see Eqn. (C.70) for the concrete values). The parabolic surface shows the continuous approximation $\tilde{f}(x, y)$ obtained by second-order Taylor expansion about the center position $\mathbf{a} = (0, 0)$. The vertical line marks the position of the local maximum $\tilde{f}(\check{\mathbf{x}}) = 16.5967$ at $\check{\mathbf{x}} = (-0.3832, 0.0875)$.



Local extrema in 3D

In the case of a three-variable, scalar function $f: \mathbb{R}^3 \mapsto \mathbb{R}$, with a given expansion point $\mathbf{a} = (x_a, y_a, z_a)^\top$ and

$$\nabla_f(\mathbf{a}) = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \quad \text{and} \quad \mathbf{H}_f(\mathbf{a}) = \begin{pmatrix} H_{00} & H_{01} & H_{02} \\ H_{01} & H_{11} & H_{12} \\ H_{02} & H_{12} & H_{22} \end{pmatrix} \quad (\text{C.77})$$

being the gradient vector and the Hessian matrix of f at point \mathbf{a} , respectively, the estimated extremal *position* is

$$\check{\mathbf{x}} = (\check{x}, \check{y}, \check{z})^\top = \mathbf{a} - \mathbf{H}_f^{-1}(\mathbf{a}) \cdot \nabla_f(\mathbf{a}) \quad (\text{C.78})$$

$$= \begin{pmatrix} x_a \\ y_a \\ z_a \end{pmatrix} - \frac{1}{H_{02}^2 \cdot H_{11} + H_{01}^2 \cdot H_{22} + H_{00} \cdot H_{12}^2 - H_{00} \cdot H_{11} \cdot H_{22} - 2 \cdot H_{01} \cdot H_{02} \cdot H_{12}} \\ \cdot \begin{pmatrix} H_{12}^2 - H_{11} \cdot H_{22} & H_{01} \cdot H_{22} - H_{02} \cdot H_{12} & H_{02} \cdot H_{11} - H_{01} \cdot H_{12} \\ H_{01} \cdot H_{22} - H_{02} \cdot H_{12} & H_{02}^2 - H_{00} \cdot H_{22} & H_{00} \cdot H_{12} - H_{01} \cdot H_{02} \\ H_{02} \cdot H_{11} - H_{01} \cdot H_{12} & H_{00} \cdot H_{12} - H_{01} \cdot H_{02} & H_{01}^2 - H_{00} \cdot H_{11} \end{pmatrix} \cdot \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

Note that the inverse of the 3×3 Hessian matrix \mathbf{H}_f^{-1} is again symmetric and can be calculated in closed form (as shown in Eqn. (C.78)).⁴

Again using Eqn. (C.64), the estimated extremal value at position $\check{\mathbf{x}} = (\check{x}, \check{y}, \check{z})^\top$ is found as

$$\tilde{f}(\check{\mathbf{x}}) = f(\mathbf{a}) + \frac{1}{2} \cdot \nabla_f^\top(\mathbf{a}) \cdot (\check{\mathbf{x}} - \mathbf{a}) \quad (\text{C.79})$$

$$= f(\mathbf{a}) + \frac{d_x \cdot (\check{x} - x_a) + d_y \cdot (\check{y} - y_a) + d_z \cdot (\check{z} - z_a)}{2}. \quad (\text{C.80})$$

⁴ Nevertheless, the use of standard numerical methods is recommended.

Appendix D

Statistical Prerequisites

This part summarizes some essential statistical concepts for vector-valued data, intended as a supplement particularly to Chapters 11 and 17.

D.1 Mean, Variance, and Covariance

For the following definitions we assume a sequence $X = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{n-1})$ of n vector-valued, m -dimensional measurements, with “samples”

$$\mathbf{x}_i = (x_{i,0}, x_{i,1}, \dots, x_{i,m-1})^\top \in \mathbb{R}^m. \quad (\text{D.1})$$

D.1.1 Mean

The n -dimensional *sample mean vector* is defined as

$$\boldsymbol{\mu}(X) = (\mu_0, \mu_1, \dots, \mu_{m-1})^\top \quad (\text{D.2})$$

$$= \frac{1}{n} \cdot (\mathbf{x}_0 + \mathbf{x}_1 + \dots + \mathbf{x}_{n-1}) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \mathbf{x}_i. \quad (\text{D.3})$$

Geometrically speaking, the vector $\boldsymbol{\mu}(X)$ corresponds to the *centroid* of the sample vectors \mathbf{x}_i in m -dimensional space. Each scalar element μ_p is the mean of the associated component (also called *variate* or *dimension*) p over all n samples, that is

$$\mu_p = \frac{1}{n} \cdot \sum_{i=0}^{n-1} x_{i,p}, \quad (\text{D.4})$$

for $p = 0, \dots, m-1$.

D.1.2 Variance and Covariance

The *covariance* quantifies the strength of interaction between a pair of components p, q in the sample X , defined as

$$\sigma_{p,q}(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p) \cdot (x_{i,q} - \mu_q). \quad (\text{D.5})$$

For efficient calculation, this expression can be rewritten in the form

$$\sigma_{p,q}(X) = \frac{1}{n} \cdot \underbrace{\left[\sum_{i=0}^{n-1} (x_{i,p} \cdot x_{i,q}) \right]}_{S_{p,q}(X)} - \frac{1}{n} \cdot \underbrace{\left(\sum_{i=0}^{n-1} x_{i,p} \right)}_{S_p(X)} \cdot \underbrace{\left(\sum_{i=0}^{n-1} x_{i,q} \right)}_{S_q(X)}, \quad (\text{D.6})$$

which does not require the explicit calculation of μ_p and μ_q . In the special case of $p = q$, we get

$$\sigma_{p,p}(X) = \sigma_p^2(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p)^2 \quad (\text{D.7})$$

$$= \frac{1}{n} \cdot \left[\sum_{i=0}^{n-1} x_{i,p}^2 - \frac{1}{n} \cdot \left(\sum_{i=0}^{n-1} x_{i,p} \right)^2 \right], \quad (\text{D.8})$$

which is the *variance within* the component p . This corresponds to the ordinary (one-dimensional) variance $\sigma_p^2(X)$ of the n scalar sample values $x_{0,p}, x_{1,p}, \dots, x_{n-1,p}$ (see also Sec. 3.7.1).

D.1.3 Biased vs. Unbiased Variance

If the variance (or covariance) of some population is estimated from a small set of random samples, the results obtained by the formulation given in the previous section are known to be statistically *biased*.¹ The most common form of correcting for this bias is to use the factor $1/(n-1)$ instead of $1/n$ in the variance calculations. For example, Eqn. (D.5) would change to

$$\check{\sigma}_{p,q}(X) = \frac{1}{n-1} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p) \cdot (x_{i,q} - \mu_q) \quad (\text{D.9})$$

to yield an *unbiased* sample variance. In the following (and throughout the text), we ignore the bias issue and consistently use the factor $1/n$ for all variance calculations. Note, however, that many software packages² use the bias-corrected factor $1/(n-1)$ by default and thus may return different results (which can be easily scaled for comparison).

D.2 The Covariance Matrix

The *covariance matrix* Σ for the m -dimensional sample X is a square matrix of size $m \times m$ that is composed of the covariance values $\sigma_{p,q}$ for all pairs (p, q) of components, that is,

¹ Note that the estimation of the mean by the *sample mean* (Eqn. (D.3)) is not affected by this bias problem.

² For example, *Apache Commons Math*, *Matlab*, *Mathematica*.

$$\Sigma(X) = \begin{pmatrix} \sigma_{0,0} & \sigma_{0,1} & \cdots & \sigma_{0,m-1} \\ \sigma_{1,0} & \sigma_{1,1} & \cdots & \sigma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m-1,0} & \sigma_{m-1,1} & \cdots & \sigma_{m-1,m-1} \end{pmatrix} \quad (\text{D.10})$$

$$= \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} & \cdots & \sigma_{0,m-1} \\ \sigma_{1,0} & \sigma_1^2 & \cdots & \sigma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m-1,0} & \sigma_{m-1,1} & \cdots & \sigma_{m-1}^2 \end{pmatrix}. \quad (\text{D.11})$$

Note that any diagonal element of $\Sigma(X)$ is the ordinary (scalar) variance $\sigma_p^2(X)$ (see Eqn. (D.7)), for $p = 0, \dots, m - 1$, which can never be negative. All other entries of a covariance matrix may be positive or negative in general. Since $\sigma_{p,q} = \sigma_{q,p}$, a covariance matrix is always symmetric, with up to $(m^2 + m)/2$ unique elements. Thus, any covariance matrix has the important property of being *positive semidefinite*, which implies that all its eigenvalues (see Sec. B.4) are positive (i.e., non-negative). The covariance matrix can also be written in the form

$$\Sigma(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \underbrace{[\mathbf{x}_i - \boldsymbol{\mu}(X)] \cdot [\mathbf{x}_i - \boldsymbol{\mu}(X)]^\top}_{= [\mathbf{x}_i - \boldsymbol{\mu}(X)] \otimes [\mathbf{x}_i - \boldsymbol{\mu}(X)]}, \quad (\text{D.12})$$

where \otimes denotes the outer (vector) product.

The *trace* (sum of the diagonal elements) of the covariance matrix,

$$\sigma_{\text{total}}(X) = \text{trace}(\Sigma(X)), \quad (\text{D.13})$$

is called the *total variance* of the multivariate sample. Alternatively, the (Frobenius) *norm* of the covariance matrix $\Sigma(X)$, defined as

$$\|\Sigma(X)\|_2 = \left(\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \sigma_{i,j}^2 \right)^{1/2}, \quad (\text{D.14})$$

can be used to quantify the overall variance in the sample data.

D.2.1 Example

Assume that the sample X consists of the following set of four 3D vectors (i.e., $m = 3$ and $n = 4$)

$$\mathbf{x}_0 = \begin{pmatrix} 75 \\ 37 \\ 12 \end{pmatrix}, \quad \mathbf{x}_1 = \begin{pmatrix} 41 \\ 27 \\ 20 \end{pmatrix}, \quad \mathbf{x}_2 = \begin{pmatrix} 93 \\ 81 \\ 11 \end{pmatrix}, \quad \mathbf{x}_3 = \begin{pmatrix} 12 \\ 48 \\ 52 \end{pmatrix},$$

with each $\mathbf{x}_i = (x_{i,R}, x_{i,G}, x_{i,B})^\top$ representing a particular RGB color. The resulting *sample mean vector* (see Eqn. (D.3)) is

$$\boldsymbol{\mu}(X) = \begin{pmatrix} \mu_R \\ \mu_G \\ \mu_B \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} 75 + 41 + 93 + 12 \\ 37 + 27 + 81 + 48 \\ 12 + 20 + 11 + 52 \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} 221 \\ 193 \\ 95 \end{pmatrix} = \begin{pmatrix} 55.25 \\ 48.25 \\ 23.75 \end{pmatrix},$$

and the associated *covariance matrix* (Eqn. (D.11)) is

$$\Sigma(X) = \begin{pmatrix} 972.188 & 331.938 & -470.438 \\ 331.938 & 412.688 & -53.188 \\ -470.438 & -53.188 & 278.188 \end{pmatrix}.$$

As predicted, this matrix is symmetric and all diagonal elements are non-negative. Note that *no* sample bias-correction (see Sec. D.1.3) was used in this example. The *total variance* (Eqn. (D.13)) of the sample set is

$$\sigma_{\text{total}}(X) = \text{trace}(\Sigma(X)) = 972.188 + 412.688 + 278.188 \approx 1663.06,$$

and the *Froebenius norm* of the covariance matrix (see Eqn. (D.14)) is $\|\Sigma(X)\|_2 \approx 1364.36$.

D.2.2 Practical Calculation

The calculation of covariance matrices is implemented in almost any software package for statistical analysis or linear algebra. For example, with the *Apache Commons Math* library this could be accomplished as follows:

```
import org.apache.commons.math3.stat.correlation.Covariance;
...
double[][] X;           // X[i] is the i-th sample vector
Covariance cov = new Covariance(X, false); // no bias correction
RealMatrix S = cov.getCovarianceMatrix();
...
```

D.3 Mahalanobis Distance

The Mahalanobis distance³ [157] is used to measure distances in multi-dimensional distributions. Unlike the Euclidean distance it takes into account the amount of scatter in the distribution and the correlation between features. In particular, the Mahalanobis distance can be used to measure distances in distributions, where the individual components substantially differ in scale. Depending on their scale, a few components (or even a single component) may dominate the ordinary (Euclidean) distance outcome and the “smaller” components have no influence whatsoever.

D.3.1 Definition

Given a distribution of m -dimensional samples $X = (\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$, with $\mathbf{x}_k \in \mathbb{R}^m$, the Mahalanobis distance between two samples \mathbf{x}_a , \mathbf{x}_b is defined as

$$d_M(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_M = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T \cdot \boldsymbol{\Sigma}^{-1} \cdot (\mathbf{x}_a - \mathbf{x}_b)}, \quad (\text{D.15})$$

where $\boldsymbol{\Sigma}$ is the $m \times m$ covariance matrix of the distribution X , as described in Sec. D.2.⁴

³ http://en.wikipedia.org/wiki/Mahalanobis_distance.

⁴ Note that the expression under the root in Eqn. (D.15) is the (dot) product of a row vector and a column vector, that is, the result is a non-negative scalar value.

The Mahalanobis distance normalizes each feature component to *zero mean* and *unit variance*. This makes the distance calculation independent of the scale of the individual components, that is, all components are “treated fairly” even if their range is many orders of magnitude different. In other words, no component can dominate the others even if its magnitude is disproportionately large.

D.3 MAHALANOBIS DISTANCE

D.3.2 Relation to the Euclidean Distance

Recall that the Euclidean distance between two points $\mathbf{x}_a, \mathbf{x}_b$ in \mathbb{R}^m is equivalent to the (L2) norm of the difference vector $\mathbf{x}_a - \mathbf{x}_b$, which can be written in the form

$$d_E(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_2 = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^\top \cdot (\mathbf{x}_a - \mathbf{x}_b)}. \quad (\text{D.16})$$

Note the structural similarity with the definition of the Mahalanobis distance in Eqn. (D.15), the only difference being the missing matrix Σ^{-1} . This becomes even clearer if we analogously insert the identity matrix \mathbf{I} into Eqn. (D.16), that is,

$$d_E(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_2 = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^\top \cdot \mathbf{I} \cdot (\mathbf{x}_a - \mathbf{x}_b)}, \quad (\text{D.17})$$

which obviously does not change the outcome. The purpose of Σ^{-1} in Eqn. (D.15) is to map the difference vectors (and thus the involved vectors $\mathbf{x}_a, \mathbf{x}_b$) into a transformed (scaled and rotated) space, where the actual distance measurement is performed. In contrast, with the Euclidean distance, all components contribute equally to the distance measure, without any scaling or other transformation.

D.3.3 Numerical Aspects

For calculating the Mahalanobis distance (Eqn. (D.15)) the *inverse* of the covariance matrix (Sec. D.2) is needed. By definition, a covariance matrix Σ is symmetric and its diagonal values are non-negative. Similarly (at least in theory), its inverse Σ^{-1} should also be symmetric with non-negative diagonal values. This is necessary to ensure that the quantities under the square root in Eqn. (D.15) are always positive.

Unfortunately, Σ is often ill-conditioned because of diagonal values that are very small or even zero. In this case, Σ is not positive-definite (as it should be), that is, one or more of its eigenvalues are negative, the inversion becomes numerically unstable and the resulting Σ^{-1} is non-symmetric. A simple remedy to this problem is to add a small quantity to the diagonal of the original covariance matrix Σ , that is,

$$\tilde{\Sigma} = \Sigma + \epsilon \cdot \mathbf{I}, \quad (\text{D.18})$$

to enforce positive definiteness, and to use $\tilde{\Sigma}^{-1}$ in Eqn. (D.15).

A possible alternative is to calculate the *Eigen decomposition*⁵ of Σ in the form

⁵ See <http://mathworld.wolfram.com/EigenDecomposition.html> and the class `EigenDecomposition` in the *Apache Commons Math* library.

$$\Sigma = \mathbf{V} \cdot \Lambda \cdot \mathbf{V}^\top \quad (\text{D.19})$$

where Λ is a diagonal matrix containing the eigenvalues of Σ (which may be zero or negative). From this we create a modified diagonal matrix $\tilde{\Lambda}$ by substituting all non-positive eigenvalues with a small positive quantity ϵ , that is,

$$\tilde{\Lambda}_{i,i} = \min(\Lambda_{i,i}, \epsilon). \quad (\text{D.20})$$

(typically $\epsilon \approx 10^{-6}$) and finally calculate the modified covariance matrix as

$$\tilde{\Sigma} = \mathbf{V} \cdot \tilde{\Lambda} \cdot \mathbf{V}^\top, \quad (\text{D.21})$$

which should be positive definite. The (symmetric) inverse $\tilde{\Sigma}^{-1}$ is then used in Eqn. (D.15).

D.3.4 Pre-Mapping Data for Efficient Mahalanobis Matching

Assume that we have a large set of sample vectors (“data base”) $X = (\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$ which shall be frequently queried for the instance most similar (i.e., closest) to a given search sample \mathbf{x}_s . Assuming that the search through X is performed linearly, we would need to calculate $d_M(\mathbf{x}_s, \mathbf{x}_i)$ —using Eqn. (D.15)—for all elements of \mathbf{x}_i in X .

One way to accelerate the matching is to perform the transformation defined by Σ^{-1} to the entire data set only once, such that the Euclidean norm alone can be used for the distance calculation. For the sake of simplicity we write

$$d_M^2(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_M^2 = \|\mathbf{y}\|_M^2 \quad (\text{D.22})$$

with the difference vector $\mathbf{y} = \mathbf{x}_a - \mathbf{x}_b$, such that Eqn. (D.15) becomes

$$\|\mathbf{y}\|_M^2 = \mathbf{y}^\top \cdot \Sigma^{-1} \cdot \mathbf{y}. \quad (\text{D.23})$$

The goal is to find a transformation \mathbf{U} such that we can calculate the Mahalanobis distance from the transformed vectors directly as

$$\hat{\mathbf{y}} = \mathbf{U} \cdot \mathbf{y}, \quad (\text{D.24})$$

by using the ordinary *Euclidean* norm $\|\cdot\|_2$ instead, that is, in the form

$$\|\mathbf{y}\|_M^2 = \|\hat{\mathbf{y}}\|_2^2 = \hat{\mathbf{y}}^\top \cdot \hat{\mathbf{y}} \quad (\text{D.25})$$

$$= (\mathbf{U} \cdot \mathbf{y})^\top \cdot (\mathbf{U} \cdot \mathbf{y}) = (\mathbf{y}^\top \cdot \mathbf{U}^\top) \cdot (\mathbf{U} \cdot \mathbf{y}) \quad (\text{D.26})$$

$$= \mathbf{y}^\top \cdot \mathbf{U}^\top \cdot \mathbf{U} \cdot \mathbf{y} = \mathbf{y}^\top \cdot \Sigma^{-1} \cdot \mathbf{y}. \quad (\text{D.27})$$

While we do not know the matrix \mathbf{U} yet, we see from Eqn. (D.27) that it must satisfy

$$\mathbf{U}^\top \cdot \mathbf{U} = \Sigma^{-1}. \quad (\text{D.28})$$

Fortunately, since Σ^{-1} is symmetric and positive definite, such a decomposition of Σ^{-1} always exists.

