Errata

for

Burger, Burge: Digital Image Processing – An Algorithmic Introduction Using Java

www.imagingbook.com

(i.e., short focal length) results in a small image and a large viewing angle, just as occurs when a wide-angle lens is used, while increasing the "focal length" f results in a larger image and a smaller viewing angle, just as occurs when a telephoto lens is used. The negative sign in Eqn. (2.1) means that the projected image is flipped in the horizontal and vertical directions and rotated by 180°. Equation (2.1) describes what is commonly known today as the perspective transformation.[1] Important properties of this theoretical model are that straight lines in 3D space always appear straight in 2D projections and that circles appear as ellipses.

2.2.2 The "Thin" Lens

While the simple geometry of the pinhole camera makes it useful for understanding its basic principles, it is never really used in practice. One of the problems with the pinhole camera is that it requires a very small opening to produce a sharp image. This in turn reduces the amount of light passed through and thus leads to extremely long exposure times. In reality, glass lenses or systems of optical lenses are used whose optical properties are greatly superior in many aspects but of course are also much more complex. Instead we can make our model more realistic, without unduly increasing its complexity, by replacing the pinhole with a "thin lens" as in Fig. 2.3. In this model, the lens is assumed to be symmetric and infinitely thin, such that all light rays passing through it cross through a virtual plane in the middle of the lens. The resulting image geometry is the same as that of the pinhole camera. This model is not sufficiently complex to encompass the physical details of actual lens systems, such as geometrical distortions and the distinct refraction properties of different colors. So while this simple model suffices for our purposes (that is, understanding the mechanics of image acquisition), much more detailed models that incorporate these additional complexities can be found in the literature (see, for example, [59]).

2.2.3 Going Digital

What is projected on the image plane of our camera is essentially a two-dimensional, time-dependent, continuous distribution of light energy. In order to convert this image into a digital image on our computer, three main steps are necessary:

1. The continuous light distribution must be spatially sampled.
2. This resulting function must then be sampled in the time domain to create a single image.
3. Finally, the resulting values must be quantized to a finite range of integers so that they are representable within the computer.

[1] It is hard to imagine today that the rules of perspective geometry, while known to the ancient mathematicians, were only rediscovered in 1430 by the Renaissance painter Brunelleschi.

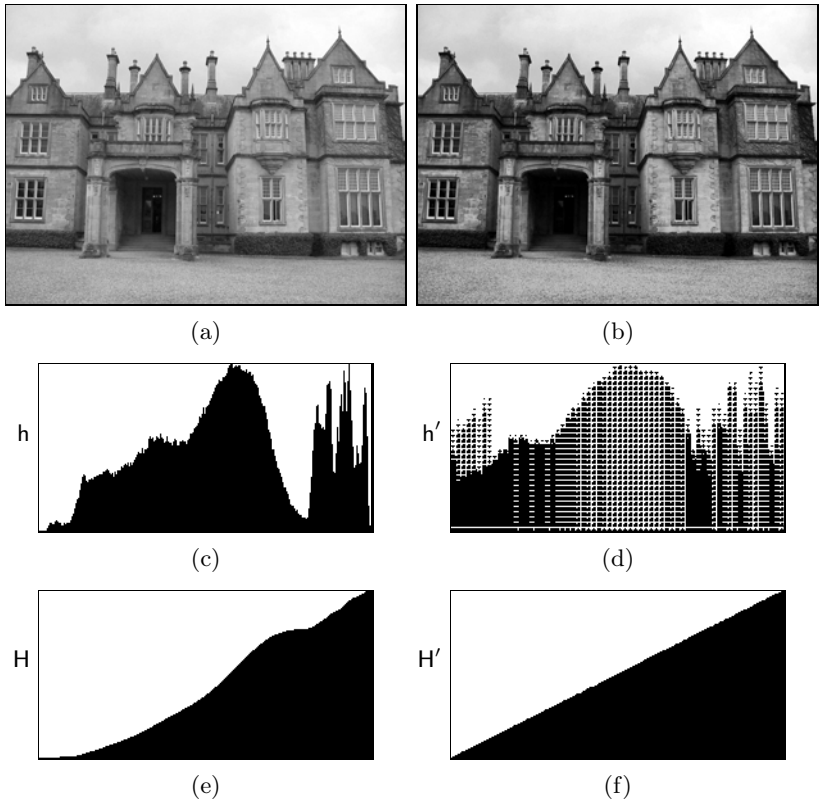


Fig. 5.10 Linear histogram equalization (example). Original image I (a) and modified image I' (b), corresponding histograms h, h' (c, d), and cumulative histograms H, H' (e, f). The resulting cumulative histogram H' (f) approximates a uniformly distributed image. Notice that new peaks are created in the resulting histogram h' (d) by merging original histogram cells, particularly in the lower and upper intensity ranges.

ImageJ by default[5] cumulates the *square root* of the histogram entries using a modified cumulative histogram of the form

$$\tilde{\mathsf{H}}(i) = \sum_{j=0}^{i} \sqrt{\mathsf{h}(j)}\,. \tag{5.12}$$

5.6 Histogram Specification

Although widely implemented, the goal of linear histogram equalization—a uniform distribution of intensity values (as described in the previous section)—appears rather ad hoc, since good images virtually never show such a distribution. In most real images, the distribution of the pixel values is not even remotely uniform but is usually more similar, if at all, to perhaps a Gaussian distribution. The images produced by linear equalization thus usually appear quite unnatural, which renders the technique practically useless.

[5] The "classic" (linear) approach, as described in Eqn. (4.5), is used when simultaneously keeping the Alt key pressed.

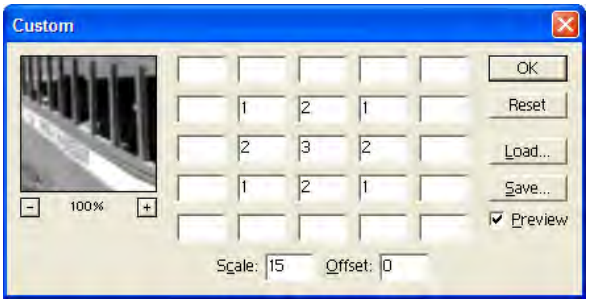


Fig. 6.6
Adobe Photoshop's "Custom Filter" implements linear filters up to a size of 5×5. The filter's coordinate origin ("hot spot") is assumed to be at the center (value set to 3 in this example), and empty cells correspond to zero coefficients. In addition to the (integer) coefficients, common Scale and Offset values can be specified.

centered with an odd number of $(2K + 1)$ columns and $(2L + 1)$ rows $(K, L \geq 0)$. If the image is of size $M \times N$,

$$I(u, v) \quad \text{with} \quad 0 \leq u < M \quad \text{and} \quad 0 \leq v < N,$$

then the filter can be computed for all image coordinates (u', v') with

$$K \leq u' \leq (M - K - 1) \qquad \text{and} \qquad L \leq v' \leq (N - L - 1),$$

as illustrated in Fig. 6.7. Program 6.3 (which is adapted from Prog. 6.2) shows a 7×5 smoothing filter as an example for implementing linear filters of arbitrary size. This example uses integer-valued filter coefficients in combination with a common scale factor `s`, as described above. As usual, the "hot spot" of the filter is assumed to be at the matrix center, and the range of all iterations depends on the dimensions of the filter matrix. In this case, clamping of the results is included (in lines 32–33) as a preventive measure.

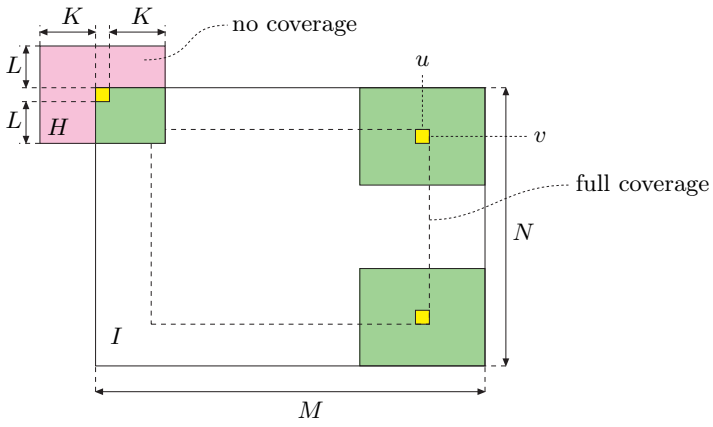


Fig. 6.7
Border geometry. The filter can be applied only at locations (u, v) where the filter matrix H of size $(2K+1) \times (2L+1)$ is fully contained in the image.

6.2.7 Types of Linear Filters

Since the effects of a linear filter are solely specified by the filter matrix (which can take on arbitrary values), an infinite number of different

Exercise 6.7. Implement a weighted median filter (Sec. 6.4.3) as an ImageJ plugin, specifying the weights as a constant, two-dimensional `int` array. Test the filter on suitable images and compare the results with those from a standard median filter. Explain why, for example, the weight matrix

$$W(i,j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{5} & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

does *not* make sense.

Exercise 6.8. Verify the properties of the *impulse* function with respect to linear filters Eqn. (6.30). Create a black image with a white pixel at its center and use this image as the two-dimensional impulse. See if linear filters really deliver the filter matrix H as their impulse response.