The standard method for calculating \mathbf{U} in Eqn. (D.28) is by the Cholesky decomposition,⁶ which can factorize any symmetric, positive definite matrix \mathbf{A} in the form

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T \quad \text{or} \quad \mathbf{A} = \mathbf{U}^T \cdot \mathbf{U}, \quad (\text{D.29})$$

where \mathbf{L} is a *lower-triangular* matrix or, alternatively, \mathbf{U} is an *upper-triangular* matrix (the second variant is the one we need).⁷ Since the transformation of the difference vectors $\mathbf{y} \rightarrow \mathbf{U} \cdot \mathbf{y}$ is a linear operation, the result is the same if we apply the transformation individually to the original vectors, that is,

$$\hat{\mathbf{y}} = \mathbf{U} \cdot \mathbf{y} = \mathbf{U} \cdot (\mathbf{x}_a - \mathbf{x}_b) = \mathbf{U} \cdot \mathbf{x}_a - \mathbf{U} \cdot \mathbf{x}_b. \quad (\text{D.30})$$

This means that, given the transformation \mathbf{U} , we can obtain the Mahalanobis distance between two points $\mathbf{x}_a, \mathbf{x}_b$ (as defined in Eqn. (D.15)) by simply calculating the Euclidean distance in the form

$$d_M(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{U} \cdot (\mathbf{x}_a - \mathbf{x}_b)\|_2 = \|\mathbf{U} \cdot \mathbf{x}_a - \mathbf{U} \cdot \mathbf{x}_b\|_2. \quad (\text{D.31})$$

In summary, this suggests the following solution to a large-database Mahalanobis matching problem:

1. Calculate the covariance matrix Σ for the original dataset $X = (\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$.
2. Condition Σ , such that it is positive definite (see Sec. D.3.3).
3. Find the matrix \mathbf{U} , such that $\mathbf{U}^T \cdot \mathbf{U} = \Sigma^{-1}$ (by Cholesky decomposition of Σ^{-1}).
4. Transform all samples of the original data set $X = (\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$ to $\hat{X} = (\hat{\mathbf{x}}_0, \dots, \hat{\mathbf{x}}_{n-1})$, with $\hat{\mathbf{x}}_k = \mathbf{U} \cdot \mathbf{x}_k$. This now becomes the actual “database”.
5. Apply the same transformation to the search sample \mathbf{x}_s , that is, calculate $\hat{\mathbf{x}}_s = \mathbf{U} \cdot \mathbf{x}_s$.
6. Find the index l of the best-matching element in X (in terms of the Mahalanobis distance) by calculating the *Euclidean* (!) distance between the transformed vectors, that is

$$l = \underset{0 \leq k < n}{\operatorname{argmin}} \|\hat{\mathbf{x}}_s - \hat{\mathbf{x}}_k\|^2. \quad (\text{D.32})$$

Since the matching is now performed with the ordinary Euclidean distance and the Mahalanobis calculation is not required during the search, the savings should be substantial. Also, this opens an easy path to the use of advanced, tree-based matching techniques, such as the common k -nearest neighbor methods.

⁶ See <http://mathworld.wolfram.com/CholeskyDecomposition.html>.

⁷ The Cholesky decomposition (CD) requires that the supplied matrix \mathbf{A} is symmetric and positive definite, otherwise the decomposition will fail. In fact, the CD itself is commonly used to test if a given matrix is positive definite. It is implemented by class `CholeskyDecomposition` of the *Apache Commons Math* library.

D.4 The Gaussian Distribution

The Gaussian distribution plays a major role in decision theory, pattern recognition, and statistics in general, because of its convenient analytical properties. A continuous, scalar quantity X is said to be subject to a Gaussian distribution, if the probability of observing a particular value x is

$$p(X=x) = p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \quad (\text{D.33})$$

The Gaussian distribution is completely defined by its mean μ and variance σ^2 . The Gaussian distribution, also called a “normal” distribution, is commonly denoted in the form

$$p(x) \sim \mathcal{N}(X | \mu, \sigma^2) \quad \text{or} \quad X \sim \mathcal{N}(\mu, \sigma^2), \quad (\text{D.34})$$

saying that “ X is normally distributed with parameters μ and σ^2 .” As required for any valid probability distribution,

$$\mathcal{N}(X | \mu, \sigma^2) > 0 \quad \text{and} \quad \int_{-\infty}^{\infty} \mathcal{N}(X | \mu, \sigma^2) dx = 1. \quad (\text{D.35})$$

Thus the area under the probability distribution curve is always one, that is, $\mathcal{N}()$ is normalized. The Gaussian function in Eqn. (D.33) has its maximum height (called “mode”) at position $x = \mu$, where its value is

$$p(x=\mu) = \frac{1}{\sqrt{2\pi\sigma^2}}. \quad (\text{D.36})$$

If a random variable X is normally distributed with mean μ and variance σ^2 , then the result of a linear mapping of the kind $X' = aX + b$ is again a random variable that is normally distributed, with parameters $\bar{\mu} = a \cdot \mu + b$ and $\bar{\sigma}^2 = a^2 \cdot \sigma^2$:

$$X \sim \mathcal{N}(\mu, \sigma^2) \Rightarrow a \cdot X + b \sim \mathcal{N}(a \cdot \mu + b, a^2 \cdot \sigma^2), \quad (\text{D.37})$$

for $a, b \in \mathbb{R}$.

Moreover, if X_1, X_2 are statistically *independent*, normally distributed random variables with means μ_1, μ_2 and variances σ_1^2, σ_2^2 , respectively, then a linear combination of the form $a_1 X_1 + a_2 X_2$ is again normally distributed with $\mu_{12} = a_1 \cdot \mu_1 + a_2 \cdot \mu_2$ and $\sigma_{12}^2 = a_1^2 \cdot \sigma_1^2 + a_2^2 \cdot \sigma_2^2$, that is,

$$(a_1 X_1 + a_2 X_2) \sim \mathcal{N}(a_1 \cdot \mu_1 + a_2 \cdot \mu_2, a_1^2 \cdot \sigma_1^2 + a_2^2 \cdot \sigma_2^2). \quad (\text{D.38})$$

D.4.1 Maximum Likelihood Estimation

The probability density function $p(x)$ of a statistical distribution tells us how probable it is to observe the result x for some fixed distribution parameters, such as μ and σ , in case of a normal distribution. If these parameters are *unknown* and need to be estimated,⁸ it is interesting to ask the reverse question:

⁸ As required, for example, for “minimum error thresholding” in Chapter 11, Sec. 11.1.6.

How likely are particular parameter values for a given set of empirical observations (assuming a certain type of distribution)?

This is (in a casual sense) what the term “likelihood” stands for. In particular, a distribution’s *likelihood function* quantifies the probability that a given (fixed) set of observations was generated by some varying distribution parameters.

Note that the probability of observing the outcome x from the normal distribution,

$$p(x) = p(x | \mu, \sigma^2), \quad (\text{D.39})$$

is really a *conditional* probability, stating how probable it is to observe the value x from a given normal distribution with known parameters μ and σ^2 . Conversely, a likelihood function for the normal distribution could be viewed as a conditional function

$$L(\mu, \sigma^2 | x), \quad (\text{D.40})$$

which quantifies the likelihood of (μ, σ^2) being the correct distribution parameters for a given observation x . The maximum likelihood method tries to find optimal parameters by *maximizing* the value of a distribution’s likelihood function L .

If we draw two independent⁹ samples x_a, x_b that are subjected to the same distribution, their *joint probability* (i.e., the probability of x_a and x_b occurring together in the sample) is the product of their individual probabilities, that is,

$$p(x_a \wedge x_b) = p(x_a) \cdot p(x_b). \quad (\text{D.41})$$

In general, if we are given a vector of m independent observations $X = (x_1, x_2, \dots, x_m)$ from the same distribution, the probability of observing exactly this set of values is

$$\begin{aligned} p(X) &= p(x_0 \wedge x_1 \wedge \dots \wedge x_{m-1}) \\ &= p(x_0) \cdot p(x_1) \cdot \dots \cdot p(x_{m-1}) = \prod_{i=0}^{m-1} p(x_i). \end{aligned} \quad (\text{D.42})$$

Thus, if the sample X originates from a normal distribution \mathcal{N} , a suitable likelihood function is

$$L(\mu, \sigma^2 | X) = p(X | \mu, \sigma^2) \quad (\text{D.43})$$

$$= \prod_{i=0}^{m-1} \mathcal{N}(x_i | \mu, \sigma^2) = \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x_i - \mu)^2}{2\sigma^2}}. \quad (\text{D.44})$$

The parameters $(\hat{\mu}, \hat{\sigma}^2)$, for which $L(\mu, \sigma^2 | X)$ is a maximum, are called the maximum-likelihood estimate for X .

Note that it is not necessary for a likelihood function to be a proper (i.e., normalized) probability distribution, since it is only necessary to calculate whether a particular set of distribution parameters

D.4 THE GAUSSIAN DISTRIBUTION

⁹ Although this assumption is often violated, independence is important to keep statistical problems simple and tractable. In particular, the values of adjacent image pixels are usually not independent.

is more probable than another. Thus the likelihood function L may be any monotonic function of the corresponding probability p in Eqn. (D.43), in particular its *logarithm*, which is commonly used to avoid multiplying small values.

D.4.2 Gaussian Mixtures

In practice, probabilistic models are often too complex to be described by a single Gaussian (or other standard) distribution. Without losing the mathematical convenience of Gaussian models, highly complex distributions can be modeled as combinations of multiple Gaussian distributions with different parameters. Such a Gaussian *mixture model* is a linear superposition of K Gaussian distributions of the form

$$p(x) = \sum_{j=0}^{K-1} \pi_j \cdot \mathcal{N}(x | \mu_j, \sigma_j^2), \quad (\text{D.45})$$

where the weights (“mixing coefficients”) π_j express the probability that an event x was generated by the j^{th} component (with $\sum_{j=0}^{K-1} \pi_j = 1$).¹⁰ The interpretation of this mixture model is, that there are K independent Gaussian “components” (each with its parameters μ_j , σ_j) that contribute to a common stream of events x_i . If a particular value x is observed, it is assumed to be the result of exactly *one* of the K components, but the identity of that component is unknown.

Assume, as a special case, that a probability distribution $p(x)$ is the superposition (mixture) of *two* Gaussian distributions, that is,

$$p(x) = \pi_a \cdot \mathcal{N}(x | \mu_a, \sigma_a^2) + \pi_b \cdot \mathcal{N}(x | \mu_b, \sigma_b^2). \quad (\text{D.46})$$

Any observed value x is assumed to be generated by either the first component (with μ_a, σ_a^2 and prior probability π_a) or the second component (with μ_b, σ_b^2 and prior probability π_b). These parameters as well as the prior probabilities are unknown but can be estimated by maximizing the likelihood function L . Note that, in general, the unknown parameters cannot be calculated in closed form but only with numerical methods. For further details and solution techniques see [24, 64, 228], for example.

D.4.3 Creating Gaussian Noise

Synthetic Gaussian noise is often used for testing in image processing, particularly for assessing the quality of smoothing filters. While the generation of pseudo-random values that follow a Gaussian distribution is not a trivial task in general,¹¹ it is readily implemented in Java by the standard class `Random`. For example, the Java method `addGaussianNoise()` in Prog. D.1 adds Gaussian noise with zero mean ($\mu = 0$) and standard deviation `sigma` (σ) to a grayscale image `I` of type `FloatProcessor` (ImageJ). The random values produced

¹⁰ The weight π_j is also called the *prior* probability of the component j .

¹¹ Typically the so-called *polar method* is used for generating Gaussian random values [138, Sec. 3.4.1].

by successive calls to the method `nextGaussian()` in line 10 follow a Gaussian distribution $\mathcal{N}(0, 1)$, with mean $\mu = 0$ and variance $\sigma^2 = 1$. As implied by Eqn. (D.37),

$$X \sim \mathcal{N}(0, 1) \Rightarrow a + s \cdot X \sim \mathcal{N}(a, s^2), \quad (\text{D.47})$$

and thus scaling the results from `nextGaussian()` by s and additive shifting by a makes the resulting random variable `noise` normally distributed with $\mathcal{N}(a, s^2)$.

```
1 import java.util.Random;
2
3 void addGaussianNoise (FloatProcessor I, double sigma) {
4     int w = I.getWidth();
5     int h = I.getHeight();
6     Random rnd = new Random();
7     for (int v = 0; v < h; v++) {
8         for (int u = 0; u < w; u++) {
9             float val = I.getf(u, v);
10            float noise = (float) (rnd.nextGaussian() * sigma);
11            I.setf(u, v, val + noise);
12        }
13    }
14 }
```

D.4 THE GAUSSIAN DISTRIBUTION

Prog. D.1

Java method for adding Gaussian noise to an image of type `FloatProcessor`.

Appendix E

Gaussian Filters

This part supplements the material presented in Ch. 25 (SIFT).

E.1 Cascading Gaussian Filters

To compute a Gaussian scale space efficiently (as used in the SIFT method, for example), the scale layers are usually not obtained directly from the input image by smoothing with Gaussians of increasing size. Instead, each layer can be calculated recursively from the previous layer by filtering with relatively small Gaussians. Thus, the entire scale space is implemented as a concatenation or “cascade” of smaller Gaussian filters.¹

If Gaussian filters of sizes σ_1, σ_2 are applied successively to the same image, the resulting smoothing effect is identical to using a single larger Gaussian filter H_σ^G , that is,

$$(I * H_{\sigma_1}^G) * H_{\sigma_2}^G = I * (H_{\sigma_1}^G * H_{\sigma_2}^G) = I * H_\sigma^G, \quad (\text{E.1})$$

with $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$ being the size of the resulting combined Gaussian filter H_σ^G [129, Sec. 4.5.4]. Put in other words, the *variances* (squares of the σ values) of successive Gaussian filters add up, that is,

$$\sigma^2 = \sigma_1^2 + \sigma_2^2. \quad (\text{E.2})$$

In the special case of the *same* Gaussian filter being applied twice ($\sigma_1 = \sigma_2$), the effective width of the combined filter is $\sigma = \sqrt{2} \cdot \sigma_1$.

E.2 Gaussian Filters and Scale Space

In a Gaussian scale space, the scale corresponding to each level is proportional to the width (σ) of the Gaussian filter required to derive this level from the original (completely unsmoothed) image. Given an image that is already pre-smoothed by a Gaussian filter of width

¹ See Chapter 25, Sec. 25.1.1 for details.

σ_1 and should be smoothed to some target scale $\sigma_2 > \sigma_1$, the required width of the additional Gaussian filter is

$$\sigma_d = \sqrt{\sigma_2^2 - \sigma_1^2}. \quad (\text{E.3})$$

Usually the neighboring layers of the scale space differ by a constant scale factor (κ) and the transformation from one scale level to another can be accomplished by successively applying Gaussian filters. Despite the constant scale factor, however, the width of the required filters is *not* constant but depends on the image's initial scale. In particular, if we want to transform an image with scale σ_0 by a factor κ to a new scale $\kappa \cdot \sigma_0$, then (from Eqn. (E.2)) for σ_d the relation

$$(\kappa \cdot \sigma_0)^2 = \sigma_0^2 + \sigma_d^2 \quad (\text{E.4})$$

must hold. Thus, the width σ_d of the required Gaussian smoothing filter is

$$\sigma_d = \sigma_0 \cdot \sqrt{\kappa^2 - 1}. \quad (\text{E.5})$$

For example, doubling the scale ($\kappa = 2$) of an image that is pre-smoothed with σ_0 requires a Gaussian filter of width $\sigma_d = \sigma_0 \cdot (2^2 - 1)^{1/2} = \sigma_0 \cdot \sqrt{3} \approx \sigma_0 \cdot 1.732$.

E.3 Effects of Gaussian Filtering in the Frequency Domain

For the 1D Gaussian function

$$g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{x^2}{2\sigma^2}} \quad (\text{E.6})$$

the continuous Fourier transform² $\mathcal{F}(g_\sigma)$ is

$$G_\sigma(\omega) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{\omega^2\sigma^2}{2}}. \quad (\text{E.7})$$

Doubling the width (σ) of a Gaussian filter corresponds to cutting the bandwidth by half. If σ is doubled, the Fourier transform becomes

$$G_{2\sigma}(\omega) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{\omega^2(2\sigma)^2}{2}} = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{4\omega^2\sigma^2}{2}} \quad (\text{E.8})$$

$$= \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{(2\omega)^2\sigma^2}{2}} = G_\sigma(2\omega) \quad (\text{E.9})$$

and, in general, when scaling the filter by a factor k ,

$$G_{k\sigma}(\omega) = G_\sigma(k\omega). \quad (\text{E.10})$$

That is, if σ is *increased* (or the kernel widened) by a factor k , the corresponding Fourier transform gets *contracted* by the same factor. In terms of linear filtering this means that widening the kernel by some factor k decimates the resulting signal bandwidth by $\frac{1}{k}$.

² See also Chapter 18, Sec. 18.1.

E.4 LoG-Approximation by the DoG

E.4 LOG-APPROXIMATION BY THE DoG

The 2D LoG kernel (see Ch. 25, Sec. 25.1.1),

$$L_\sigma(x, y) = (\nabla^2 g_\sigma)(x, y) = \frac{1}{\pi\sigma^4} \left(\frac{x^2 + y^2 - 2\sigma^2}{2\sigma^2} \right) \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}, \quad (\text{E.11})$$

has a (negative) peak at the origin with the associated function value

$$L_\sigma(0, 0) = -\frac{1}{\pi\sigma^4}. \quad (\text{E.12})$$

Thus, the *scale normalized* LoG kernel, defined in Eqn. (25.10) as

$$\hat{L}_\sigma(x, y) = \sigma^2 \cdot L_\sigma(x, y), \quad (\text{E.13})$$

has the peak value

$$\hat{L}_\sigma(0, 0) = -\frac{1}{\pi\sigma^2} \quad (\text{E.14})$$

at the origin. In comparison, for a given scale factor κ , the unscaled DoG function

$$\begin{aligned} \text{DoG}_{\sigma, \kappa}(x, y) &= G_{\kappa\sigma}(x, y) - G_\sigma(x, y) \\ &= \frac{1}{2\pi\kappa^2\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\kappa^2\sigma^2}} - \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}, \end{aligned} \quad (\text{E.15})$$

has a peak value

$$\text{DoG}_{\sigma, \kappa}(0, 0) = -\frac{\kappa^2 - 1}{2\pi\kappa^2\sigma^2}. \quad (\text{E.16})$$

By scaling the DoG function by some factor λ to match the LoG's center peak value, such that $L_\sigma(0, 0) = \lambda \cdot \text{DoG}_{\sigma, \kappa}(0, 0)$, the original LoG (Eqn. (E.11)) is approximated by the DoG in the form

$$L_\sigma(x, y) \approx \frac{2\kappa^2}{\sigma^2(\kappa^2 - 1)} \cdot \text{DoG}_{\sigma, \kappa}(x, y). \quad (\text{E.17})$$

Similarly, the scale-normalized LoG (Eqn. (E.13)) is approximated by the DoG as³

$$\hat{L}_\sigma(x, y) \approx \frac{2\kappa^2}{\kappa^2 - 1} \cdot \text{DoG}_{\sigma, \kappa}(x, y). \quad (\text{E.18})$$

Since the factor in Eqn. (E.18) depends on κ only, the DoG approximation is (for a constant size ratio κ) implicitly proportional to the scale normalized LoG for any scale σ .

³ A different formulation, $\hat{L}_\sigma(x, y) \approx \frac{1}{\kappa-1} \cdot \text{DoG}_{\sigma, \kappa}(x, y)$, is given in [153], which is the same as Eqn. (E.18) for $\kappa \rightarrow 1$, but not for $\kappa > 1$. The essence is that the leading factor is constant and independent of σ , and can thus be ignored when comparing the magnitude of the filter responses at varying scales.