Exercise 6.9. Describe the effect of a linear filter with the following filter matrix:

$$H(i,j) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mathbf{0} & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

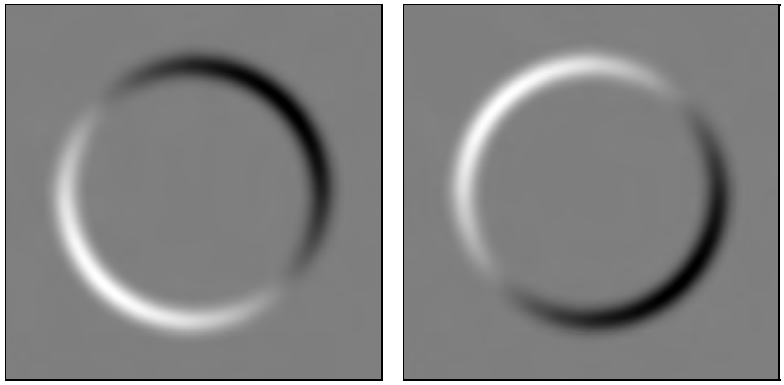
Exercise 6.10. Design a linear filter (matrix) that creates a horizontal blur over a length of 7 pixels, thus simulating the effect of camera movement during exposure.

Exercise 6.11. Program your own ImageJ plugin that implements a Gaussian smoothing filter with variable filter width (radius σ). The plugin should dynamically create the required filter kernels with a size of at least 5σ in both directions. Make use of the fact that the Gaussian function is x/y-separable (see Sec. 6.3.3).

Exercise 6.12. The "*Laplacian of Gaussian*" (LoG) filter (Fig. 6.8) is based on the sum of the second derivatives of the two-dimensional Gaussian. It is defined as

$$\mathrm{LoG}_\sigma(x,y) = -\Big(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\Big) \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Implement the LoG filter as an ImageJ plugin of variable width (σ), analogous to Exercise 6.11. Find out if the LoG function is x/y-separable.



$D_1 = I * H_1^R$ $D_2 = I * H_2^R$

Fig. 7.6 Diagonal gradient components produced by the two Roberts filters.

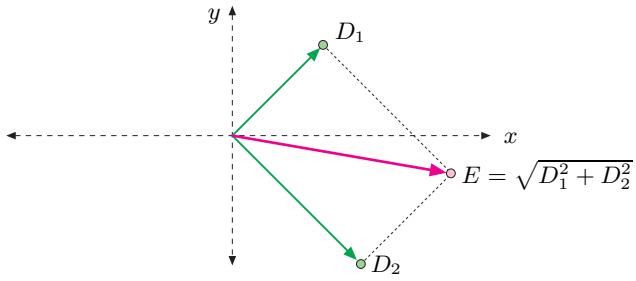


Fig. 7.7 Definition of edge strength for the Roberts operator. The edge strength $E(u, v)$ corresponds to the length of the vector obtained by adding the two orthogonal gradient components (filter results) $D_1(u, v)$ and $D_2(u, v)$.

operator by *Kirsch* [63] and the extended Sobel or *Robinson*[4] operator, which uses the following eight filters with orientations spaced at 45°:

$$H_0^K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad H_4^K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \tag{7.17}$$

$$H_1^K = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \qquad H_5^K = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}, \tag{7.18}$$

$$H_2^K = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad H_6^K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \tag{7.19}$$

$$H_3^K = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \qquad H_7^K = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}. \tag{7.20}$$

Only the results of four of the eight filters $H_0, H_1, \ldots H_7$ above must actually be computed since the four others are identical except for the reversed sign. For example, from the fact that $H_4^K = -H_0^K$ and the convolution being linear (Eqn. (6.18)), it follows that

$$I * H_4^K = I * -H_0^K = -(I * H_0^K); \tag{7.21}$$

[4] G. Robinson. Edge detection by compass gradient masks. *Computer Graphics and Image Processing* **6**(5), 492–501 (1977).

i. e., the result for filter H_4^K is simply the negative result for filter H_0^K. The directional outputs $D_0, D_1, \ldots D_7$ for the eight Robinson filters can thus be computed as follows:

$$\begin{array}{llll} D_0 \leftarrow I * H_0^K & D_1 \leftarrow I * H_1^K & D_2 \leftarrow I * H_2^K & D_3 \leftarrow I * H_3^K \\ D_4 \leftarrow -D_0 & D_5 \leftarrow -D_1 & D_6 \leftarrow -D_2 & D_7 \leftarrow -D_3. \end{array} \tag{7.22}$$

The edge strength E^K at position (u, v) is defined as the maximum of the eight filter outputs; i. e.,

$$\begin{aligned} E^K(u,v) &\triangleq \max\big(D_0(u,v), D_1(u,v), \ldots D_7(u,v)\big) \\ &= \max\big(|D_0(u,v)|, |D_1(u,v)|, |D_2(u,v)|, |D_3(u,v)|\big) \end{aligned} \tag{7.23}$$

and the strongest-responding filter also determines the local edge orientation as

$$\Phi^K(u,v) \triangleq \frac{\pi}{4} j\,, \qquad \text{with } \; j = \underset{0 \le i \le 7}{\operatorname{argmax}} \; D_i(u,v). \tag{7.24}$$

In practice, however, this and other "compass operators" show only minor benefits over the simpler operators described earlier, including the small advantage of not requiring the computation of square roots (which is considered a relatively "expensive" operation).

7.3.4 Edge Operators in ImageJ

The current version of ImageJ implements the Sobel operator (as described in Eqn. (7.10)) for practically any type of image. It can be invoked via the

Process→Find Edges

menu and is also available through the method `void findEdges()` for objects of type `ImageProcessor`.

7.4 Other Edge Operators

One problem with edge operators based on first derivatives (as described in the previous section) is that each resulting edge is as wide as the underlying intensity transition and thus edges may be difficult to localize precisely. An alternative class of edge operators makes use of the second derivatives of the image function, including some popular modern edge operators that also address the problem of edges appearing at various levels of scale. These issues are briefly discussed in the following.

case horizontally to the left) completely (that is, until it reaches the edge of the region) and only then examines the remaining directions. In contrast the *breadth-first* method markings proceed outward, step by step, equally in all directions.

Due to the way exploration takes place, the memory requirement of the *breadth-first* variant of the *flood-fill* version is generally much lower than that of the *depth-first* variant. For example, when flood filling the region in Fig. 11.2 using the implementation given (Prog. 11.1), the stack in the *depth-first* variant grows to a maximum of 28,822 elements, while the queue used by the *breadth-first* variant never exceeds a maximum of 438 nodes.

11.1.2 Sequential Region Labeling

Sequential region marking is a classical, nonrecursive technique that is known in the literature as "region labeling". The algorithm consists in essence of two steps: (1) a preliminary labeling of the image regions and (2) resolving cases where more than one label occurs (i. e., has been assigned in the previous step) in the same region. Even though this algorithm is relatively complex, especially the second stage, its moderate memory requirements have made it the method of choice in practice over other simpler methods. The entire process is summarized in Alg. 11.2.

Step 1: Preliminary labeling

In the first stage of region labeling, the image is traversed from top left to bottom right sequentially to assign a preliminary label to every foreground pixel. Depending on the definition of neighborhood (either 4- or 8-connected) used, the vicinity of each pixel must be examined ($\times$ marks the actual pixel at the position (u, v)):

$$\mathcal{N}_4(u,v) = \begin{array}{|c|c|c|}\hline & N_2 & \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array} \quad \text{or} \quad \mathcal{N}_8(u,v) = \begin{array}{|c|c|c|}\hline N_2 & N_3 & N_4 \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array}$$

When using the 4-connected neighborhood $\mathcal{N}_4$, only the two neighbors $N_1 = I(u-1, v)$ and $N_2 = I(u, v-1)$ need to be considered, but when using the 8-connected neighborhood $\mathcal{N}_8$, all four neighbors $N_1 \ldots N_4$ must be examined. In the following example, we will use an 8-connected neighborhood and the image from Fig. 11.3 (a).

Propagating labels

Again we assume that, in the image, the value $I(u, v) = 0$ represents background pixels and the value $I(u, v) = 1$ represents foreground pixels. We will also consider neighboring pixels that lie outside of the image matrix (e. g., on the array borders) to be part of the background. The