Appendix F

Java Notes

As a text for undergraduate engineering curricula, this book assumes basic programming skills in a procedural language, such as Java, C#, or C. The examples in the main text should be easy to understand with the help of an introductory book on Java or one of the many online tutorials. Experience shows, however, that difficulties with some basic Java concepts pertain and often cause complications, even at higher levels. The following sections address some of these typical problem spots.

F.1 Arithmetic

Java is a “strongly typed” programming language, which means in particular that any variable has a fixed type that cannot be altered dynamically. Also, the result of an expression is determined by the types of the involved operands and *not* (in the case of an assignment) by the type of the “receiving” variable.

F.1.1 Integer Division

Division involving integer operands is a frequent cause of errors. If the variables `a` and `b` are both of type `int`, then the expression `a / b` is evaluated according to the rules of integer division. The result—the number of times `b` is contained in `a`—is again of type `int`. For example, after the Java statements

```
int a = 2;  
int b = 5;  
double c = a / b; // resulting value of c is zero!
```

the value of `c` is *not* 0.4 but 0.0 because the expression `a / b` on the right yields the `int`-value 0, which is then automatically converted to the `double` value 0.0.

If we wanted to evaluate `a / b` as a *floating-point* operation (as most pocket calculators do), at least one of the involved operands

must be converted to a floating-point value, such as by an explicit type cast, for example,

```
double c = (double) a / b; // value of c is 0.4
```

or alternatively

```
double c = a / (double) b; // value of c is 0.4
```

Example

Assume, for example, that we want to scale any pixel value a of an image such that the maximum pixel value a_{\max} is mapped to 255 (see Ch. 4). In mathematical notation, the scaling of the pixel values is simply expressed as

$$c \leftarrow \frac{a_i}{a_{\max}} \cdot 255$$

and it may be tempting to convert this 1:1 into Java code, such as

```
int a_max = ip.getMaxValue();
for ... {
    int a = ip.getPixel(u, v);
    int c = (a / a_max) * 255; // ← problem!
    ip.putPixel(u, v, c);
}
...
```

As we can easily predict, the resulting image will be all black (zero values), except those pixels whose value was a_{\max} originally (they are set to 255). The reason is again that the division a / a_{\max} has two operands of type `int`, and the result is thus zero whenever the denominator (`a_max`) is greater than the numerator (`a`).

Of course, the entire operation could be performed in the floating-point domain by converting one of the operands (as we have shown), but this is not even necessary in this case. Instead, we may simply swap the order of operations and start with the multiplication:

```
int c = a * 255 / a_max;
```

Why does this work now? The subexpression $a * 255$ is evaluated first,¹ generating large intermediate values that pose no problem for the subsequent (integer) division. Nevertheless, *rounding* should always be considered to obtain more accurate results when computing fractions of integers (see Sec. F.1.5).

F.1.2 Modulus Operator

The result of the modulus operator $a \bmod b$ (used in several places in the main text) is defined [92, p. 82] as the remainder of the “floored” division a/b ,

$$a \bmod b \equiv \begin{cases} a & \text{for } b = 0, \\ a - b \cdot \lfloor a/b \rfloor & \text{otherwise,} \end{cases} \quad (\text{F.1})$$

¹ In Java, expressions at the same level are always evaluated in left-to-right order, and therefore no parentheses are required in this example (though they would do no harm either).

for $a, b \in \mathbb{R}$. This type of operator or library method was not available in the standard Java API until recently.² The following Java method implements the mod operation according to the definition in Eqn. (F.1):³

```
int Mod(int a, int b) {
    if (b == 0)
        return a;
    if (a * b >= 0)
        return a - b * (a / b);
    else
        return a - b * (a / b - 1);
}
```

F.1 ARITHMETIC

Note that the *remainder* operator `%`, defined as

$$a \% b \equiv a - b \cdot \text{truncate}(a/b), \quad \text{for } b \neq 0, \quad (\text{F.2})$$

is often used in this context, but yields the same results only for *positive* operands $a \geq 0$ and $b > 0$. For example,

13 mod 4 = 1	13 % 4 = 1
13 mod -4 = -3	13 % -4 = 1
-13 mod 4 = 3	-13 % 4 = -1
-13 mod -4 = -1	-13 % -4 = -1

vs.

F.1.3 Unsigned Byte Data

Most grayscale and indexed images in Java and ImageJ are composed of pixels of type `byte`, and the same holds for the individual components of most color images. A single byte consists of eight bits and can thus represent $2^8 = 256$ different bit patterns or values, usually mapped to the numeric range $0, \dots, 255$. Unfortunately, Java (unlike C and C++) does *not* provide a suitable “unsigned” 8-bit data type. The primitive Java type `byte` is “signed”, using one of its eight bits for the \pm sign, and is intended to hold values in the range $-128, \dots, +127$.

Java’s `byte` data can still be used to represent the values 0 to 255, but conversions must take place to perform proper arithmetic computations. For example, after execution of the statements

```
int a = 200;
byte b = (byte) p;
```

the variables `a` (32-bit `int`) and `b` (8-bit `byte`) contain the binary patterns

```
a = 000000000000000000000000000000011001000
b = 11001000
```

Interpreted as a (signed) `byte` value, with the leftmost bit⁴ as the sign bit, the variable `b` has the decimal value -56 . Thus after the statement

² Starting with Java version 1.8 the mod operation (as defined in Eqn. (F.1)) is implemented by the standard method `Math.floorMod(a, b)`.

³ The definition in Eqn. (F.1) is not restricted to integer operands.

⁴ Java uses the standard “2s-complement” representation, where a sign bit = 1 stands for a negative value.

```
int a1 = b; // a1 == -56
```

the value of the new `int` variable `a1` is `-56`! To (ab-)use signed `byte` data as *unsigned* data, we can circumvent Java's standard conversion mechanism by disguising the content of `b` as a logic (i.e., nonarithmetic) *bit pattern*; for example, by

```
int a2 = (0xff & b); // a2 == 200
```

where `0xff` (in hexadecimal notation) is an `int` value with the binary bit pattern `00000000000000000000000011111111` and `&` is the bitwise AND operator. Now the variable `a2` contains the right integer value (200) and we thus have a way to use Java's (signed) `byte` data type for storing *unsigned* values. Within ImageJ, access to pixel data is routinely implemented in this way, which is considerably faster than using the convenience methods `getPixel()` and `putPixel()`.

F.1.4 Mathematical Functions in Class Math

Java provides most standard mathematical functions as static methods in class `Math`, as listed in Table F.1. The `Math` class is part of the `java.lang` package and thus requires no explicit import to be used. Most `Math` methods accept arguments of type `double` and also return values of type `double`. As a simple example, a typical use of the cosine function $y = \cos(x)$ is

```
double x;
double y = Math.cos(x);
```

Similarly, the `Math` class defines some common numerical constants as static variables; for example, the value of π could be obtained by

```
double pi = Math.PI;
```

Table F.1
Mathematical methods and constants defined by Java's `Math` class.

<code>double abs(double a)</code> <code>int abs(int a)</code> <code>float abs(float a)</code> <code>long abs(long a)</code> <code>double ceil(double a)</code> <code>double floor(double a)</code> <code>int floorMod(int a, int b)</code> <code>long floorMod(long a, long b)</code> <code>double rint(double a)</code> <code>long round(double a)</code> <code>int round(float a)</code>	<code>double max(double a, double b)</code> <code>float max(float a, float b)</code> <code>int max(int a, int b)</code> <code>long max(long a, long b)</code> <code>double min(double a, double b)</code> <code>float min(float a, float b)</code> <code>int min(int a, int b)</code> <code>long min(long a, long b)</code> <code>double random()</code>
<code>double toDegrees(double rad)</code> <code>double sin(double a)</code> <code>double cos(double a)</code> <code>double tan(double a)</code> <code>double atan2(double y, double x)</code>	<code>double toRadians(double deg)</code> <code>double asin(double a)</code> <code>double acos(double a)</code> <code>double atan(double a)</code>
<code>double log(double a)</code> <code>double sqrt(double a)</code>	<code>double exp(double a)</code> <code>double pow(double a, double b)</code>
<code>double E</code>	<code>double PI</code>

Java's `Math` class (confusingly) offers three different methods for rounding floating-point values:

```
double rint(double x)
long   round(double x)
int    round(float x)
```

For example, a `double` value `x` can be rounded to `int` in any of the following ways:

```
double x; int k;
k = (int) Math.rint(x);
k = (int) Math.round(x);
k = Math.round((float) x);
```

If the operand `x` is known to be positive (as is typically the case with pixel values) rounding can be accomplished without using any method calls by

```
k = (int) (x + 0.5); // only if x >= 0
```

In this case, the expression $(x + 0.5)$ is first computed as a floating-point (`double`) value, which is then truncated (toward zero) by the explicit `(int)` typecast.

F.1.6 Inverse Tangent Function

The inverse tangent function $\varphi = \tan^{-1}(a)$ or $\varphi = \arctan(a)$ is used in several places in the main text. This function is implemented by the method `atan(double a)` in Java's `Math` class (Table F.1). The return value of `atan()` is in the range $[-\frac{\pi}{2}, \dots, \frac{\pi}{2}]$ and thus restricted to only two of the four quadrants. Without any additional constraints, the resulting angle is ambiguous. In many practical situations, however, a is given as the ratio of two catheti ($\Delta x, \Delta y$) of a right-angled triangle in the form

$$\varphi = \arctan\left(\frac{y}{x}\right), \quad (\text{F.3})$$

for which we introduced the two-parameter function

$$\varphi = \text{ArcTan}(x, y) \quad (\text{F.4})$$

in the main text. The function `ArcTan(x, y)` is implemented by the standard method `atan2(dy, dx)` in Java's `Math` class (note the reversed parameters though) and returns an unambiguous angle φ in the range $[-\pi, \dots, \pi]$; that is, in any of the four quadrants of the unit circle.⁵ Also, the `atan2()` method returns a useful value even if both arguments are zero.

⁵ The function `atan2(dy, dx)` is available in most current programming languages, including Java, C, and C++.

F.1.7 Classes `Float` and `Double`

The representation of floating-point numbers in Java follows the IEEE standard, and thus the types `float` and `double` include the values

<code>Float.MIN_VALUE,</code>	<code>Double.MIN_VALUE,</code>
<code>Float.MAX_VALUE,</code>	<code>Double.MAX_VALUE,</code>
<code>Float.POSITIVE_INFINITY,</code>	<code>Double.POSITIVE_INFINITY,</code>
<code>Float.NEGATIVE_INFINITY,</code>	<code>Double.NEGATIVE_INFINITY,</code>
<code>Float.NaN,</code>	<code>Double.NaN.</code>

These values are defined as constants in the corresponding wrapper classes `Float` and `Double`, respectively. If any `INFINITY` or `NaN`⁶ value occurs in the course of a computation (e.g., as the result of dividing by zero),⁷ Java continues without raising an error, so incorrect values may ripple through a whole chain of calculations, making the actual bugs difficult to locate.

F.1.8 Testing Floating-Point Values Against Zero

Comparing floating-point values or testing them for zero is a non-trivial issue and a frequent cause of errors. In particular, one should *never* write

```
if (x == 0.0) {...} ← problem!
```

if `x` is a floating-point variable. This is often needed, for example, to make sure that it is safe to divide another quantity by `x`. The aforementioned test, however, is not sufficient since `x` may be non-zero but still too small as a divisor.

A much better alternative is to test if `x` is “close” to zero, that is, within some small positive/negative (*epsilon*) interval. While the proper choice of this interval depends on the specific situation, the following settings are usually sufficient for safe operation:⁸

```
static final float EPSILON_FLOAT = 1e-7f;
static final double EPSILON_DOUBLE = 2e-16;

float x;
double y;

if (Math.abs(x) < EPSILON_FLOAT) {
    ... // x is practically zero
}

if (Math.abs(y) < EPSILON_DOUBLE) {
    ... // y is practically zero
}
```

⁶ `NaN` stands for “not a number”.

⁷ In Java, this only holds for floating-point operations, whereas integer division by zero always causes an *exception*.

⁸ These settings account for the limited *machine accuracy* (ϵ_m) of the IEEE 754 standard types `float` ($\epsilon_m \approx 1.19 \cdot 10^{-7}$) and `double` ($\epsilon_m \approx 2.22 \cdot 10^{-16}$) [190, Ch. 1, Sec. 1.1.2].

F.2.1 Creating Arrays

Unlike in most traditional programming languages (such as FORTRAN or C), arrays in Java can be created *dynamically*, meaning that the size of an array can be specified at runtime using the value of some variable or arithmetic expression. For example:

```
int N = 20;
int[] A = new int[N];
int[] B = new int[N * N];
```

Once allocated, however, the size of any Java array is fixed and cannot be subsequently altered.⁹ Note that Java arrays may be of length zero!

After its definition, an array variable can be assigned any other compatible array or the constant value `null`, for example,¹⁰

```
A = B;      // A now references the data in B
B = null;
```

With the assignment `A = B`, the array initially referenced by `A` becomes unaccessible and thus turns into *garbage*. In contrast to C and C++, where unnecessary storage needs to be deallocated explicitly, this is taken care of in Java by its built-in “garbage collector”. It is also convenient that newly created arrays of numerical element types (`int`, `float`, `double`, etc.) are automatically initialized to zero.

F.2.2 Array Size

Since an array may be created dynamically, it is important that its actual size can be determined at runtime. This is done by accessing the `length` attribute¹¹

```
int k = A.length; // number of elements in A
```

The size is a property of the array itself and can therefore be obtained inside any method from array arguments passed to it. Thus (unlike in C, for example) it is not necessary to pass the size of an array as a separate function argument.

If an array has more than one dimension, the size (`length`) along every dimension must be queried separately (see Sec. F.2.4). Also arrays are not necessarily rectangular; for example, the rows of a 2D array may have different lengths (including zero).

F.2.3 Accessing Array Elements

In Java, the index of the first array element is always 0 and the index of the last element is $N - 1$ for an array with a total of N elements. To iterate through a 1D array `A` of arbitrary size, one would typically use a construct like

⁹ For additional flexibility, Java provides a number of universal container classes (e.g., the classes `Set` and `List`) for a wide range of applications.

¹⁰ This is not possible if the array variable was defined with the `final` attribute.

¹¹ Notice that the `length` attribute of an array is not a method!

```
for (int i = 0; i < A.length; i++) {
    // do something with A[i]
}
```

Alternatively, if only the array *values* are relevant and the array *index* (*i*) is not needed, one could use the following (even simpler) loop construct:

```
for (int a : A) {
    // do something with array values a
}
```

In both cases, the Java compiler can generate very efficient runtime code, since the source code makes obvious that the `for` loop does not access any elements outside the array limits and thus no explicit boundary checking is needed at execution time. This fact is very important for implementing efficient image processing programs in Java.

Images in Java and ImageJ are usually stored as 1D arrays (accessible through the `ImageProcessor` method `getPixels()` in ImageJ), with pixels arranged in row-first order.¹² Statistical calculations and most point operations can thus be efficiently implemented by directly accessing the underlying 1D array. For example, the `run` method of the contrast enhancement plugin in Prog. 4.1 (see Chapter 4, p. 58) could also be implemented in the following manner:

```
public void run(ImageProcessor ip) {
    // ip is assumed to be of type ByteProcessor
    byte[] pixels = (byte[]) ip.getPixels();
    for (int i = 0; i < pixels.length; i++) {
        int a = 0xFF & pixels[i];           // direct read operation
        int b = (int) (a * 1.5 + 0.5);
        if (b > 255)
            b = 255;
        pixels[i] = (byte) (0xFF & b);     // direct write operation
    }
}
```

F.2.4 2D Arrays

Multidimensional arrays are a frequent source of confusion. In Java, all arrays are 1D in principle, and multi-dimensional arrays are implemented as 1D arrays of arrays etc. (see Fig. F.1). If, for example, the 3×3 matrix

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (\text{F.5})$$

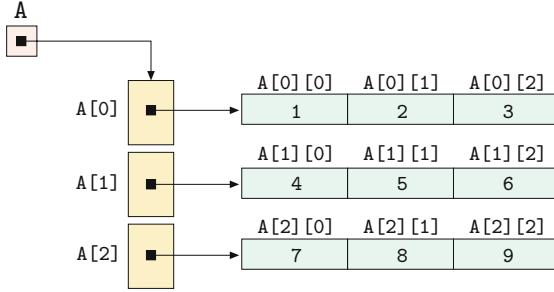
is defined as a 2D `int` array,

```
int[][] A = {{1,2,3},
             {4,5,6},
             {7,8,9}};
```

¹² This means that horizontally adjacent image pixels are stored next to each other in computer memory.

Fig. F.1

Layout of elements of a 2D Java array (corresponding to Eqn. (F.5)). In Java, multidimensional arrays are generally implemented as *1D* arrays whose elements are again 1D arrays.



then A is actually a *1D* array with three elements, each of which is again a 1D array. The elements $A[0]$, $A[1]$ and $A[2]$ are of type `int[]` and correspond to the three rows of the matrix A (see Fig. F.1).

The usual assumption is that the array elements are arranged in *row-first* order, as illustrated in Fig. F.1. The first index thus corresponds to the *row* number r and the second index corresponds to the *column* number c , that is,

$$a_{r,c} \equiv A[r][c]. \quad (\text{F.6})$$

This conforms to the mathematical convention and makes the array definition in the code segment above look exactly the same as the original matrix in Eqn. (F.5). Note that in this scheme the first array index corresponds to the *vertical* coordinate and the second index to the *horizontal* coordinate.

However, if an array is used to specify the contents of an *image* $I(u,v)$ or a *filter kernel* $H(i,j)$, we usually assume that the first index (u or i , respectively) is associated with the horizontal x -coordinate and the second index (v bzw. j) with the vertical y -coordinate. For example, if we represent the filter kernel

$$H = \begin{bmatrix} h_{0,0} & h_{1,0} & h_{2,0} \\ h_{0,1} & h_{1,1} & h_{2,1} \\ h_{0,2} & h_{1,2} & h_{2,2} \end{bmatrix} = \begin{bmatrix} -1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

as a 2D Java array,

```
double[][] H = {{-1,-2, 0},
                {-2, 0, 2},
                { 0, 2, 1}};
```

then the row and column indexes must be *reversed* in order to access the correct elements. In this case we have the relation

$$h_{i,j} \equiv H[j][i], \quad (\text{F.7})$$

that is, the ordering of the indexes for array H is not the same as for the i/j coordinates of the filter kernel. In this case the *first* array index (j) corresponds to the *vertical* coordinate and the *second* index (i) to the *horizontal* coordinate. The advantage is that (as shown in the aforementioned code segment) the definition of the filter kernel

can be written in the usual matrix form¹³ (otherwise we would have to specify the transposed kernel matrix).

If a 2D array is merely used as an image container (whose contents are never defined in matrix form) any convention can be used for the ordering of the indexes. For example, the ImageJ method `getFloatArray()` of class `ImageProcessor`, when called in the form

```
float[][] I = ip.getFloatArray();
```

returns the image as a 2D array (`I`), whose indexes are arranged in the usual x/y order, that is,

$$I(x, y) \equiv I[x][y]. \quad (\text{F.8})$$

In this case, the image pixels are arranged in column-order, that is, *vertically* adjacent elements are stored next to each other in memory.