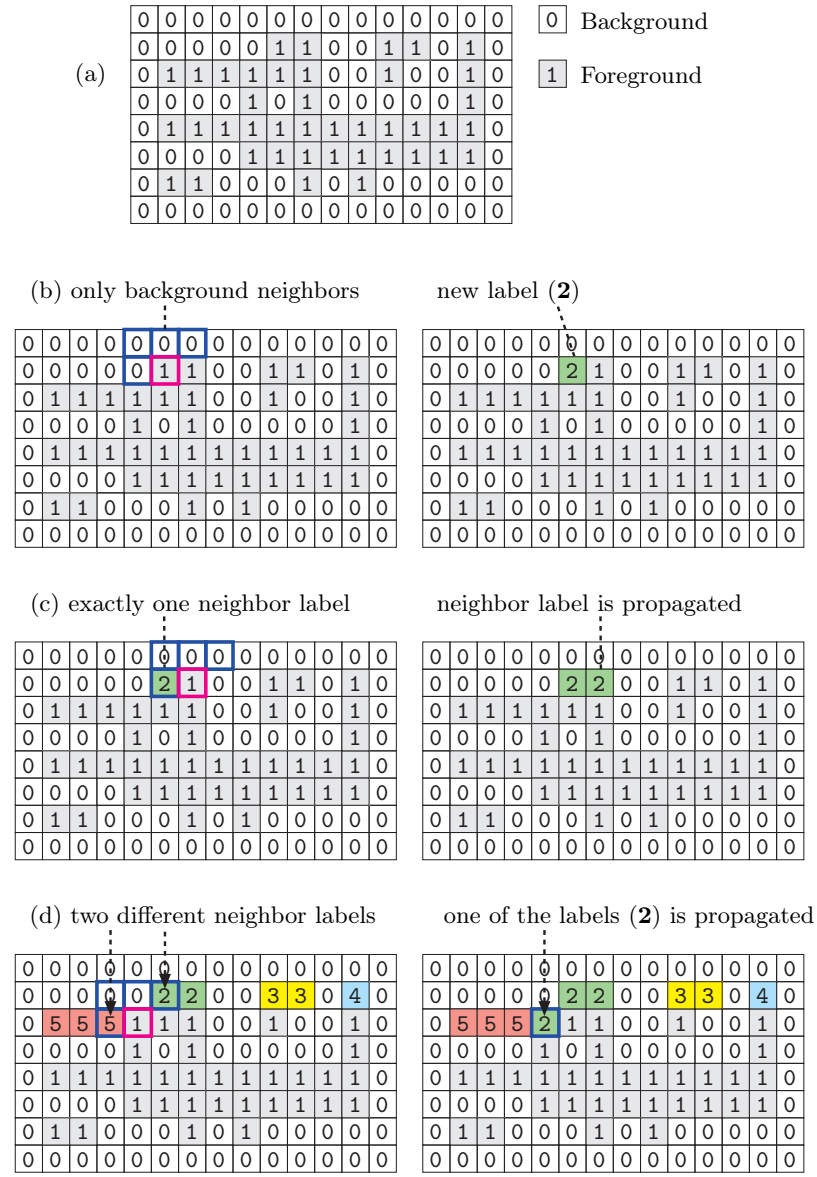


Fig. 11.3
Sequential region labeling—label propagation. Original image (a). The first foreground pixel [**1**] is found in (b): all neighbors are background pixels [**0**], and the pixel is assigned the first label [**2**]. In the next step (c), there is exactly *one* neighbor pixel marked with the label **2**, so this value is propagated. In (d) there are *two* neighboring pixels, and they have differing labels (**2** and **5**); one of these values is propagated, and the collision $\langle \mathbf{2}, \mathbf{5} \rangle$ is registered.

"nodes" of the graph and the registered collisions $\mathcal{C}$ make up its "edges" (Fig. 11.4 (b)).

Once all the distinct labels within a single region have been collected, the labels of all the pixels in the region are updated so they have the same value (for example, using the smallest original label in the region) as in Fig. 11.5.

11.2.2 Combining Region Labeling and Contour Finding

This method, based on [23], combines the concepts of sequential region labeling (Sec. 11.1) and traditional contour tracing into a single algorithm able to perform both tasks simultaneously during a single pass through the image. It identifies and labels regions and at the same time traces both their inner and outer contours. The algorithm does not require complicated data structures and is very efficient when compared with other methods with similar capabilities.

We now sketch the fundamental idea of the algorithm. While the main idea of the algorithm can be sketched out in a few simple steps, the actual implementation requires attention to a number of details, so we have provided the complete Java source for an ImageJ plugin implementation in Appendix D (pp. 532–542). The most important steps of the method are illustrated in Fig. 11.9:

1. As in the sequential region labeling (Alg. 11.2), the binary image I is traversed from the top left to the bottom right. Such a traversal ensures that all pixels in the image are eventually examined and assigned an appropriate label.
2. At a given position in the image, the following cases may occur:

   Case A: The transition from a background pixel to a previously unmarked foreground pixel A means that A lies on the outer edge of a new region. A new *label* is allocated and the associated *outer* contour is traversed and marked by calling the method TraceContour() (see Fig. 11.9 (a) and Alg. 11.3 (line 20)). Furthermore, all background pixels directly bordering the region are marked with the value -1.

   Case B: The transition from a foreground pixel B to an unmarked background pixel means that B lies on the edge of an *inner* contour (Fig. 11.9 (b)). Starting from B, the inner contour is traversed and its pixels are labeled with labels from the surrounding region (Fig. 11.9 (c)). Also, all bordering background pixels are again assigned the value of -1.

   Case C: When a foreground pixel does not lie on a contour (i. e., it is not on an edge), then the neighboring pixel to the left has already been labeled (Fig. 11.9 (d)) and this label is propagated to the current pixel.

In Algs. 11.3 and 11.4, the entire procedure is presented again and explained precisely. The method CombinedContourLabeling() traverses the image line-by-line and calls the method TraceContour() whenever a new inner or outer contour must be traced. The labels of the image elements along the contour, as well as the neighboring foreground pixels, are stored in the "label map" LM by the method FindNextPoint() (Alg. 11.4).

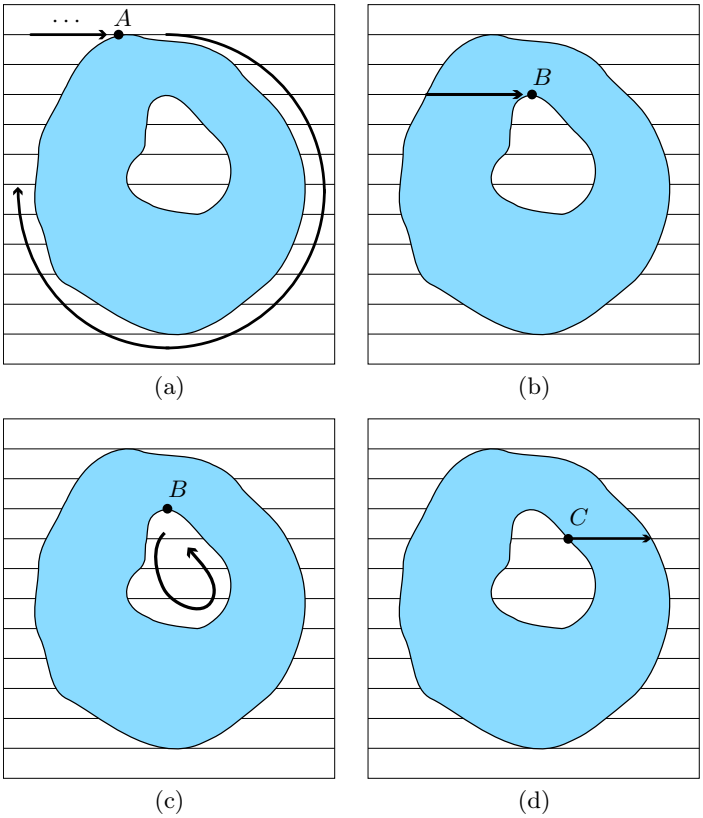


Fig. 11.9
Combined region labeling and contour following (after [23]). The image is traversed from the top left to the lower right a row at a time. In (a), the first point A on the outer edge of the region is found. Starting from point A, the pixels on the edge along the outer contour are visited and labeled until A is reached again. In (b), the first point B on an inner contour is found. The pixels along the inner contour are visited and labeled until arriving back at B (c). In (d), an already labeled point C on an inner contour is found. Its label is propagated along the image row within the region.

11.2.3 Implementation

The complete implementation of the algorithm in Java (ImageJ) can be found in Appendix D (beginning on page 532). The implementation closely follows the description in Algs. 11.3 and 11.4 but illustrates several additional details:[4]

- First the image I (`pixelArray`) and the associated label map LM (`labelArray`) are enlarged by adding one pixel around the borders. The new pixels are marked as *background* (0) in the image I. This simplifies contour following and eliminates the need to handle a number of special situations.
- As contours are found they are stored in an object of the class `ContourSet`, separated into outer and inner contours. The contours themselves are represented by the classes `OuterContour` and `InnerContour`, with a common superclass `Contour`. Every contour consists of an ordered sequence of coordinate points of the class `Node`

[4] In the following description the names in parentheses after the algorithmic symbols denote the corresponding identifiers used in the Java implementation.

Algorithm 11.3
Combined contour tracing and region labeling. Given a binary image I, the method CombinedContourLabeling() returns a set of contours and an array containing region labels for all pixels in the image. When a new point on either an outer or inner contour is found, then an ordered list of the contour's points is constructed by calling the method TraceContour() (line 20 and line 27). TraceContour() itself is described in Alg. 11.4.

```
1:  CombinedContourLabeling (I)
        I: binary image
        Returns a set of contours and a label map (labeled image).
2:      Create an empty set of contours: C ← {}
3:      Create a label map LM of the same size as I and initialize:
4:      for all (u, v) do
5:          LM(u, v) ← 0                                  ▷ label map LM
6:      R ← 0                                             ▷ region counter R
7:      Scan the image from left to right and top to bottom:
8:      for v ← 0 ... N−1 do
9:          L_k ← 0                                       ▷ current label L_k
10:         for u ← 0 ... M−1 do
11:             if I(u, v) is a foreground pixel then
12:                 if (L_k ≠ 0) then                     ▷ continue existing region
13:                     LM(u, v) ← L_k
14:                 else
15:                     L_k ← LM(u, v)
16:                     if (L_k = 0) then                 ▷ hit new outer contour
17:                         R ← R + 1
18:                         L_k ← R
19:                         x_S ← (u, v)
20:                         c_outer ← TraceContour(x_S, 0, L_k, I, LM)
21:                         C ← C ∪ {c_outer}             ▷ collect new contour
22:                         LM(u, v) ← L_k
23:             else                                      ▷ I(u, v) is a background pixel
24:                 if (L_k ≠ 0) then
25:                     if (LM(u, v) = 0) then            ▷ hit new inner contour
26:                         x_S ← (u−1, v)
27:                         c_inner ← TraceContour(x_S, 1, L_k, I, LM)
28:                         C ← C ∪ {c_inner}             ▷ collect new contour
29:                     L_k ← 0
30:     return (C, LM).                                   ▷ return the set of contours and the label map
```

continued in Alg. 11.4 ▷▷

(defined on p. 203). The Java container class `ArrayList` (templated on the type `Node`) is used as a dynamic data structure for storing the point sequences of the outer and inner contours.