Size of multi-dimensional arrays

The size of a multi-dimensional array can be obtained by querying the size of its sub-arrays. For example, given the following 3D array with dimensions $P \times Q \times R$,

```
int A[][][] = new int[P][Q][R];
```

the size of `A` along its three dimensions is obtained by the statements

```
int p = A.length;           // = P
int q = A[0].length;        // = Q
int r = A[0][0].length;      // = R
```

This at least works for “rectangular” Java arrays, that is, multi-dimensional arrays with all sub-arrays at the same level having *identical* lengths, which is warranted by the array initialization in the aforementioned case. However, every 1D sub-array of `A` may be replaced by a suitable 1D array of *different* length,¹⁴ for example, by the statement

```
A[0][0] = new int[0];
```

To avoid “index-out-of-bounds” errors, the length of each sub-array should be determined dynamically. The following example shows a “bullet-proof” iteration over all elements of a 3D array `A` whose sub-arrays may have different lengths or may even be empty:

```
int A[][][];  
...  
for (int i = 0; i < A.length; i++) {  
    for (int j = 0; j < A[i].length; j++) {  
        for (int k = 0; k < A[i][j].length; k++) {  
            // safely access A[i][j][k]  
        }  
    }  
}
```

¹³ This scheme is used, for example, in the implementation of the 3×3 filter plugin in Prog. 5.2 (Chapter 5, p. 95).

¹⁴ Even if the array `A` was originally declared `final`, the structure and contents of its sub-arrays may be modified any time.

In Java, as mentioned earlier, we can create arrays dynamically; that is, the size of an array can be specified at runtime. This is convenient because we can adapt the size of the arrays to the given problem. For example, we could write

```
Corner[] corners = new Corner[n];
```

to create an array that can hold n objects of type `Corner` (as defined in Chapter 7, Sec. 7.3). Note that the new array `corners` is not filled with corners yet but initialized with `null` references, so the newly created array holds no objects at all. We can insert a `Corner` object into its first (or any other) cell, for example, by

```
corners[0] = new Corner(10, 20, 6789.0f);
```

F.2.6 Searching for Minimum and Maximum Values

Unfortunately, the standard Java API does not provide methods for retrieving the minimum and maximum values of a numeric array. Although these values are easily found by iterating over all elements of the sequence, care must be taken regarding the initialization.

For example, finding the extreme values of a sequence of `int`-values could be accomplished as follows:¹⁵

```
int[] A = ...
int minval = Integer.MAX_VALUE;
int maxval = Integer.MIN_VALUE;
for (int val : A) {
    minval = Math.min(minval, val);
    maxval = Math.max(maxval, val);
}
```

Note the use of the constants `MIN_VALUE` and `MAX_VALUE`, which are defined for any numeric Java type.

However, in the case of *floating-point* values, these are not the proper values for initialization.¹⁶ Instead, `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` should be used, as shown in the following code segment:

```
double[] B = ...
double minval = Double.POSITIVE_INFINITY;
double maxval = Double.NEGATIVE_INFINITY;
for (double val : B) {
    minval = Math.min(minval, val);
    maxval = Math.max(maxval, val);
}
```

¹⁵ Alternatively, one could initialize `minval` and `maxval` with the first array element `A[0]`.

¹⁶ Because `Double.MIN_VALUE` and `Float.MIN_VALUE` specify to the smallest *positive* values.

F.2.7 Sorting Arrays

Arrays can be sorted efficiently with the standard method

```
Arrays.sort(type[] arr)
```

in class `java.util.Arrays`, where `arr` can be any array of primitive `type` (`int`, `float`, etc.) or an array of objects. In the latter case, the array may not have `null` entries. Also, the class of every contained object must implement the `Comparable` interface, that is, provide a public method `compareTo()` that returns an `int` value of `-1`, `0`, or `1`, depending upon the intended ordering relation. For example, the class `Corner` defines the `compareTo()` method as follows:

```
public class Corner implements Comparable<Corner> {
    float x, y, q;
    ...
    public int compareTo(Corner other) {
        if (this.q > other.q) return -1;
        else if (this.q < other.q) return 1;
        else return 0;
    }
}
```

References

1. Adobe Systems. “Adobe RGB (1998) Color Space Specification” (2005). <http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf>.
2. M. AHMED AND R. WARD. A rotation invariant rule-based thinning algorithm for character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(12), 1672–1678 (2002).
3. L. ALVAREZ, P.-L. LIONS, AND J.-M. MOREL. Image selective smoothing and edge detection by nonlinear diffusion (II). *SIAM Journal on Numerical Analysis* **29**(3), 845–866 (1992).
4. Apache Software Foundation. “Commons Math: The Apache Commons Mathematics Library”. <http://commons.apache.org/math/index.html>.
5. K. ARBTER, W. E. SNYDER, H. BURKHARDT, AND G. HIRZINGER. Application of affine-invariant Fourier descriptors to recognition of 3-D objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12**(7), 640–647 (1990).
6. G. R. ARCE, J. BACCA, AND J. L. PAREDES. Nonlinear filtering for image analysis and enhancement. In A. BOVIK, editor, “Handbook of Image and Video Processing”, pp. 109–133. Academic Press, New York, second ed. (2005).
7. C. ARCELLI AND G. SANNITI DI BAJA. A one-pass two-operation process to detect the skeletal pixels on the 4-distance transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**(4), 411–414 (1989).
8. K. ARNOLD, J. GOSLING, AND D. HOLMES. “The Java Programming Language”. Prentice Hall, fifth ed. (2012).
9. S. ARYA, D. M. MOUNT, N. S. NETANYAHU, R. SILVERMAN, AND A. Y. WU. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM* **45**(6), 891–923 (1998).
10. J. ASTOLA, P. HAAVISTO, AND Y. NEUVO. Vector median filters. *Proceedings of the IEEE* **78**(4), 678–689 (1990).
11. J. BABAUD, A. P. WITKIN, M. BAUDIN, AND R. O. DUDA. Uniqueness of the Gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(1), 26–33 (1986).
12. W. BAILER. “Writing ImageJ Plugins—A Tutorial” (2003). <http://www.imagingbook.com>.
13. S. BAKER AND I. MATTHEWS. Lucas-Kanade 20 years on: A unifying framework: Part 1. Technical Report CMU-RI-TR-02-16, Robotics Institute, Carnegie Mellon University (2003).
14. S. BAKER AND I. MATTHEWS. Lucas-Kanade 20 years on: A unifying framework. *International Journal of Computer Vision* **56**(3), 221–255 (2004).
15. D. H. BALLARD AND C. M. BROWN. “Computer Vision”. Prentice Hall, Englewood Cliffs, NJ (1982).
16. D. BARASH. Fundamental relationship between bilateral filtering, adaptive smoothing, and the nonlinear diffusion equation. *IEEE*

- Transactions on Pattern Analysis and Machine Intelligence* **24**(6), 844–847 (2002).
17. C. B. BARBER, D. P. DOBKIN, AND H. HUHDANPAA. The quick-hull algorithm for convex hulls. *ACM Transactions on Mathematical Software* **22**(4), 469–483 (1996).
 18. M. BARNI. A fast algorithm for 1-norm vector median filtering. *IEEE Transactions on Image Processing* **6**(10), 1452–1455 (1997).
 19. H. G. BARROW, J. M. TENENBAUM, R. C. BOLLES, AND H. C. WOLF. Parametric correspondence and chamfer matching: two new techniques for image matching. In R. REDDY, editor, “Proceedings of the 5th International Joint Conference on Artificial Intelligence”, pp. 659–663, Cambridge, MA (1977). William Kaufmann, Los Altos, CA.
 20. H. BAY, A. ESS, T. TUYTELAARS, AND L. VAN GOOL. SURF: Speeded up robust features. *Computer Vision, Graphics, and Image Processing: Image Understanding* **110**(3), 346–359 (2008).
 21. J. S. BEIS AND D. G. LOWE. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In “Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR’97)”, pp. 1000–1006, Puerto Rico (June 1997).
 22. R. BENCINA AND M. KALTENBRUNNER. The design and evolution of fiducials for the reacTIVision system. In “Proceedings of the 3rd International Conference on Generative Systems in the Electronic Arts”, Melbourne (2005).
 23. J. BERNSEN. Dynamic thresholding of grey-level images. In “Proceedings of the International Conference on Pattern Recognition (ICPR)”, pp. 1251–1255, Paris (October 1986). IEEE Computer Society.
 24. C. M. BISHOP. “Pattern Recognition and Machine Learning”. Springer, New York (2006).
 25. R. E. BLAHUT. “Fast Algorithms for Digital Signal Processing”. Addison-Wesley, Reading, MA (1985).
 26. I. BLAYVAS, A. BRUCKSTEIN, AND R. KIMMEL. Efficient computation of adaptive threshold surfaces for image binarization. *Pattern Recognition* **39**(1), 89–101 (2006).
 27. J. BLINN. Consider the lowly 2×2 matrix. *IEEE Computer Graphics and Applications* **16**(2), 82–88 (1996).
 28. J. BLINN. “Jim Blinn’s Corner: Notation, Notation, Notation”. Morgan Kaufmann (2002).
 29. J. BLOCH. “Effective Java”. Addison-Wesley, second ed. (2008).
 30. G. BORGEFORS. Distance transformations in digital images. *Computer Vision, Graphics and Image Processing* **34**, 344–371 (1986).
 31. G. BORGEFORS. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **10**(6), 849–865 (1988).
 32. A. I. BORISENKO AND I. E. TARAPOV. “Vector and Tensor Analysis with Applications”. Dover Publications, New York (1979).
 33. J. E. BRESENHAM. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM* **20**(2), 100–106 (1977).
 34. E. O. BRIGHAM. “The Fast Fourier Transform and Its Applications”. Prentice Hall, Englewood Cliffs, NJ (1988).
 35. I. N. BRONSTEIN AND K. A. SEMENDJAJEW. “Handbook of Mathematics”. Springer-Verlag, Berlin, third ed. (2007).
 36. I. N. BRONSTEIN, K. A. SEMENDJAJEW, G. MUSIOL, AND H. MÜHLIG. “Taschenbuch der Mathematik”. Verlag Harri Deutsch, fifth ed. (2000).
 37. M. BROWN AND D. LOWE. Invariant features from interest point groups. In “Proceedings of the British Machine Vision Conference”, pp. 656–665 (2002).

38. H. BUNKE AND P. S.-P. WANG, editors. “Handbook of Character Recognition and Document Image Analysis”. World Scientific, Singapore (2000).
39. W. BURGER AND M. J. BURGE. “Digital Image Processing—An Algorithmic Introduction using Java”. Texts in Computer Science. Springer, New York (2008).
40. W. BURGER AND M. J. BURGE. “ImageJ Short Reference for Java Developers” (2008). <http://www.imagingbook.com>.
41. P. J. BURT AND E. H. ADELSON. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications* **31**(4), 532–540 (1983).
42. J. F. CANNY. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986).
43. K. R. CASTLEMAN. “Digital Image Processing”. Prentice Hall, Upper Saddle River, NJ (1995).
44. E. E. CATMULL AND R. ROM. A class of local interpolating splines. In R. E. BARNHILL AND R. F. RIESENFELD, editors, “Computer Aided Geometric Design”, pp. 317–326. Academic Press, New York (1974).
45. F. CATTÉ, P.-L. LIONS, J.-M. MOREL, AND T. COLL. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM Journal on Numerical Analysis* **29**(1), 182–193 (1992).
46. C. I. CHANG, Y. DU, J. WANG, S. M. GUO, AND P. D. THOUIN. Survey and comparative analysis of entropy and relative entropy thresholding techniques. *IEE Proceedings—Vision, Image and Signal Processing* **153**(6), 837–850 (2006).
47. F. CHANG, C. J. CHEN, AND C. J. LU. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision, Graphics, and Image Processing: Image Understanding* **93**(2), 206–220 (2004).
48. P. CHARBONNIER, L. BLANC-FERAUD, G. AUBERT, AND M. BARLAUD. Two deterministic half-quadratic regularization algorithms for computed imaging. In “Proceedings IEEE International Conference on Image Processing (ICIP-94)”, vol. 2, pp. 168–172, Austin (November 1994).
49. Y. CHEN AND G. LEEDHAM. Decompose algorithm for thresholding degraded historical document images. *IEE Proceedings—Vision, Image and Signal Processing* **152**(6), 702–714 (2005).
50. H. D. CHENG, X. H. JIANG, Y. SUN, AND J. WANG. Color image segmentation: advances and prospects. *Pattern Recognition* **34**(12), 2259–2281 (2001).
51. P. R. COHEN AND E. A. FEIGENBAUM. “The Handbook of Artificial Intelligence”. William Kaufmann, Los Altos, CA (1982).
52. B. COLL, J. L. LISANI, AND C. SBERT. Color images filtering by anisotropic diffusion. In “Proceedings of the IEEE International Conference on Systems, Signals, and Image Processing (IWSSIP)”, pp. 305–308, Chalkida, Greece (2005).
53. D. COMANICIU AND P. MEER. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(5), 603–619 (2002).
54. T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. “Introduction to Algorithms”. MIT Press, Cambridge, MA, second ed. (2001).
55. R. L. COSGRIFF. Identification of shape. Technical Report 820-11, Antenna Laboratory, Ohio State University, Department of Electrical Engineering, Columbus, Ohio (December 1960).

56. A. CRIMINISI, I. D. REID, AND A. ZISSERMAN. A plane measuring device. *Image and Vision Computing* **17**(8), 625–634 (1999).
57. T. R. CRIMMINS. A complete set of Fourier descriptors for two-dimensional shapes. *IEEE Transactions on Systems, Man, and Cybernetics* **12**(6), 848–855 (1982).
58. F. C. CROW. Summed-area tables for texture mapping. *SIGGRAPH Computer Graphics* **18**(3), 207–212 (1984).
59. A. CUMANI. Edge detection in multispectral images. *Computer Vision, Graphics and Image Processing* **53**(1), 40–51 (1991).
60. A. CUMANI. Efficient contour extraction in color images. In “Proceedings of the Third Asian Conference on Computer Vision”, ACCV, pp. 582–589, Hong Kong (January 1998). Springer.
61. L. S. DAVIS. A survey of edge detection techniques. *Computer Graphics and Image Processing* **4**, 248–270 (1975).
62. R. DERICHE. Using Canny’s criteria to derive a recursively implemented optimal edge detector. *International Journal of Computer Vision* **1**(2), 167–187 (1987).
63. S. DI ZENZO. A note on the gradient of a multi-image. *Computer Vision, Graphics and Image Processing* **33**(1), 116–125 (1986).
64. R. O. DUDA, P. E. HART, AND D. G. STORK. “Pattern Classification”. Wiley, New York (2001).
65. F. DURAND AND J. DORSEY. Fast bilateral filtering for the display of high-dynamic-range images. In “Proceedings of the 29th annual conference on Computer graphics and interactive techniques (SIGGRAPH’02)”, pp. 257–266, San Antonio, Texas (July 2002).
66. B. ECKEL. “Thinking in Java”. Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006). Earlier versions available online.
67. M. ELAD. On the origin of the bilateral filter and ways to improve it. *IEEE Transactions on Image Processing* **11**(10), 1141–1151 (2002).
68. A. FERREIRA AND S. UBEDA. Computing the medial axis transform in parallel with eight scan operations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**(3), 277–282 (1999).
69. N. I. FISHER. “Statistical Analysis of Circular Data”. Cambridge University Press (1995).
70. D. FLANAGAN. “Java in a Nutshell”. O’Reilly, Sebastopol, CA, fifth ed. (2005).
71. L. M. J. FLORACK, B. M. TER HAAR ROMENY, J. J. KOENDERINK, AND M. A. VIERGEVER. Scale and the differential structure of images. *Image and Vision Computing* **10**(6), 376–388 (1992).
72. J. FLUSSER. On the independence of rotation moment invariants. *Pattern Recognition* **33**(9), 1405–1410 (2000).
73. J. FLUSSER. Moment forms invariant to rotation and blur in arbitrary number of dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **25**(2), 234–246 (2003).
74. J. FLUSSER, B. ZITOVA, AND T. SUK. “Moments and Moment Invariants in Pattern Recognition”. John Wiley & Sons (2009).
75. J. D. FOLEY, A. VAN DAM, S. K. FEINER, AND J. F. HUGHES. “Computer Graphics: Principles and Practice”. Addison-Wesley, Reading, MA, second ed. (1996).
76. A. FORD AND A. ROBERTS. “Colour Space Conversions” (1998). <http://www.poynton.com/PDFs/coloureq.pdf>.
77. W. FÖRSTNER AND E. GÜLCH. A fast operator for detection and precise location of distinct points, corners and centres of circular features. In A. GRÜN AND H. BEYER, editors, “Proceedings, International Society for Photogrammetry and Remote Sensing Intercommission Conference on the Fast Processing of Photogrammetric Data”, pp. 281–305, Interlaken (June 1987).

-
78. D. A. FORSYTH AND J. PONCE. “Computer Vision—A Modern Approach”. Prentice Hall, Englewood Cliffs, NJ (2003).
79. H. FREEMAN. Computer processing of line drawing images. *ACM Computing Surveys* **6**(1), 57–97 (1974).
80. J. H. FRIEDMAN, J. L. BENTLEY, AND R. A. FINKEL. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software* **3**(3), 209–226 (1977).
81. D. L. FRITZSCHE. A systematic method for character recognition. Technical Report 1222-4, Antenna Laboratory, Ohio State University, Department of Electrical Engineering, Columbus, Ohio (November 1961).
82. M. GERVAUTZ AND W. PURGATHOFER. A simple method for color quantization: octree quantization. In A. GLASSNER, editor, “Graphics Gems I”, pp. 287–293. Academic Press, New York (1990).
83. T. GEVERS, A. GIJSENIJ, J. VAN DE WEIJER, AND J.-M. GEUSEBROEK. “Color in Computer Vision”. Wiley (2012).
84. T. GEVERS AND H. STOKMAN. Classifying color edges in video into shadow-geometry, highlight, or material transitions. *IEEE Transactions on Multimedia* **5**(2), 237–243 (2003).
85. T. GEVERS, J. VAN DE WEIJER, AND H. STOKMAN. Color feature detection. In R. LUKAC AND K. N. PLATANIOTIS, editors, “Color Image Processing: Methods and Applications”, pp. 203–226. CRC Press (2006).
86. C. A. GLASBEY. An analysis of histogram-based thresholding algorithms. *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing* **55**(6), 532–537 (1993).
87. A. S. GLASSNER. “Principles of Digital Image Synthesis”. Morgan Kaufmann Publishers, San Francisco (1995).
88. R. C. GONZALEZ AND R. E. WOODS. “Digital Image Processing”. Addison-Wesley, Reading, MA (1992).
89. R. C. GONZALEZ AND R. E. WOODS. “Digital Image Processing”. Pearson Prentice Hall, Upper Saddle River, NJ, third ed. (2008).
90. M. GRABNER, H. GRABNER, AND H. BISCHOF. Fast approximated SIFT. In “Proceedings of the 7th Asian Conference of Computer Vision”, pp. 918–927 (2006).
91. R. L. GRAHAM. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters* **1**, 132–133 (1972).
92. R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK. “Concrete Mathematics: A Foundation for Computer Science”. Addison-Wesley, Reading, MA, second ed. (1994).
93. G. H. GRANLUND. Fourier preprocessing for hand print character recognition. *IEEE Transactions on Computers* **21**(2), 195–201 (1972).
94. P. GREEN. Colorimetry and colour differences. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 3, pp. 40–77. Wiley, New York (2002).
95. F. GUICHARD, L. MOISAN, AND J.-M. MOREL. A review of P.D.E. models in image processing and image analysis. *J. Phys. IV France* **12**(1), 137–154 (2002).
96. W. W. HAGER. “Applied Numerical Linear Algebra”. Prentice Hall (1988).
97. E. L. HALL. “Computer Image Processing and Recognition”. Academic Press, New York (1979).
98. A. HANBURY. Circular statistics applied to colour images. In “Proceedings of the 8th Computer Vision Winter Workshop”, pp. 55–60, Valtice, Czech Republic (February 2003).