- The method `traceContour()` (see p. 538) traverses an outer or inner contour, beginning from the starting point $\boldsymbol{x}_S$ (`xS`, `yS`). It calls the method `findNextPoint()`, to determine the next contour point $\boldsymbol{x}_T$ (`xT`, `yT`) following $\boldsymbol{x}_S$:
  - In the case that no following point is found, then $\boldsymbol{x}_S = \boldsymbol{x}_T$ and the region (contour) consists of a single isolated pixel. The method `traceContour()` is finished.
  - In the other case the remaining contour points are found by repeatedly calling `findNextPoint()`, and for every successive pair of points the *current* point $\boldsymbol{x}_c$ (`xC`, `yC`) and the *previous* point $\boldsymbol{x}_p$

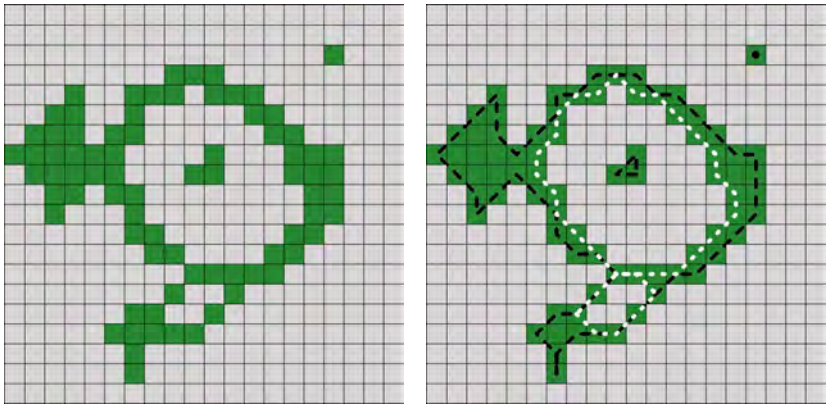
Algorithm 11.4
Combined contour finding and region labeling (continued from Alg. 11.3). Starting from $\boldsymbol{x}_S$, the procedure TraceContour traces along the contour in the direction $d_S = 0$ for outer contours or $d_S = 1$ for inner contours. During this process, all contour points as well as neighboring background points are marked in the label array LM. Given a point $\boldsymbol{x}_c$, TraceContour uses FindNextPoint() to determine the next point along the contour (line 10). The function Delta() returns the next coordinate in the sequence, taking into account the search direction d.

```
1:  TraceContour(x_S, d_S, L_c, I, LM)
        x_S: start position, d_S: initial search direction,
        L_c: label for this contour
        I: original image, LM: label map.
        Traces and returns the contour starting at x_S.
2:      (x_T, d_next) ← FindNextPoint(x_S, d_S, I, LM)
3:      c ← [x_T]                          ▷ create a contour starting with x_T
4:      x_p ← x_S                          ▷ previous position x_p = (u_p, v_p)
5:      x_c ← x_T                          ▷ current position x_c = (u_c, v_c)
6:      done ← (x_S ≡ x_T)                 ▷ isolated pixel?
7:      while (¬done) do
8:          LM(u_c, v_c) ← L_c
9:          d_search ← (d_next + 6) mod 8
10:         (x_n, d_next) ← FindNextPoint(x_c, d_search, I, LM)
11:         x_p ← x_c
12:         x_c ← x_n
13:         done ← (x_p ≡ x_S ∧ x_c ≡ x_T)   ▷ back at start point?
14:         if (¬done) then
15:             Append(c, x_n)             ▷ add point x_n to contour c
16:     return c.                          ▷ return this contour

17: FindNextPoint(x_c, d, I, LM)
        x_c: start point, d: search direction,
        I: original image, LM: label map.
18:     for i ← 0 ... 6 do                 ▷ search in 7 directions
19:         x' ← x_c + Delta(d)            ▷ x' = (u', v')
20:         if I(u', v') is a background pixel then
21:             LM(u', v') ← −1            ▷ mark background as visited (−1)
22:             d ← (d + 1) mod 8
23:         else                           ▷ found a nonbackground pixel at x'
24:             return (x', d)
25:     return (x_c, d).                   ▷ found no next point, return start point

26: Delta(d) = (Δx, Δy),   with
```

d	0	1	2	3	4	5	6	7
Δx	1	1	0	−1	−1	−1	0	1
Δy	0	1	1	1	0	−1	−1	−1

(`xP`, `yP`) are recorded. Only when *both* points correspond to the original starting points on the contour, $\boldsymbol{x}_p = \boldsymbol{x}_S$ and $\boldsymbol{x}_c = \boldsymbol{x}_T$, we know that the contour has been completely traversed.