REFERENCES

99. J. C. HANCOCK. "An Introduction to the Principles of Communication Theory". McGraw-Hill (1961).
100. I. HANNAH, D. PATEL, AND R. DAVIES. The use of variance and entropic thresholding methods for image segmentation. *Pattern Recognition* **28**(4), 1135–1143 (1995).
101. W. W. HARMAN. "Principles of the Statistical Theory of Communication". McGraw-Hill (1963).
102. C. G. HARRIS AND M. STEPHENS. A combined corner and edge detector. In C. J. TAYLOR, editor, "4th Alvey Vision Conference", pp. 147–151, Manchester (1988).
103. R. HARTLEY AND A. ZISSERMAN. "Multiple View Geometry in Computer Vision". Cambridge University Press, 2 ed. (2013).
104. P. S. HECKBERT. Color image quantization for frame buffer display. *Computer Graphics* **16**(3), 297–307 (1982).
105. P. S. HECKBERT. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, Dept. of Electrical Engineering and Computer Science (1989).
106. R. HESS. An open-source SIFT library. In "Proceedings of the International Conference on Multimedia, MM'10", pp. 1493–1496, Firenze, Italy (October 2010).
107. J. HOLM, I. TASTL, L. HANLON, AND P. HUBEL. Color processing for digital photography. In P. GREEN AND L. MACDONALD, editors, "Colour Engineering", ch. 9, pp. 179–220. Wiley, New York (2002).
108. C. M. HOLT, A. STEWART, M. CLINT, AND R. H. PERROTT. An improved parallel thinning algorithm. *Communications of the ACM* **30**(2), 156–160 (1987).
109. V. HONG, H. PALUS, AND D. PAULUS. Edge preserving filters on color images. In "Proceedings Int'l Conf. on Computational Science, ICCS", pp. 34–40, Kraków, Poland (2004).
110. B. K. P. HORN. "Robot Vision". MIT-Press, Cambridge, MA (1982).
111. P. V. C. HOUGH. Method and means for recognizing complex patterns. US Patent 3,069,654 (1962).
112. M. K. HU. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory* **8**, 179–187 (1962).
113. A. HUERTAS AND G. MEDIONI. Detection of intensity changes with subpixel accuracy using Laplacian-Gaussian masks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(5), 651–664 (1986).
114. R. W. G. HUNT. "The Reproduction of Colour". Wiley, New York, sixth ed. (2004).
115. J. HUTCHINSON. Culture, communication, and an information age madonna. *IEEE Professional Communications Society Newsletter* **45**(3), 1, 5–7 (2001).
116. J. ILLINGWORTH AND J. KITTNER. Minimum error thresholding. *Pattern Recognition* **19**(1), 41–47 (1986).
117. J. ILLINGWORTH AND J. KITTNER. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing* **44**, 87–116 (1988).
118. International Color Consortium. "Specification ICC.1:2010-12 (Profile Version 4.3.0.0): Image Technology Colour Management—Architecture, Profile Format, and Data Structure" (2010). <http://www.color.org>.
119. International Electrotechnical Commission, IEC, Geneva. "IEC 61966-2-1: Multimedia Systems and Equipment—Colour Measurement and Management, Part 2-1: Colour Management—Default RGB Colour Space—sRGB" (1999). <http://www.iec.ch>.
120. International Organization for Standardization, ISO, Geneva. "ISO 13655:1996, Graphic Technology—Spectral Measurement and Colorimetric Computation for Graphic Arts Images" (1996).

-
- 121. International Organization for Standardization, ISO, Geneva. “ISO 15076-1:2005, Image Technology Colour Management—Architecture, Profile Format, and Data Structure: Part 1” (2005). Based on ICC.1:2004-10.
 - 122. International Telecommunications Union, ITU, Geneva. “ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange” (1998).
 - 123. International Telecommunications Union, ITU, Geneva. “ITU-R Recommendation BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios” (1999).
 - 124. K. JACK. “Video Demystified—A Handbook for the Digital Engineer”. LLH Publishing, Eagle Rock, VA, third ed. (2001).
 - 125. B. JÄHNE. “Practical Handbook on Image Processing for Scientific Applications”. CRC Press, Boca Raton, FL (1997).
 - 126. B. JÄHNE. “Digitale Bildverarbeitung”. Springer-Verlag, Berlin, fifth ed. (2002).
 - 127. B. JÄHNE. “Digital Image Processing”. Springer-Verlag, Berlin, sixth ed. (2005).
 - 128. A. K. JAIN. “Fundamentals of Digital Image Processing”. Prentice Hall, Englewood Cliffs, NJ (1989).
 - 129. R. JAIN, R. KASTURI, AND B. G. SCHUNCK. “Machine Vision”. McGraw-Hill, Boston (1995).
 - 130. Y. JIA AND T. DARRELL. Heavy-tailed distances for gradient based image descriptors. In “Proceedings of the Twenty-Fifth Annual Conference on Neural Information Processing Systems (NIPS)”, Grenada, Spain (December 2011).
 - 131. X. Y. JIANG AND H. BUNKE. Simple and fast computation of moments. *Pattern Recognition* **24**(8), 801–806 (1991).
 - 132. L. JIN AND D. LI. A switching vector median filter based on the CIELAB color space for color image restoration. *Signal Processing* **87**(6), 1345–1354 (2007).
 - 133. J. N. KAPUR, P. K. SAHOO, AND A. K. C. WONG. A new method for gray-level picture thresholding using the entropy of the histogram. *Computer Vision, Graphics, and Image Processing* **29**, 273–285 (1985).
 - 134. B. KIMIA. A large binary image database. Technical Report, LEMS Vision Group, Brown University (2002).
 - 135. J. KING. Engineering color at Adobe. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 15, pp. 341–369. Wiley, New York (2002).
 - 136. R. A. KIRSCH. Computer determination of the constituent structure of biological images. *Computers in Biomedical Research* **4**, 315–328 (1971).
 - 137. L. KITCHEN AND A. ROSENFELD. Gray-level corner detection. *Pattern Recognition Letters* **1**, 95–102 (1982).
 - 138. D. E. KNUTH. “The Art of Computer Programming, Volume 2: Seminumerical Algorithms”. Addison-Wesley, third ed. (1997).
 - 139. J. J. KOENDERINK. The structure of images. *Biological Cybernetics* **50**(5), 363–370 (1984).
 - 140. A. KOSCHAN AND M. A. ABIDI. Detection and classification of edges in color images. *IEEE Signal Processing Magazine* **22**(1), 64–73 (2005).
 - 141. A. KOSCHAN AND M. A. ABIDI. “Digital Color Image Processing”. Wiley (2008).
 - 142. P. KOVESI. Arbitrary Gaussian filtering with 25 additions and 5 multiplications per pixel. Technical Report UWA-CSSE-09-002, The

- University of Western Australia, School of Computer Science and Software Engineering (2009).
143. F. P. KUHL AND C. R. GIARDINA. Elliptic Fourier features of a closed contour. *Computer Graphics and Image Processing* **18**(3), 236–258 (1982).
 144. M. KUWAHARA, K. HACHIMURA, S. EIHO, AND M. KINOSHITA. Processing of RI-angiographic image. In K. PRESTON AND M. ONOE, editors, “Digital Processing of Biomedical Images”, pp. 187–202. Plenum, New York (1976).
 145. D. C. LAY. “Linear Algebra and Its Applications”. Pearson, Boston, third ed. (2006).
 146. P. E. LESTREL, editor. “Fourier Descriptors and Their Applications in Biology”. Cambridge University Press, New York (1997).
 147. P.-S. LIAO, T.-S. CHEN, AND P.-C. CHUNG. A fast algorithm for multilevel thresholding. *Journal of Information Science and Engineering* **17**, 713–727 (2001).
 148. C. C. LIN AND R. CHELLAPPA. Classification of partial 2-D shapes using Fourier descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **9**(5), 686–690 (1987).
 149. B. J. LINDBLOOM. Accurate color reproduction for computer graphics applications. *SIGGRAPH Computer Graphics* **23**(3), 117–126 (1989).
 150. T. LINDEBERG. “Scale-Space Theory in Computer Vision”. Kluwer Academic Publishers (1994).
 151. T. LINDEBERG. Feature detection with automatic scale selection. *International Journal of Computer Vision* **30**(2), 77–116 (1998).
 152. D. G. LOWE. Object recognition from local scale-invariant features. In “Proceedings of the 7th IEEE International Conference on Computer Vision”, vol. 2 of “ICCV’99”, pp. 1150–1157, Kerkyra, Corfu, Greece (1999).
 153. D. G. LOWE. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision* **60**, 91–110 (2004).
 154. B. D. LUCAS AND T. KANADE. An iterative image registration technique with an application to stereo vision. In P. J. HAYES, editor, “Proceedings of the 7th International Joint Conference on Artificial Intelligence IJCAI’81”, pp. 674–679, Vancouver, BC (1981). William Kaufmann, Los Altos, CA.
 155. R. LUKAC, B. SMOLKA, AND K. N. PLATANIOTIS. Sharpening vector median filters. *Signal Processing* **87**(9), 2085–2099 (2007).
 156. R. LUKAC, B. SMOLKA, K. N. PLATANIOTIS, AND A. N. VENETSANOPoulos. Vector sigma filters for noise detection and removal in color images. *Journal of Visual Communication and Image Representation* **17**(1), 1–26 (2006).
 157. P. C. MAHALANOBIS. On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India* **2**(1), 49–55 (1936).
 158. S. MALLAT. “A Wavelet Tour of Signal Processing”. Academic Press, New York (1999).
 159. C. MANCAS-THILLOU AND B. GOSSELIN. Color text extraction with selective metric-based clustering. *Computer Vision, Graphics, and Image Processing: Image Understanding* **107**(1-2), 97–107 (2007).
 160. M. J. MARON AND R. J. LOPEZ. “Numerical Analysis”. Wadsworth Publishing, third ed. (1990).
 161. D. MARR AND E. HILDRETH. Theory of edge detection. *Proceedings of the Royal Society of London, Series B* **207**, 187–217 (1980).
 162. E. H. W. MEIJERING, W. J. NIJSEN, AND M. A. VIERGEVER. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis* **5**(2), 111–126 (2001).

-
163. J. MIANO. "Compressed Image File Formats". ACM Press, Addison-Wesley, Reading, MA (1999).
164. D. P. MITCHELL AND A. N. NETRAVALI. Reconstruction filters in computer-graphics. In R. J. BEACH, editor, "Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88", pp. 221–228, Atlanta, GA (1988). ACM Press, New York.
165. P. A. MLSNA AND J. J. RODRIGUEZ. Gradient and Laplacian-type edge detection. In A. BOVIK, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York (2000).
166. P. A. MLSNA AND J. J. RODRIGUEZ. Gradient and Laplacian-type edge detection. In A. BOVIK, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York, second ed. (2005).
167. J. MOROVIC. "Color Gamut Mapping". Wiley (2008).
168. J. D. MURRAY AND W. VANRYPER. "Encyclopedia of Graphics File Formats". O'Reilly, Sebastopol, CA, second ed. (1996).
169. M. NADLER AND E. P. SMITH. "Pattern Recognition Engineering". Wiley, New York (1993).
170. M. NAGAO AND T. MATSUYAMA. Edge preserving smoothing. *Computer Graphics and Image Processing* **9**(4), 394–407 (1979).
171. S. K. NAIK AND C. A. MURTHY. Standardization of edge magnitude in color images. *IEEE Transactions on Image Processing* **15**(9), 2588–2595 (2006).
172. W. NIBLACK. "An Introduction to Digital Image Processing". Prentice-Hall (1986).
173. M. NITZBERG AND T. SHIOTA. Nonlinear image filtering with edge and corner enhancement. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **14**(8), 826–833 (1992).
174. M. NIXON AND A. AGUADO. "Feature Extraction and Image Processing". Academic Press, second ed. (2008).
175. W. OH AND W. B. LINDQUIST. Image thresholding by indicator kriging. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**(7), 590–602 (1999).
176. A. V. OPPENHEIM, R. W. SHAFER, AND J. R. BUCK. "Discrete-Time Signal Processing". Prentice Hall, Englewood Cliffs, NJ, second ed. (1999).
177. N. OTSU. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics* **9**(1), 62–66 (1979).
178. N. R. PAL AND S. K. PAL. A review on image segmentation techniques. *Pattern Recognition* **26**(9), 1277–1294 (1993).
179. S. PARIS AND F. DURAND. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision* **81**(1), 24–52 (2007).
180. T. PAVLIDIS. "Algorithms for Graphics and Image Processing". Computer Science Press / Springer-Verlag, New York (1982).
181. O. PELE AND M. WERMAN. A linear time histogram metric for improved SIFT matching. In "Proceedings of the 10th European Conference on Computer Vision (ECCV'08)", pp. 495–508, Marseille, France (October 2008).
182. P. PERONA AND J. MALIK. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12**(4), 629–639 (1990).
183. E. PERSOON AND K.-S. FU. Shape discrimination using Fourier descriptors. *IEEE Transactions on Systems, Man and Cybernetics* **7**(3), 170–179 (1977).

REFERENCES

184. E. PERSOON AND K.-S. FU. Shape discrimination using Fourier descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(3), 388–397 (1986).
185. T. Q. PHAM AND L. J. VAN VLIET. Separable bilateral filtering for fast video preprocessing. In “Proceedings IEEE International Conference on Multimedia and Expo”, pp. CD1–4, Los Alamitos, USA (July 2005). IEEE Computer Society.
186. K. N. PLATANIOTIS AND A. N. VENETSANOPoulos. “Color Image Processing and Applications”. Springer (2000).
187. F. PORIKLI. Constant time O(1) bilateral filtering. In “Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)”, pp. 1–8, Anchorage (June 2008).
188. C. A. POYNTON. “Digital Video and HDTV Algorithms and Interfaces”. Morgan Kaufmann Publishers, San Francisco (2003).
189. S. PRAKASH AND F. V. D. HEYDEN. Normalisation of Fourier descriptors of planar shapes. *Electronics Letters* **19**(20), 828–830 (1983).
190. W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, AND B. P. FLANNERY. “Numerical Recipes”. Cambridge University Press, third ed. (2007).
191. J. PREWITT. Object enhancement and extraction. In B. LIPKIN AND A. ROSENFIELD, editors, “Picture Processing and Psychopictorics”, pp. 415–431. Academic Press (1970).
192. R. R. RAKESH, P. CHAUDHURI, AND C. A. MURTHY. Thresholding in edge detection: a statistical approach. *IEEE Transactions on Image Processing* **13**(7), 927–936 (2004).
193. W. S. RASBAND. “ImageJ”. U.S. National Institutes of Health, MD (1997–2007). <http://rsb.info.nih.gov/ij/>.
194. C. E. REID AND T. B. PASSIN. “Signal Processing in C”. Wiley, New York (1992).
195. D. RICH. Instruments and methods for colour measurement. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 2, pp. 19–48. Wiley, New York (2002).
196. C. W. RICHARD AND H. HEMAMI. Identification of three-dimensional objects using Fourier descriptors of the boundary curve. *IEEE Transactions on Systems, Man, and Cybernetics* **4**(4), 371–378 (1974).
197. I. E. G. RICHARDSON. “H.264 and MPEG-4 Video Compression”. Wiley, New York (2003).
198. T. W. RIDLER AND S. CALVARD. Picture thresholding using an iterative selection method. *IEEE Transactions on Systems, Man, and Cybernetics* **8**(8), 630–632 (1978).
199. L. G. ROBERTS. Machine perception of three-dimensional solids. In J. T. TIPPET, editor, “Optical and Electro-Optical Information Processing”, pp. 159–197. MIT Press, Cambridge, MA (1965).
200. G. ROBINSON. Edge detection by compass gradient masks. *Computer Graphics and Image Processing* **6**(5), 492–501 (1977).
201. P. I. ROCKETT. An improved rotation-invariant thinning algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(10), 1671–1674 (2005).
202. A. ROSENFIELD AND J. L. PFALTZ. Sequential operations in digital picture processing. *Journal of the ACM* **12**, 471–494 (1966).
203. J. C. RUSS. “The Image Processing Handbook”. CRC Press, Boca Raton, FL, third ed. (1998).
204. P. K. SAHOO, S. SOLTANI, A. K. C. WONG, AND Y. C. CHEN. A survey of thresholding techniques. *Computer Vision, Graphics and Image Processing* **41**(2), 233–260 (1988).
205. G. SAPIRO. “Geometric Partial Differential Equations and Image Analysis”. Cambridge University Press (2001).

206. G. SAPIRO AND D. L. RINGACH. Anisotropic diffusion of multivalued images with applications to color filtering. *IEEE Transactions on Image Processing* **5**(11), 1582–1586 (1996).
207. J. SAUVOLA AND M. PIETIKÄINEN. Adaptive document image binarization. *Pattern Recognition* **33**(2), 1135–1143 (2000).
208. H. SCHILDT. “Java: A Beginner’s Guide”. McGraw-Hill Osborne Media (2014).
209. C. SCHMID AND R. MOHR. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(5), 530–535 (1997).
210. C. SCHMID, R. MOHR, AND C. BAUCKHAGE. Evaluation of interest point detectors. *International Journal of Computer Vision* **37**(2), 151–172 (2000).
211. Y. SCHWARZER, editor. “Die Farbenlehre Goethes”. Westerweide Verlag, Witten (2004).
212. M. SEUL, L. O’GORMAN, AND M. J. SAMMON. “Practical Algorithms for Image Analysis”. Cambridge University Press, Cambridge (2000).
213. M. SEZGIN AND B. SANKUR. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging* **13**(1), 146–165 (2004).
214. L. G. SHAPIRO AND G. C. STOCKMAN. “Computer Vision”. Prentice Hall, Englewood Cliffs, NJ (2001).
215. G. SHARMA AND H. J. TRUSSELL. Digital color imaging. *IEEE Transactions on Image Processing* **6**(7), 901–932 (1997).
216. F. Y. SHIH AND S. CHENG. Automatic seeded region growing for color image segmentation. *Image and Vision Computing* **23**(10), 877–886 (2005).
217. N. SILVESTRINI AND E. P. FISCHER. “Farbsysteme in Kunst und Wissenschaft”. DuMont, Cologne (1998).
218. S. N. SINHA, J.-M. FRAHM, M. POLLEFEYS, AND Y. GENC. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications* **22**(1), 207–217 (2011).
219. Y. SIRISATHITKUL, S. AUWATANAMONGKOL, AND B. UYYANONVARA. Color image quantization using distances between adjacent colors along the color axis with highest color variance. *Pattern Recognition Letters* **25**, 1025–1043 (2004).
220. S. M. SMITH AND J. M. BRADY. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision* **23**(1), 45–78 (1997).
221. B. SMOLKA, M. SZCZEPANSKI, K. N. PLATANIOTIS, AND A. N. VENETSANOPoulos. Fast modified vector median filter. In “Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns”, CAIP’01, pp. 570–580, London, UK (2001). Springer-Verlag.
222. M. SONKA, V. HLAVAC, AND R. BOYLE. “Image Processing, Analysis and Machine Vision”. PWS Publishing, Pacific Grove, CA, second ed. (1999).
223. M. SPIEGEL AND S. LIPSCHUTZ. “Schaum’s Outline of Vector Analysis”. McGraw-Hill, New York, second ed. (2009).
224. M. STOKES AND M. ANDERSON. “A Standard Default Color Space for the Internet—sRGB”. Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html (1996).
225. S. SÜSSTRUNK. Managing color in digital image libraries. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 17, pp. 385–419. Wiley, New York (2002).
226. B. TANG, G. SAPIRO, AND V. CASELLES. Color image enhancement via chromaticity diffusion. *IEEE Transactions on Image Processing* **10**(5), 701–707 (2001).

227. C.-Y. TANG, Y.-L. WU, M.-K. HOR, AND W.-H. WANG. Modified SIFT descriptor for image matching under interference. In “Proceedings of the International Conference on Machine Learning and Cybernetics (ICMLC)”, pp. 3294–3300, Kunming, China (July 2008).
228. S. THEODORIDIS AND K. KOUTROUMBAS. “Pattern Recognition”. Academic Press, New York (1999).
229. C. TOMASI AND R. MANDUCHI. Bilateral filtering for gray and color images. In “Proceedings Int'l Conf. on Computer Vision”, ICCV'98, pp. 839–846, Bombay (1998).
230. F. TOMITA AND S. TSUJI. Extraction of multiple regions by smoothing in selected neighborhoods. *IEEE Transactions on Systems, Man, and Cybernetics* **7**, 394–407 (1977).
231. Ø. D. TRIER AND T. TAXT. Evaluation of binarization methods for document images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **17**(3), 312–315 (1995).
232. E. TRUCCO AND A. VERRI. “Introductory Techniques for 3-D Computer Vision”. Prentice Hall, Englewood Cliffs, NJ (1998).
233. D. TSCHUMPERLÉ. “PDEs Based Regularization of Multivalued Images and Applications”. PhD thesis, Université de Nice, Sophia Antipolis, France (2005).
234. D. TSCHUMPERLÉ. Fast anisotropic smoothing of multi-valued images using curvature-preserving PDEs. *International Journal of Computer Vision* **68**(1), 65–82 (2006).
235. D. TSCHUMPERLÉ AND R. DERICHE. Diffusion PDEs on vector-valued images: local approach and geometric viewpoint. *IEEE Signal Processing Magazine* **19**(5), 16–25 (2002).
236. D. TSCHUMPERLÉ AND R. DERICHE. Vector-valued image regularization with PDEs: A common framework for different applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(4), 506–517 (2005).
237. K. TURKOWSKI. Filters for common resampling tasks. In A. GLASSNER, editor, “Graphics Gems I”, pp. 147–165. Academic Press, New York (1990).
238. T. TUYTELAARS AND L. J. VAN GOOL. Matching widely separated views based on affine invariant regions. *International Journal of Computer Vision* **59**(1), 61–85 (2004).
239. J. VAN DE WEIJER. “Color Features and Local Structure in Images”. PhD thesis, University of Amsterdam (2005).
240. M. I. VARDAVOULIA, I. ANDREADIS, AND P. TSALIDES. A new vector median filter for colour image processing. *Pattern Recognition Letters* **22**(6-7), 675–689 (2001).
241. A. VEDALDI AND B. FULKERSON. VLFeat: An open and portable library of computer vision algorithms. <http://www.vlfeat.org/> (2008).
242. F. R. D. VELASCO. Thresholding using the ISODATA clustering algorithm. *IEEE Transactions on Systems, Man, and Cybernetics* **10**(11), 771–774 (1980).
243. D. VERNON. “Machine Vision”. Prentice Hall (1999).
244. P. VIOLA AND M. JONES. Robust real-time face detection. *International Journal of Computer Vision* **57**(2), 137–154 (2004).
245. T. P. WALLACE AND P. A. WINTZ. An efficient three-dimensional aircraft recognition algorithm using normalized Fourier descriptors. *Computer Vision, Graphics and Image Processing* **13**(2), 99–126 (1980).
246. D. WALLNER. Color management and transformation through ICC profiles. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 11, pp. 247–261. Wiley, New York (2002).

-
247. A. WATT. “3D Computer Graphics”. Addison-Wesley, Reading, MA, third ed. (1999).
248. A. WATT AND F. POLICARPO. “The Computer Image”. Addison-Wesley, Reading, MA (1999).
249. J. WEICKERT. “Anisotropic Diffusion in Image Processing”. PhD thesis, Universität Kaiserslautern, Fachbereich Mathematik (1996).
250. J. WEICKERT. A review of nonlinear diffusion filtering. In B. M. TER HAAR ROMENY, L. FLORACK, J. J. KOENDERINK, AND M. A. VIERGEVER, editors, “Proceedings First International Conference on Scale-Space Theory in Computer Vision, Scale-Space’97”, Lecture Notes in Computer Science, pp. 3–28, Utrecht (July 1997). Springer.
251. J. WEICKERT. Coherence-enhancing diffusion filtering. *International Journal of Computer Vision* **31**(2/3), 111–127 (1999).
252. J. WEICKERT. Coherence-enhancing diffusion of colour images. *Image and Vision Computing* **17**(3/4), 201–212 (1999).
253. B. WEISS. Fast median and bilateral filtering. *ACM Transactions on Graphics* **25**(3), 519–526 (2006).
254. M. WELK, J. WEICKERT, F. BECKER, C. SCHNÖRR, C. FEDEERN, AND B. BURGETH. Median and related local filters for tensor-valued images. *Signal Processing* **87**(2), 291–308 (2007).
255. P. WENDYKIER. “High Performance Java Software for Image Processing”. PhD thesis, Emory University (2009).
256. G. WOLBERG. “Digital Image Warping”. IEEE Computer Society Press, Los Alamitos, CA (1990).
257. M.-F. WU AND H.-T. SHEU. Contour-based correspondence using Fourier descriptors. *IEE Proceedings—Vision, Image and Signal Processing* **144**(3), 150–160 (1997).
258. G. WYSZECKI AND W. S. STILES. “Color Science: Concepts and Methods, Quantitative Data and Formulae”. Wiley-Interscience, New York, second ed. (2000).
259. Q. YANG, K.-H. TAN, AND N. AHUJA. Real-time O(1) bilateral filtering. In “Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)”, pp. 557–564, Miami (2009).
260. S. D. YANOWITZ AND A. M. BRUCKSTEIN. A new method for image segmentation. *Computer Vision, Graphics, and Image Processing* **46**(1), 82–95 (1989).
261. G. W. ZACK, W. E. ROGERS, AND S. A. LATT. Automatic measurement of sister chromatid exchange frequency. *Journal of Histochemistry and Cytochemistry* **25**(7), 741–753 (1977).
262. C. T. ZAHN AND R. Z. ROSKIES. Fourier descriptors for plane closed curves. *IEEE Transactions on Computers* **21**(3), 269–281 (1972).
263. P. ZAMPERONI. A note on the computation of the enclosed area for contour-coded binary objects. *Signal Processing* **3**(3), 267–271 (1981).
264. E. ZEIDLER, editor. “Teubner-Taschenbuch der Mathematik”. B. G. Teubner Verlag, Leipzig, second ed. (2002).
265. T. Y. ZHANG AND C. Y. SUEN. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM* **27**(3), 236–239 (1984).
266. S.-Y. ZHU, K. N. PLATANIOTIS, AND A. N. VENETSANOPoulos. Comprehensive analysis of edge detection in color image processing. *Optical Engineering* **38**(4), 612–625 (1999).
267. S. ZOKAI AND G. WOLBERG. Image registration using log-polar mappings for recovery of large-scale similarity and projective transformations. *IEEE Transactions on Image Processing* **14**(10), 1422–1434 (2005).

REFERENCES

Index

Symbols

\forall , 717
 \exists , 717
 \div , 417, 714
 $*$, 100–102, 125, 283, 490, 541,
 616, 714, 739
 \circledast , 568, 714
 \otimes , 714, 723, 751
 \times , 714
 \oplus , 185, 714
 \ominus , 186, 714
 \circ , 714
 \bullet , 714
 ∂ , 123, 397, 715, 736, 737
 ∇ , 123, 392, 397, 442–444, 715,
 736
 ∇^2 , 139, 434, 611, 715, 738, 763
 \smile , 713, 714
 \cup , 717
 \cap , 717
 \backslash , 717
 \cdots , 714
 \cdots , 714
 \wedge , 715
 \vee , 715
 \sim , 714, 756
 \approx , 714
 \equiv , 714
 \leftarrow , 714
 \leftarrow^+ , 714
 \coloneqq , 714
 $\|\|$, 714, 717
 $\|\|\|$, 714
 $\lceil\rceil$, 714
 $\lfloor\rfloor$, 714
0, 715
 μ , 716, 749, 756
 σ , 716
 τ , 716
 $\&$ (operator), 768
 \mid (operator), 296
 $/$ (operator), 714
 $\%$ (operator), 767
 $\&$ (operator), 296
 \gg (operator), 296
 \ll (operator), 296

A

`abs` (method), 84, 768
absolute value, 714
accumulator, 164
achromatic, 308
`acos` (method), 768
`AdaptiveThreshold` (class), 284,
 286
`AdaptiveThresholdGauss` (alg.), 285
`ADD` (constant), 85
`add` (method), 84, 157
`addChoice` (method), 88
`addGaussianNoise` (method), 758,
 759
`addNumericField` (method), 88
`adj`, 715
adjugate matrix, 521, 715
Adobe
 Illustrator, 12
 Photoshop, 63, 96, 116, 143
 RGB, 354
affine
 combination, 369
 mapping, 515–517, 526
`AffineMapping` (class), 532, 604
aggregate distance, 379
 trimmed, 385
aliasing, 468, 472, 475, 476, 487,
 556
alpha
 channel, 14, 296
 value, 85, 296
ambient lighting, 345
amplitude, 454, 455
`Analyze` (menu), 35
`AND` (constant), 84
and, 197, 715
`angleFromIndex` (method), 175
angular frequency, 454, 472, 476,
 482
anisotropic diffusion, 433–448
Apache Commons Math library,
 696, 727–729, 731
`applyTable` (method), 71, 79, 80,
 83

INDEX	<p>applyTo (method), 200, 385, 389, 449, 532–534, 537, 606</p> <p>approximation, 547, 548</p> <p>ArcTan, 236, 715, 769</p> <p>area</p> <ul style="list-style-type: none"> polygon, 231 region, 231 <p>arithmetic operation, 84</p> <p>array</p> <ul style="list-style-type: none"> 1D, 771 2D, 772 accessing elements, 771 creation, 771 in Java, 771 size, 771 sorting, 776 <p>ArrayList (class), 155</p> <p>Arrays (class), 324, 776</p> <p>ARToolkit, 173</p> <p>asin (method), 768</p> <p>associativity, 186</p> <p>atan (method), 768</p> <p>atan2 (method), 715, 768, 769</p> <p>auto-contrast, 61</p> <ul style="list-style-type: none"> modified, 62 <p>AVERAGE (constant), 85</p> <p>AVI, 608, 664</p> <p>AWT, 296, 360</p> <p>B</p> <p>background, 181, 254</p> <p>BackgroundMode (class), 286</p> <p>bandwidth, 468, 620, 623, 762</p> <p>Bartlett window, 492, 494, 495</p> <p>basis function, 471–475, 481, 487, 503, 504, 510</p> <p>Bayesian decision making, 268</p> <p>BeanShell, 34</p> <p>Bernsen thresholding, 274–275</p> <p>BernsenThreshold (alg.), 275</p> <p>BernsenThreshold (class), 287</p> <p>bias, 171, 750, 752</p> <p>bicubic interpolation, 553</p> <p>BicubicInterpolator (class), 560, 561</p> <p>big endian, 19, 20</p> <p>bilateral filter, 420–432</p> <ul style="list-style-type: none"> color, 424 Gaussian, 423 separable, 428 <p>BilateralFilter (class), 449</p> <p>BilateralFilterColor (alg.), 428</p> <p>BilateralFilterGray (alg.), 424</p> <p>BilateralFilterGraySeparable (alg.), 432</p> <p>BilateralFilterSeparable (class), 449</p> <p>bilinear</p> <ul style="list-style-type: none"> interpolation, 551 mapping, 525, 526 <p>BilinearInterpolator (class), 534, 560</p> <p>BilinearMapping (class), 533</p> <p>binarization, 59, 253</p> <p>binary</p> <ul style="list-style-type: none"> code, 195 image, 11, 132, 181, 209 morphology, 181 value, 19 <p>BinaryMorphologyFilter (class), 198–200</p> <p>BinaryMorphologyFilter.Box (class), 200</p> <p>BinaryMorphologyFilter.Disk (class), 200</p> <p>BinaryProcessor (class), 59</p> <p>BinaryRegion (class), 224, 246</p> <p>binning, 45–47, 54</p> <p>bit</p> <ul style="list-style-type: none"> depth, 9 mask, 296 operation, 297 <p>bitmap image, 11, 225</p> <p>bitwise AND operator, 768</p> <p>black box, 101</p> <p>black-generation function, 322</p> <p>blending, 85</p> <p>Blitter (interface), 84, 85, 88, 145</p> <p>blob, 624</p> <p>block sum</p> <ul style="list-style-type: none"> first-order, 52 second-order, 53 <p>blur</p> <ul style="list-style-type: none"> filter, 89, 90 Gaussian, 115 <p>blur (method), 284</p> <p>blurFloat (method), 284, 287</p> <p>blurGaussian (method), 115, 284</p> <p>BMP, 18, 20, 299</p> <p>border handling, 282</p> <p>boundary, 665</p> <ul style="list-style-type: none"> pixels, 280 <p>bounding box, 218, 231, 232, 239, 241</p> <p>box filter, 93, 103, 125, 283, 415</p> <p>Bradford model, 356, 359</p> <p>BradfordAdaptation (class), 363</p> <p>breadth-first, 212</p> <p>BreadthFirstLabeling (class), 246</p> <p>Brent’s method, 696</p> <p>BrentOptimizer (class), 696</p>
-------	---

Bresenham algorithm, 177
brightness, 58, 263
BuildGaussianScaleSpace (alg.), 624
BuildSiftScaleSpace (alg.), 631
byte, 19
byte (type), 767
ByteProcessor (class), 56, 84, 276, 289, 301, 709

C
C, 715
camera obscura, 4
Canny edge operator, 132–138, 404–406
color, 404–406, 410
grayscale, 410
CannyEdgeDetector (alg.), 135
CannyEdgeDetector (class), 138, 410, 411
card, 38, 714, 715, 717
cardinal spline, 546
cardinality, 714, 715, 717
cascaded Gaussian filters, 616, 761
Catmull-Rom interpolation, 546
CCITT, 12
cdf, *see* cumulative distribution function
ceil, 714
ceil (method), 768
center line detection, 194
centralMoment (method), 235
centroid, 218, 233, 241, 673, 676, 749
CGM format, 12
chain code, 226, 231
chamfer
 algorithm, 577
 matching, 580
ChamferMatcher (class), 585
characteristic equation, 724
Cholesky decomposition, 755
CholeskyDecomposition (class), 755
chord algorithm, 255
chroma, 319
chromatic adaptation, 355
 Bradford model, 356, 359
 XYZ scaling, 355
ChromaticAdaptation (class), 363
chromaticity diagram, 365
CIE, 341
 chromaticity diagram, 342, 345
L*a*b*, 323, 346, 347
LAB, 346
standard illuminant, 344

XYZ, 342, 346, 347, 352, 353, 361
CIELAB, 289, 381, 440
CIELUV, 348, 381, 440
circle, 176, 519, 674, 675
circular component, 328, 374
circularity, 231
circumference, 230
city block distance, 577
clamping, 58, 83, 94
clone (method), 324
close (method), 200
closing, 192, 203
clutter, 581
CMYK, 320–323
collectCorners (method), 156
Collections (class), 157
collinear, 733
collision, 216
Color (class), 309–311, 360
color
 covariance matrix, 418
 difference, 350
 edge, 370, 391–410
 edge magnitude, 399
 edge orientation, 401
 filter, 367–389, 424, 438
 image, 11, 291–328
 keying, 316
 linear mixture, 370
 management, 362
 out-of-gamut, 372
 picker, 328
 pixel, 294, 296
 saturation, 306
 space, 370–374
 table, 295, 299, 300, 326
 temperature, 344
 thresholding, 289
color quantization, 43, 295, 301, 329–338
 3:3:2, 330
 median-cut, 332
 octree, 333
 populosity, 331
color space, 303
 CMYK, 320
 colorimetric, 341–365
 HLS, 307
 HSB, 306, 361
 HSV, 306, 361
 in Java, 358
 Kodak, 361
 LAB, 346
 LUV, 348
 RGB, 292

sRGB, 350
XYZ, 342
YC_bC_r, 319
YIQ, 318
YUV, 317
color system
additive, 291
subtractive, 320
ColorCannyEdgeDetector (alg.), 405
ColorEdgeDetector (class), 410
ColorModel (class), 300, 360
ColorProcessor (class), 296–299,
302, 305, 324
ColorQuantizer (class), 337
ColorSpace (class), 359–361, 363
column vector, 720
comb function, 465
commutativity, 186, 187
compactness, 231
Comparable (interface), 776
compareTo (method), 155
comparing images, 565–584
complementary set, 184
Complex (class), 478, 705
complex
conjugate, 717
number, 456, 717
component
histogram, 47
ordering, 294
compression, 42
computeMatch (method), 574
computer
graphics, 2
vision, 3
concatenation, 596, 714
conditional probability, 268, 757
conductivity
coefficient, 434
function, 436, 438, 441, 442, 450
conic section, 519
connected components problem,
218
container, 155
Contour (class), 224, 246
contour, 131, 219–222
contrast, 40, 58, 263
automatic adjustment, 61
convertToByte (method), 88, 145,
224
convertToByteProcessor
(method), 305
convertToColorProcessor
(method), 158
convertToFloat (method), 145
convertToFloatProcessor
(method), 154, 281, 606, 662
convex hull, 232, 241, 249, 369
convexity, 232, 245
convolution, 100–102, 283, 284,
368, 499, 568, 739
associativity, 102
commutativity, 101
linearity, 101
property, 463, 496
convolve (method), 115, 145
Convolver (class), 115, 145
convolveX (method), 154
convolveXY (method), 154
convolveY (method), 154
coordinate
homogeneous, 515–516, 726–727
transformation, 514
COPY (constant), 85
copyBits (method), 84, 88, 145
Corner (class), 155
corner, 147
detection, 147–159
point, 159
response function, 149, 152
strength, 149
CorrCoeffMatcher (class), 574, 575
correlation, 100, 499, 567
coefficient, 569
cos (method), 768
cosine function, 461
1D, 454
2D, 483, 484
cosine transform, 15, 503–511
cosine² window, 494, 495
countColors (method), 324
covariance, 749
efficient calculation, 750
matrix, 238, 244, 249, 750
covariance matrix
color, 418
create (method), 560
createProcessor (method), 562
createRealMatrix (method), 727,
729
createRealVector (method), 727
creating new images, 56
cross
correlation, 570
product, 694, 723
CRT, 292
CS_CIEXYZ (constant), 361
CS_GRAY (constant), 361
CS_LINEAR_RGB (constant), 361
CS_PYCC (constant), 361
CS_sRGB (constant), 361

cubic
 interpolation, 544, 547
 spline, 546
cumulative
 distribution function, 67, 264
 histogram, 49, 63, 66, 67
cycle length, 454

D

D50, 345, 358, 361
D65, 345, 347, 351
dB, *see* decibel
DCT, 503–511
 1D, 503–504
 2D, 504–509
DCT (method), 506, 509, 510
Dct1d (class), 509
Dct2d (class), 509
debugging, 114
decibel, 338
Decimate (alg.), 624
decimated scale, 637
decimation, 622
deconvolution, 500
delta function, 464
depth of an image, 9
depth-first, 212
DepthFirstLabeling (class), 246
derivative, 434
 estimation from discrete
 samples, 739
 first, 122, 150, 399, 610, 734, 736
 partial, 123, 397, 611, 715
 second, 130, 139, 611, 632
desaturation, 306, 316
 selective, 317
det, 714, 715
determinant, 521, 635, 714, 715,
 724, 733, 745
DFT, 469–501, 667–673, 715
 1D, 469–479
 2D, 481–501
 forward, 668
 inverse, 668
 periodicity, 670, 679
 spectrum, 668
 truncated, 672, 673, 679
DFT (method), 478
Di Zenzo/Cumani algorithm, 402
diameter, 232
DICOM, 26
DIFFERENCE (constant), 85
difference
 filter, 99
 set, 717

difference-of-Gaussians (DoG),
 613, 763
differential equation, 434
diffusion process, 434
digital image, 7
dilate (method), 200, 201
dilation, 185, 203, 251
dimension, 749
Dirac function, 104, 186, 460, 464
direction of maximum contrast,
 404
directional gradient, 398, 737
discrete
 cosine transform, 503–511
 Fourier transform, 469–501, 715
 sine transform, 503
disk filter, 283
distance, 566, 716
 city block, 577
 Mahalanobis, 243, 249
 Manhattan, 577
 mask, 578
 maximum difference, 567
 norm, 382, 656, 660
 squared, 157
 sum of differences, 567
 sum of squared differences, 567
 transform, 576
 weighted, 243
distance norm, 379
distanceComplex (method), 706
distanceMagnitude (method), 706
DistanceTransform (class), 582,
 585
distribution
 normal (Gaussian), 756–758
 uniform, 54, 64, 66
divergence, 434, 442, 737, 738
DIVIDE (constant), 85
DiZenzoCumaniEdgeDetector
 (class), 410
DOES_8C (constant), 300, 301
DOES_8G (constant), 28, 44
DOES_ALL (constant), 451
DOES_RGB (constant), 297, 298
DOES_STACKS (constant), 451
domain, 716
 filter, 420
dominant orientation, 637, 640
dot product, 722, 728
dotProduct (method), 728
dots per inch (dpi), 8, 476
Double (class), 770
double (type), 95
dpi, 476
drawCorner (method), 158

- E**
- E**(constant), 768
 - e**, 715
 - e*, 715
 - eccentricity, 237, 250
 - Eclipse, 31, 32
 - edge
 - direction, 134
 - linking, 137
 - localization, 134
 - map, 131, 132, 161
 - normal, 401
 - orientation, 392, 403
 - sharpening, 139–146
 - strength, 149, 392
 - suppression, 634
 - tangent, 134, 392, 446
 - tracing, 135
 - edge operator, 124–410
 - Canny, 132–138, 404–406
 - compass, 128
 - in ImageJ, 130
 - Kirsch, 129
 - LoG, 130, 133
 - monochromatic, 392–395
 - Prewitt, 125, 133
 - Roberts, 127, 133
 - Robinson, 128
 - Sobel, 125, 128, 130, 133
 - vector-valued (color), 395–404
 - edge-preserving smoothing filter, 413–451
 - Edit(menu), 33
 - effective gamma value, 81
 - EigenDecomposition**(class), 729, 753
 - eigendecomposition, 753
 - eigenpair, 724
 - eigensystem, 446
 - eigenvalue, 148, 149, 238, 399, 402, 409, 446, 634, 723–726, 737, 751
 - ratio, 635
 - eigenvector, 149, 400, 446, 723–726, 737
 - 2 × 2 matrix, 724
 - ellipse, 177, 238, 519, 677, 683
 - parameters, 677
 - elliptical window, 493
 - elongatedness, 237
 - EMF format, 12
 - Encapsulated PostScript (EPS), 12
 - entropy, 263, 264
 - erode**(method), 200, 201
 - erosion, 186, 203
 - error**(method), 30
 - Euclidean distance, 157, 573
 - Euler number, 245
 - Euler's notation, 456
 - evidence, 269
 - EXIF, 16, 351
 - exp**, 715
 - exp**(method), 104, 768
 - extractImage**(method), 606
 - extremum of a function, 633
- F**
- \mathcal{F} , 715
 - false, 715
 - fast Fourier transform, 479, 484, 498
 - FastIsodataThreshold**(alg.), 260
 - FastKuwaharaFilter**(alg.), 417
 - fax encoding, 226
 - feature, 229
 - vector, 242
 - FFT, 496, *see* fast Fourier transform, 668
 - Fiji, 25
 - file format
 - BMP, 18
 - EXIF, 16
 - GIF, 13
 - JFIF, 15
 - JPEG-2000, 16
 - magic number, 20
 - PBM, 18
 - Photoshop, 20
 - PNG, 14
 - RAS, 19
 - RGB, 19
 - TGA, 19
 - TIFF, 12–13
 - XBM/XPM, 19
 - fill**(method), 56
 - filter, 89–118
 - anisotropic diffusion, 433–448
 - bilateral, 420–432
 - blur, 89, 90, 115
 - border handling, 92, 113
 - box, 93, 98, 103, 125, 283, 415
 - cascaded, 616
 - color, 420, 424, 438
 - color image, 143, 367–389, 416

computation, 93
 debugging, 114
 derivative, 123
 difference, 99
 disk, 283
 domain, 420
 edge, 124–130
 edge-preserving smoothing, 413–451
 efficiency, 112
 Gaussian, 98, 103, 115, 134, 148, 150, 283, 413, 423, 446, 610, 617, 761–763
 HSV color space, 375
 ImageJ, 115–116
 impulse response, 104
 in frequency space, 496
 indexed image, 299
 inverse, 499
 jitter, 118
 kernel, 91, 100, 368, 392
 Kuwahara-type, 414–420
 Laplacian, 99, 117, 139, 145
 Laplacian-of-Gaussian, 610
 linear, 91–105, 115, 367–377, 739
 low-pass, 98, 284, 415, 623
 maximum, 105, 116, 207
 median, 107, 116, 181
 min/max, 281
 minimum, 105, 116, 207
 morphological, 181–208
 multi-dimensional, 379
 Nagao-Matsuyama, 415
 nonhomogeneous, 118
 nonlinear, 105–112, 116, 378–389
 normalized, 95
 Perona-Malik, 436–441
 range, 421
 scalar median, 378, 388
 separable, 102, 103, 140, 284, 613, 620
 sharpening vector median, 382
 smoothing, 94, 95, 98, 143, 368, 370
 sombrero, 612
 successive Gaussians, 616
 Tomita-Tsuji, 417
 Tschumperle-Deriche, 444–448
 unsharp masking, 142
 vector median, 378, 389
 weighted median, 109
final (type), 771, 774
Find_Corners (plugin), 158
Find_Straight_Lines (plugin), 173
FindCommands (menu), 33
findCorners (method), 157, 158
findEdges (method), 130
 finite differences, 434
 FITS, 26
 flat image, 14
Float (class), 770
 floating-point image, 11
FloatProcessor (class), 154
 flood filling, 210–212
 floor, 714
floor (method), 768
floorMod (method), 767, 768
 Flusser's moments, 242
 foreground, 181, 254
 four-point mapping, 519
 Fourier, 457
 analysis, 457
 coefficients, 457
 integral, 457
 series, 457
 shape descriptor, 229, 665–711
 spectrum, 229, 458, 469
 transform, 454–501, 667–673, 715, 762
 transform pair, 459, 461, 462
 Fourier descriptor, 665–711
 elliptical, 709
 from polygon, 682
 geometric effects, 687–692
 invariance, 692–700, 708
 Java implementation, 704
 magnitude, 700
 matching, 700–704, 706
 normalization, 692–700, 707
 pair, 676–681
 phase, 690
 reconstruction, 668, 685
 reflection, 691
 start point, 689
 trigonometric, 667, 682, 710
FourierDescriptor (class), 704
FourierDescriptorFromPolygon (alg.), 685
FourierDescriptorFromPolygon (class), 707
FourierDescriptorUniform (alg.), 669, 673
FourierDescriptorUniform (class), 707
 frequency, 454, 476
 2D, 486
 angular, 454, 455, 472, 482
 common, 455
 directional, 487
 distribution, 67
 effective, 486, 487

fundamental, 457, 476
maximum, 468, 487
space, 459, 475, 496
Frobenius norm, 418, 751
fromCIEXYZ (method), 358–360
fromRGB (method), 364
function
 basis, 471–475
 complex-valued, 666
 cosine, 454
 delta, 464
 Dirac, 460, 464
 distance, 700, 701
 gradient, 397
 hash, 701
 impulse, 460, 464
 Jacobian, 397
 partial derivative, 397
 periodic, 454, 671
 scalar-valued, 735
 sine, 454
 trigonometric, 134
 vector-valued, 395, 735
fundamental
 frequency, 457, 476
 period, 476

G

gamma (method), 84
gamma correction, 74–82, 305,
 358, 361, 372
applications, 78
inverse, 82
modified, 80–82, 352
gamut, 321, 345, 351, 354
garbage, 771
Gaussian
 area formula, 231
 component, 758
 derivative, 610
 distribution, 54, 258, 266, 268,
 269, 756, 758
 filter, 98, 103, 115, 148, 150,
 282, 423, 446, 610, 617,
 761–763
 filter size, 103
 function, 460, 462
 kernel, 283
 mixture, 266
 noise, 758
 normalized, 284
 scale space, 615, 761
 separable, 103
 successive, 616, 761
 weight, 638
 window, 492, 493, 495

GaussianBlur (class), 115, 145,
 284, 286, 287
GaussianFilter (class), 145
GenericDialog (class), 85, 86, 88,
 117
GenericFilter (class), 385, 389,
 449
geometric operation, 513–537
get (method), 29, 30, 58, 66, 113,
 307
get2dHistogram (method), 327
getAccumulator (method), 174
getAccumulatorImage (method),
 175
getAccumulatorMax (method), 175
getAccumulatorMaxImage
 (method), 175
getAngle (method), 176
getBlues (method), 301, 302
getBounds (method), 606
getCoefficient (method), 706
getCoefficients (method), 705
getColorModel (method), 300, 301
getComponents (method), 361
getCornerPoints (method), 606
getCount (method), 176
getCovarianceMatrix (method),
 752
getData (method), 727
getDistance (method), 176
getEdgeBinary (method), 410
getEdgeMagnitude (method), 410
getEdgeOrientation (method),
 410
getEdgeTraces (method), 410
getEigenvector (method), 729
getEntry (method), 727
getf (method), 575, 576, 759
getForegroundColor (method),
 328
getGreens (method), 301, 302
getHeight (method), 29, 30, 759
getHistogram (method), 45, 56,
 66, 71, 289
getImage (method), 30
getInnerContours (method), 224
getIntArray (method), 585
getInterpolatedValue (method),
 560
getInverse (method), 532
getIteration (method), 606
getLines (method), 174
getMapSize (method), 300, 301
getMatch (method), 575, 585, 604,
 606, 607
getMatchValue (method), 576, 585

getMaxCoefficientPairs
 (method), 706
getMaxNegHarmonic (method), 706
getMaxPosHarmonic (method), 706
getNextChoiceIndex (method), 88
getNextNumber (method), 88
getOpenImages (method), 88
getOuterContours (method), 224
GetPartialReconstruction (alg.), 684
getPix (method), 562
getPixel (method), 29, 113, 298,
 768
getPixels (method), 154, 297, 772
getPixelSize (method), 301
getPolygon (method), 538
getProcessor (method), 30, 88
getRadius (method), 176
getRealEigenvalues (method),
 729
getReconstruction (method), 706
getReconstructionPoint
 (method), 707
getReds (method), 301, 302
getReferenceMappingTo (method),
 604, 606
getReferencePoint (method), 175,
 176
getReferencePoints (method),
 604
getRegions (method), 224
getRmsError (method), 604, 606
getRoi (method), 538, 606
getShortTitle (method), 56, 88
getSiftFeatures (method), 662
getSolver (method), 730, 731
GetStartPointPhase (alg.), 698
getThreshold (method), 286, 288
getType (method), 30, 606
getWeightingFactors (method),
 305
getWidth (method), 29, 30, 759
GIF, 13, 20, 26, 43, 226, 295, 299
GIMP, 447
global operation, 57
GlobalThreshold (class), 284
grad, 715, 736
gradient, 122, 123, 148, 150, 392,
 434, 436, 633, 715, 736, 738
 directional, 398, 736, 737
 magnitude, 133, 637, 638
 maximum direction, 737
 multi-dimensional, 397
 orientation, 133, 637, 638
 scalar, 397, 401
vector, 133, 134
vector field, 736

graph, 208, 218
GRAY8 (constant), 30
grayscale
 conversion, 304, 353
 image, 10, 14
 morphology, 202
GrayscaleEdgeDetector (class),
 410

H

H, 715
h, 715
Hadamard transform, 510
Hanning window, 491, 492, 494,
 495
harmonic number, 671
Harris corner detector, 148, 636
HarrisCornerDetector (class), 158
hasComplexEigenvalues (method),
 729
hasConverged (method), 604, 606
hash function, 701
HDTV, 319
heat equation, 434
Hertz, 455, 476
Hessian matrix, 443–445, 447, 448,
 630, 632–634, 647, 715, 738,
 739, 743
 discrete estimation, 445
Hessian normal form, 165, 173
hexadecimal, 19, 296, 768
hierarchical technique, 131
histogram, 37–55, 324–325, 715
 binning, 45
 calculation, 43
 color image, 46
 component, 47
 cumulative, 49, 63, 67
 equalization, 63
 matching, 70
 multiple peaks, 640
 normalized, 67
 orientation, 637, 639
 smoothing, 639
 specification, 66–73
HLS, 306, 307, 311–314, 316
HLSToRGB (method), 315
hom, 715, 726
homogeneous
 coordinate, 515–516, 715,
 726–727
 linear equation, 724
 point operation, 57, 64, 66
 region, 414
homography, 524
hot spot, 91, 184

INDEX	<p>Hough transform, 132, 161–180 algorithm, 168 bias, 171 edge strength, 171 ellipse, 177 for circles, 176 for lines, 176 generalized, 178 hierarchical, 172 implementation, 173</p> <p>HoughLine (class), 176 HoughTransformLines (class), 173, 174 HSB, <i>see</i> HSV HSBtoRGB (method), 311, 312, 361 HSV, 289, 306, 309, 314, 316, 318, 361 HsvLinearFilter (alg.), 377 Hu’s moments, 241 Huffman coding, 15 hysteresis thresholding, 134, 135</p> <p>I</p> <p>i, 456, 715, 717 I_n, 716 ICC, 358 profile, 362 ICC_ColorSpace (class), 362 ICC_Profile (class), 362 iconic image, 14 idCT (method), 506, 509, 510 idempotent, 193 identity matrix, 442, 716, 724 IJ (class), 30 IjUtils (class), 88 Illuminant (enum-type), 363 illuminant, 344 image acquisition, 4 analysis, 2 binary, 11, 209 bitmap, 11 color, 11 compression, 42 coordinates, 9 creating new, 56 defects, 41 depth, 9, 11 digital, 7 display, 56 file format, 11 flat, 14 floating-point, 11 grayscale, 10, 14 iconic, 14 indexed color, 11, 14, 294, 337</p> <p>inpainting, 447 intensity, 10 matching, 565–584 padding, 114 palette, 11 plane, 5 pyramid, 621 raster, 12 redisplay, 35 size, 8 space, 101, 496 special, 11 stack, 451, 664 true color, 14 vector, 12 warping, 526</p> <p>ImageAccessor (class), 560–562 ImageExtractor (class), 606, 607 ImageInterpolator (class), 532 ImageJ, 23–35 debugging, 32 filter, 115–116 geometric operation, 531 macro, 26, 31 main window, 26 plugin, 26–31 point operation, 82–87 program structure, 26 snapshot, 31 stack, 25 tutorial, 34 undo, 26, 31 website, 34</p> <p>ImageJ2, 25 ImagePlus (class), 29, 30, 56, 158, 299, 302, 538 ImageProcessor (class), 27, 29, 30, 297, 298, 300–302, 307, 772 ImageStack (class), 608 imagingbook library, VIII, 33, 34 ImgLib2, 25 impulse, 450 function, 104, 460, 464 response, 104, 190 in place processing, 483 IndexColorModel (class), 301–303 indexed color image, 11, 14, 294, 295, 299, 337 initializeMatch (method), 606 insert (method), 145 int (type), 35, 767 integral image, 51–53, 289, 560 IntegralImage (class), 53 intensity histogram, 47 image, 10</p>
-------	---

interest point, 147, 610
intermeans algorithm, 258
interpolation, 539–563, 594, 597
 1D, 539–549
 2D, 549–556
 B-spline, 546, 547
 bicubic, 553, 556
 bilinear, 551, 556
 by convolution, 543
 Catmull-Rom, 545, 546
 cubic, 544
 ideal, 540
 kernel, 543
 Lanczos, 548, 554, 563
 Mitchell-Netravali, 546, 547
 nearest-neighbor, 543, 550, 556,
 557
 spline, 546
InterpolationMethod (class), 560,
 562
intersection
 in Hough space, 168
line, 173, 179
set, 191, 717
invariance, 231, 234, 241, 244, 565,
 692–700
rotation, 696
scale, 693
start point, 694
inverse
 filter, 499
 matrix, 599, 720
 power function, 77
 tangent function, 769
inverse (method), 728
inversion, 59
invert (method), 59, 84
Isodata
 clustering, 258
 thresholding, 258–260
IsodataThreshold (alg.), 259
IsodataThreshold (class), 285
isotropic, 90, 98, 123, 140, 141,
 148, 159, 188, 611
iterateOnce (method), 604, 606
ITU601, 319
ITU709, 78, 82, 305, 319, 328, 351

J

J, 716
Jacobian matrix, 397, 398, 716,
 736, 737
Java
 applet, 25
 arithmetic, 765
 array, 771

 AWT, 27
 class file, 31
 compiler, 31, 772
 integer division, 66, 765
 JVM, 20
 mathematical functions, 768
 rounding, 769
 runtime environment, 25
 virtual machine, 20
JavaScript, 34
JBuilder, 31
JFIF, 15, 18, 20
jitter filter, 118
joint probability, 757
JPEG, 12, 14–18, 20, 26, 43, 226,
 295, 337, 351, 353, 508, 509
JPEG-2000, 16

K

k-d algorithm, 659
kernel, 100
key point
 position refinement, 632
 selection, 630
Kimia image dataset, 242, 250,
 686, 711
Kirsch operator, 129
Kodak Photo YCC color space,
 361
kriging, 289
Kronecker product, 723
Kuwahara-type filter, 414–420
KuwaharaFilter (alg.), 416
KuwaharaFilter (class), 449
KuwaharaFilterColor (alg.), 418

L

LAB, 346
LabColorSpace (class), 359, 363,
 364
label, 210
Lanczos interpolation, 548, 554,
 563
LanczosInterpolator (class), 560
Laplacian, 99, 434, 435, 444
 filter, 99, 139, 141, 145
 operator, 139, 611, 738
Laplacian-of-Gaussian, 117, 610
 approximation by difference of
 Gaussians, 613, 763
 normalized, 612
left-sided vector-matrix product,
 721, 728
Lena, 107
lens, 6
likelihood, 757
line

endpoints, 172
 equation, 162, 165
 Hessian normal form, 165
 intercept/slope form, 162
 intersection, 173
 linear
 blending, 85, 88
 convolution, 100–102
 correlation, 100
 equation, 723, 724
 transformation, 521
 linearity, 463
 lines per inch (lpi), 8
LinkedList (class), 212
List (class), 771
 list, 713
 concatenation, 714
 little endian, 19, 20
 local
 extremum, 630, 734
 mapping, 528
 structure matrix, 148, 400, 402, 445
lock (method), 33
LoG
 filter, 117
 operator, 133
log (method), 31, 84, 768
 log-polar matching, 574
long (type), 35
 lookup table, 82
 low-pass filter, 284, 415
 LSB, 19
 Lucas-Kanade matcher, 587–608
LucasKanadeForwardMatcher
 (class), 605–607
LucasKanadeInverseMatcher
 (class), 605–607
LucasKanadeMatcher (class), 604, 606
LUDecomposition (class), 730
 luma, 320, 354, 440
 luminance, 289, 304, 319, 320, 354, 371, 440
 LUT, 200, 201
 LUV, 348
LuvColorSpace (class), 362
 LZW, 12, 13

M

machine accuracy, 770
 macro recorder, 33
Macros (menu), 34
 magic number, 20
 magnitude, 714
 Mahalanobis distance, 243, 249
 major axis, 235
makeCrf (method), 154
MakeDogOctave (alg.), 631
MakeGaussianKernel2D (alg.), 285
MakeGaussianOctave (alg.), 624, 631
makeGaussKernel1d (method), 104, 145
makeIndexColorImage (method), 301
MakeInvariant (alg.), 697
makeInvariant (method), 707, 710
makeMapping (method), 606
MakeRotationInvariant (alg.), 697
makeRotationInvariant (method), 707
MakeScaleInvariant (alg.), 697
makeScaleInvariant (method), 707
MakeStartPointInvariant (alg.), 698
makeStartPointInvariant
 (method), 707
makeTranslationInvariant
 (method), 707
 Manhattan distance, 577
mapMultiply (method), 728
Mapping (class), 533, 534
 mapping
 affine, 516, 517, 526
 bilinear, 525, 526
 four-point, 519
 linear, 521
 local, 528
 nonlinear, 526
 perspective, 520
 projective, 519–526
 ripple, 527
 spherical, 527
 three-point, 516
 twirl, 526
 mask, 142, 225
MatchDescriptors (alg.), 657
matchDescriptors (method), 663
matchHistograms (method), 71
 matching, 700–704
Math (class), 768, 769
 matrix, 719, 731
 adjugate, 521, 715
 decomposition, 521, 731, 755
 Hessian, 443–445, 447, 448, 630, 632–634, 647, 715, 738, 743
 identity, 442, 716, 724
 inverse, 599, 720, 728
 Jacobian, 397, 398, 716, 736, 737
 norm, 418, 751, 752
 rank, 716, 724

singular, 724
symmetric, 725
trace, 716
transpose, 716, 720
MatrixUtils (class), 727
MAX (constant), 85, 116
max (method), 84, 768
MaxEntropyThresholder (class), 285
maximum
 entropy thresholding, 263–266
 filter, 207, 281
 frequency, 468, 487
 likelihood estimation, 756
 local contrast, 399
MaximumEntropyThreshold (alg.), 267
mean, 50–51, 53, 255, 257, 279, 414, 749, 756, 758, 759
 from histogram, 50
 vector, 749
MeanThresholder (class), 285
Measure (menu), 35
media-oriented color, 353
medial axis transform, 194
MEDIAN (constant), 116
median, 51, 256
 filter, 107, 116, 181, 378
 filter (weighted), 109
median-cut algorithm, 332
MedianCutQuantizer (class), 337, 338
MedianThresholder (class), 285
mesh partitioning, 528
Mexican hat filter, 99, 612
mid-range, 257
MIN (constant), 85, 116
min (method), 84, 768
MinErrorThresholder (class), 285
minimum error thresholding, 266–272
minimum filter, 207, 281
MinimumErrorThreshold (alg.), 273
Mitchell-Netravali interpolation, 547
mixture model, 758
mod, 478, 716, 766
mode, 756
modified auto-contrast, 62
modulus, *see* mod
moment, 226, 233–244
 central, 234
 Flusser, 242
 Hu, 241
 invariant, 241
 least inertia, 235
moment (method), 235
monochromatic edge detection, 392–395
MonochromaticColorEdge (alg.), 395
MonochromaticEdgeDetector
 (class), 410
morphing, 529
morphological filter, 181–208
 binary, 181
 closing, 192, 203
 color, 202
 dilation, 185, 203
 erosion, 186, 203
 grayscale, 202
 opening, 192, 203
 outline, 189
MPEG, 509
MSB, 19
mult (method), 154
multi-resolution techniques, 131
MultiGradientColorEdge (alg.), 402
MULTIPLY (constant), 85
multiply (method), 84, 145, 728
My_Inverter (plugin), 29

INDEX

N

N, 716
N, 254, 269, 756, 759
Nagao-Matsuyama filter, 415
NaN (constant), 770
nCentralMoment (method), 235
nearest-neighbor interpolation, 543
NearestNeighborInterpolator
 (class), 560
negative frequency, 676
NEGATIVE_INFINITY (constant), 770
neighborhood, 210, 230
 2D, 274, 380, 383, 421, 422, 609, 746
 3D, 630, 633
 square, 415
NetBeans, 31, 32
neutral
 element, 104, 186, 616
 point, 343
nextGaussian (method), 54, 759
nextInt (method), 54
Niblack thresholding, 275–279
NiblackThreshold (alg.), 281
NiblackThresholder (class), 286, 287
NiblackThresholderGauss (class), 287
NIH-Image, 25
nil, 716

- N**o_Changes (constant), 31, 44, 302
noise, 159
 energy, 338
 Gaussian, 758
 reduction, 413
nominal gamma value, 81
non-maximum suppression, 133,
 137, 169
nonhomogeneous filter, 118
nonhomogeneous operation, 57
norm, 379, 393, 394, 396, 425, 716
 Euclidean, 714, 720
 Frobenius, 418, 751
 matrix, 418, 751, 752
 vector, 720
normal distribution, 54, 756
normalization, 95
normalized
 histogram, 67
 kernel, 284, 369
NormType (class), 389
NTSC, 78, 317, 318
null (constant), 771
Nyquist, 468, 487
- O**
OCR, 229, 245, 251, 279
octave, 614, 617, 618, 621–624,
 628, 631, 642
octree algorithm, 333
OctreeQuantizer (class), 337
open (method), 200
opening, 192, 203
operate (method), 728
optical axis, 5
OR (constant), 84
orientation, 235, 486, 488
 dominant, 640
 histogram, 637
orthogonal, 511
oscillation, 454, 455
Otsu’s method, 260–263
OtsuThreshold (alg.), 262
OtsuThresholder (class), 285
out-of-gamut colors, 372
outer product, 103, 723, 728
outerProduct (method), 728
outlier, 257
outline, 189
outline (method), 200, 202
OutOfBoundsStrategy (class), 562
- P**
packed ordering, 294–296
padding, 114, 222
PAL, 78, 317
palette, 295, 299, 300
image, *see* indexed color image
parabolic fitting, 733–735
parameter space, 163
partial
 derivative, 123, 715
 differential equation, 434
Parzen window, 491, 492, 494, 495
pattern recognition, 3, 229
PDF, 12
pdf, *see* probability density
 function
perimeter, 230
period, 454
periodicity, 454, 482, 486, 489
Perona-Malik filter, 436–441
 color, 438
 gray, 436
Perona_Malik_Demo (plugin), 451
PeronaMalikColor (alg.), 442
PeronaMalikFilter (class), 450
PeronaMalikGray (alg.), 438
perspective
 image, 177
 mapping, 520
 projection, 5
phase, 455, 477, 690, 694, 695, 699
 angle, 455
Photoshop, 20, 378, 393
PI (constant), 768
PICT format, 12
piecewise linear function, 68
pinhole camera, 4
pipette tool, 328
pixel, 4
 value, 9
PixelInterpolator (class), 532,
 534, 560, 561
PKZIP, 14
planar ordering, 294
Plessey detector, 148
PlugIn (interface), 27, 30, 33
PlugInFilter (class), 606
PlugInFilter (interface), 27, 29,
 33, 35, 297, 389
PNG, 14, 20, 26, 299, 351
point operation, 57–87
 arithmetic, 82
 effects on histogram, 59
 gamma correction, 74
 histogram equalization, 63
 homogeneous, 83
 in ImageJ, 82–87
 inversion, 59
 thresholding, 59
point set, 184
point spread function, 105

Point2D(class), 538
polar method, 758
Polygon, 667, 682
 area, 231
 path length, 683
 uniform sampling, 667, 710
PolygonRoi(class), 538
PolygonSampler(class), 708
populosity algorithm, 331
positive definite, 754
POSITIVE_INFINITY(constant),
 770
posterior probability, 268
PostScript, 12
pow(method), 80, 768
power spectrum, 477, 485
preMultiply(method), 728
Prewitt operator, 125, 133
primary color, 292
principal curvature ratio, 635
print pattern, 499
prior probability, 268, 273
probability, 67, 756
 conditional, 268, 757
 density function, 67, 264
 distribution, 67, 264
 joint, 757
 posterior, 268
 prior, 264, 268, 270, 273
product
 cross, 714, 723
 dot, 722, 728
 matrix-vector, 721
 outer, 714, 723, 728
 scalar, 722, 728
 vector, 722–723
profile connection space, 358, 361
projection, 244, 250, 325, 722
projective mapping, 519–526
ProjectiveMapping(class), 532,
 534, 537, 604, 606
pseudo-perspective mapping, 520
pseudocolor, 326
putPixel(method), 29, 113, 298,
 768
pyramid, 131, 621

Q

\mathcal{Q} , 522, 525
QR decomposition, 521
QRDecomposition(class), 731
quadratic function, 632, 633, 640
quadrilateral, 519, 716
QuantileThreshold(alg.), 257
QuantileThresholder(class), 285
quantization, 8, 59, 329–338

linear, 330
scalar, 329
vector, 331
quasi-separable, 613

R

\mathbb{R} , 716
radiusFromIndex(method), 175
Random(class), 758, 759
Random(package), 54
random
 image, 54
 process, 67
 variable, 67, 756
random(method), 54, 768
range
 filter, 421
rank, 716, 724
rank(method), 116, 281
rank ordering, 378
RankFilters(class), 116, 275, 276,
 281
RAS format, 19
raster image, 12
RAW format, 299
RealMatrix(class), 727, 729
RealVector(class), 727, 729
Record(menu), 34
rectangular
 pulse, 460, 462
 window, 493
RecursiveLabeling(class), 246
redisplaying an image, 35
reflection, 185, 187
refraction index, 528
region, 209–251
 area, 231, 234, 249
 centroid, 233, 249
 convex hull, 232
 diameter, 232
 eccentricity, 237
 homogeneous, 414
 labeling, 210–219
 major axis, 235
 matrix representation, 225
 moment, 233
 orientation, 235
 perimeter, 230
 projection, 244
 run length encoding, 225
 topological property, 244
region of interest, 327, 536, 538,
 605, 606
RegionContourLabeling(class),
 224, 246
RegionLabeling(class), 246

INDEX	relative colorimetry, 355 remainder operator, 767 resampling, 529 resolution, 8 RGB color image, 291 color space, 292, 316 format, 19 RGBtoHLS (method), 314 RGBtoHSB (method), 310, 361 RGBtoHSV (method), 311 right-sided vector-matrix product, 721, 728 rint (method), 768 ripple mapping, 527 RippleMapping (class), 533 Roberts operator, 127, 133 Robinson operator, 128 Roi (class), 538, 606 Rotation (class), 532, 533 rotation, 241, 497, 513, 515, 688 round, 84, 716 round (method), 80, 768 rounding, 58, 84, 766, 769 roundness, 231 row vector, 720 run (method), 27 run length encoding, 225	decimated, 637 increment, 630 initial, 617 ratio, 617 relative, 618 scale space, 610 decimation, 621, 622 discrete, 616 Gaussian, 615 hierarchical, 620, 623 LoG/DoG, 619, 623 octave, 621 SIFT, 624–636 spatial position, 623 sub-sampling, 621 Scaling (class), 532 scaling, 241, 513, 515 segmentation, 253, 289 separability, 102, 117, 188, 284, 507 separable filter, 99, 140, 613 sequence, 713 SequentialLabeling (class), 246 Set (class), 771 set, 184, 713 difference, 717 intersection, 717 union, 717 set (method), 29, 30, 58, 66, 113, 307 setCoefficient (method), 706 setColor (method), 158 setColorModel (method), 300, 301, 303 setEntry (method), 727 setf (method), 759 setNormalize (method), 115, 145 setPix (method), 562 setRGBWeights (method), 305 setup (method), 27, 28, 31, 297, 300, 411 setValue (method), 56 Shah function, 465 Shannon, 468 shape feature, 229 number, 228, 249 reconstruction, 668, 679, 681, 684, 685, 706 representation, 208 rotation, 688 sharpen (method), 145 sharpening vector median filter, 382 SharpeningVectorMedianFilter (alg.), 384
-------	--	---

Shear (class), 532
shearing, 515
ShereMapping (class), 533
shift property, 463
ShortProcessor (class), 289
show (method), 56, 158, 299
showDialog (method), 88
SIFT, 609–664
 algorithm summary, 647
 descriptor, 640–647
 examples, 654–657
 feature matching, 648–660
 implementation, 634, 661–663
 parameters, 648
 scale space, 624–636
SiftDescriptor (class), 662
SiftDetector (class), 662
SiftMatcher (class), 663
signal
 energy, 338
 space, 101, 459, 475
signal-to-noise ratio, 338
similarity, 463
sin (method), 768
Sinc function, 460, 541, 550
sine
 function, 454, 461
 transform, 503
singular-value decomposition, 731
SingularValueDecomposition
 (class), 731
size (method), 706
skeletonize (method), 202, 208
skew angle, 251
smoothing filter, 91, 94, 283
SNR, 338
Sobel operator, 125, 133, 392, 394
 extended, 128
solve (method), 730, 731
sombrero filter, 612
sort (method), 110, 157, 324, 776
sorting arrays, 776
source-to-target mapping, 530
spatial sampling, 7
special image, 11
spectrum, 453
spherical mapping, 527
spline
 cardinal, 546
 Catmull-Rom, 545–547
 cubic, 546, 547
 cubic B-, 546, 547, 563
 interpolation, 546
SplineInterpolator (class), 560
sqr (method), 84, 154
sqrt (method), 84, 768
square window, 495
squared local contrast, 398, 402
sRGB, 81, 82, 305, 350, 352, 353
 ambient lighting, 345
 grayscale conversion, 353
 white point, 345
stack, 210, 299
standard deviation, 54, 275, 614, 716
standard illuminant, 344, 355
statistical independence, 756
step edge, 370
structure matrix, 447
structuring element, 184, 188, 202
sub-pixel accuracy, 745
sub-sampling, 623
SUBTRACT (constant), 85, 145
summed area table, 51
super-Gaussian window, 492, 493
SVD, 521
symmetry, 691
System.out (constant), 31

INDEX

T

t, 716
t, 716
tan (method), 768
tangent function, 769
target-to-source mapping, 526, 530
Taylor expansion, 633, 740
 multi-dimensional, 740
template matching, 565, 566
temporal sampling, 7
TGA format, 19
thin (method), 200, 208
thin lens, 6
thinning, 194–195
thinOnce (method), 200
three-point mapping, 516
threshold, 59, 132, 169
threshold (method), 59, 288
threshold surface, 288
Thresholder (class), 284
thresholding, 131, 253–289
 Bernsen, 274–275
 color image, 289
 global, 253–272
 hysteresis, 134
 Isodata, 258–260
 local adaptive, 273–284
 maximum entropy, 263–266
 minimum error, 266–272
 Niblack, 275–279
 Otsu, 260–263
 shape-based, 255
 statistical, 255

- Suvola, 279
 TIFF, 12, 16, 18, 20, 26, 226, 299
 time unit, 455
toArray(method), 157, 727
toCIEXYZ(method), 358–361
toDegrees(method), 768
 Tomita-Tsuji filter, 417
 topological property, 244
toRadians(method), 768
toRGB(method), 364
 total variance, 418, 751
 trace, 419, 443, 444, 716, 737, 738,
 751, 752
 tracking, 147, 607, 664
 transform pair, 459
TransformJ(package), 531
Translation(class), 532, 604
 translation, 241, 515, 687
 transparency, 85, 296, 303
 transpose of a matrix, 720
 tree, 210
 triangle algorithm, 255
 trigonometric coefficient, 684
 trimmed aggregate distance, 385
 tristimulus value, 344
 true, 716
 true color image, 11, 293, 295, 296
 true colorimage, 14
truncate(method), 706, 710
 truncated spectrum, 672, 673
 truncation, 84
 Tschumperle-Deriche filter,
 444–448
TschumperleDericheFilter(alg.), 448
TschumperleDericheFilter
 (class), 450
 tuple, 713
 twirl mapping, 526
TwirlMapping(class), 533
 type cast, 58, 766
- U**
 undercolor-removal function, 322
 uniform distribution, 54, 64, 66
 union, 717
 unit square, 525
 unit vector, 398, 400, 630, 715,
 736, 737
unlock(method), 33
 unsharp masking, 142–146
UnsharpMask(class), 145
unsharpMask(method), 145
unsigned byte(type), 767
updateAndDraw(method), 30, 35
- V**
 variance, 50–51, 53, 256, 275, 414,
 415, 569, 716, 749, 750, 756,
 759, 761
 between classes, 261
 bias, 750
 fast calculation, 50
 from histogram, 50
 local calculation, 279
 total, 418, 751, 752
 within class, 261
 variate, 749
 vector, 713, 719–731
 column, 720
 field, 391, 395, 397, 406, 735–739
 image, 12
 length, 720
 median filter, 378, 389
 norm, 720
 product, 722–723
 row, 720
 unit, 398, 400, 630, 715, 736, 737
 zero, 715
VectorMedianFilter(alg.), 381
VectorMedianFilter(class), 386,
 389
VectorMedianFilterSharpen
 (class), 386
 video, 608
 viewing angle, 345
- W**
 Walsh transform, 510
 warping, 526
wasCanceled(method), 88
 wave number, 472, 482, 487, 504
 wavelet, 510
 website for this book, 34
 weighted distance, 243
 white point, 308, 344, 347
 D50, 345, 358
 D65, 345, 351
 windowed matching, 573
 windowing function, 490–491
 Bartlett, 492, 494, 495
 cosine², 494, 495
 elliptical, 492, 493
 Gaussian, 492, 493, 495
 Hanning, 492, 494, 495
 Parzen, 492, 494, 495
 rectangular pulse, 493
 super-Gaussian, 492, 493
 WMF format, 12
- X**
 XBM/XPM format, 19
 XOR, 191, 716

X

color space, 304, 341–346, 371
scaling, 355

Y

YC_bC_r, 319
YIQ, 318
YUV, 317–319

Z

\mathbb{Z} , 716
zero vector, 715
ZIP, 12

INDEX

About the Authors

Wilhelm Burger received a Master's degree in Computer Science from the University of Utah (Salt Lake City) and a doctorate in Systems Science from Johannes Kepler University in Linz, Austria. As a post-graduate researcher at the Honeywell Systems & Research Center in Minneapolis and the University of California at Riverside, he worked mainly in the areas of visual motion analysis and autonomous navigation. Since 1996, he has been Head of the Digital Media Department at the University of Applied Sciences in Hagenberg, Austria. Privately, Wilhelm appreciates large-engine vehicles, chamber music, and (occasionally) a glass of dry "Veltliner".



Mark J. Burge is a senior scientist at Noblis, Inc. in Washington, D.C. He spent seven years as a research scientist with the Swiss Federal Institute of Science (ETH) in Zürich and the Johannes Kepler University in Linz, Austria. He earned tenure as a computer science professor in the University System of Georgia (USG), and later served as a program director at the National Science Foundation (NSF), at MITRE and the Intelligence Advanced Research Programs Activity (IARPA). He also lectures at the United States Naval Academy (USNA). Personally, Mark is an expert on classic Italian espresso machines.