- The method `findNextPoint()` (see p. 539) determines which point on the contour follows the current point $\boldsymbol{x}_c$ (`pt`) by searching in the *direction* d (`dir`), depending upon the position of the previous contour point. Starting in the first search direction, up to seven neighboring pixels (all neighbors except the previous contour point) are searched in clockwise direction until the next contour point is found. At the same time, all background pixels in the *label map* LM (`labelArray`) are marked with the value -1 to prevent them from being searched again. If no valid contour point is found among the



(a) (b)

Fig. 11.10
Combined contour and region marking: original image in gray (a), located contours (b) with black lines for out and white lines for inner contours. Outer contours of one-pixel regions (for example, in the upper-right of (b)) are marked by a single dot.

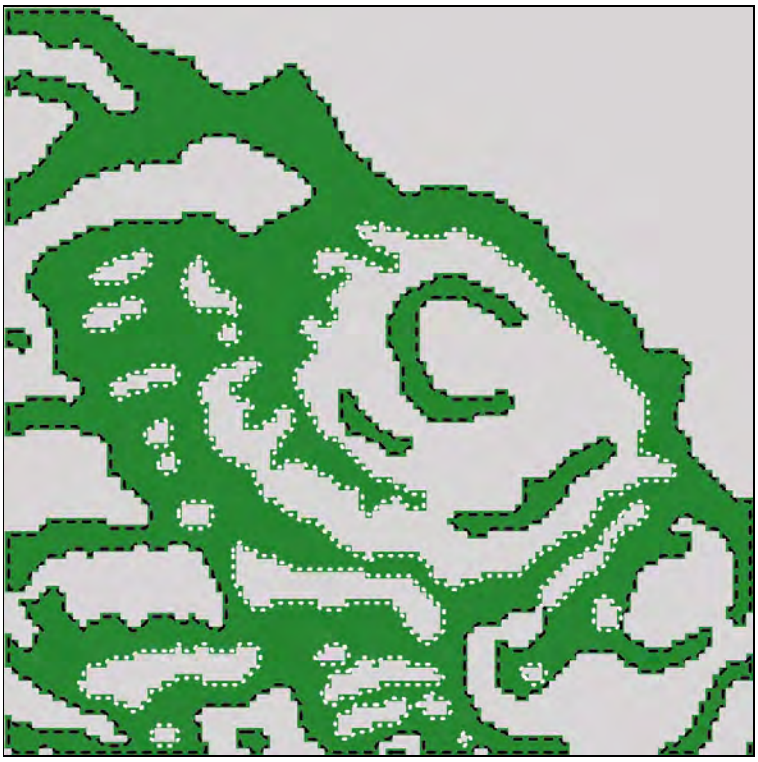


Fig. 11.11
Example of a complex contour (in a section cut from Fig. 10.12). Outer contours are marked in black and inner contours in white.

programming languages, to a simple and elegant mapping onto two-dimensional arrays, which makes possible a very natural way to work with raster images. One possible disadvantage with this representation is that it does not depend on the content of the image. In other words, it makes no difference whether the image contains only a pair of lines or is of a complex scene because the amount of memory required is constant and depends only on the dimensions of the image.

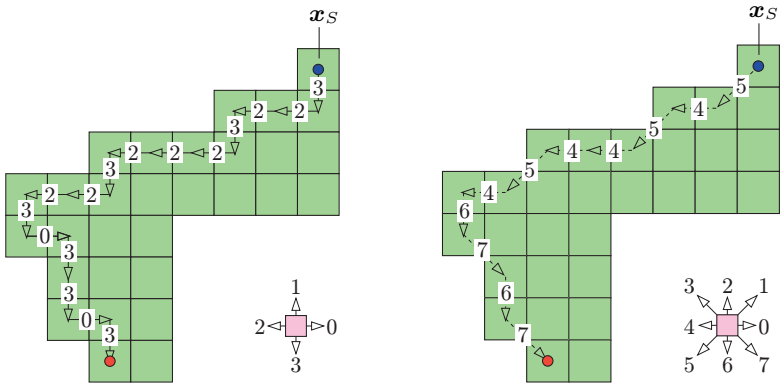


Fig. 11.14
Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point $\boldsymbol{x}_S$. Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) neighborhoods. The length of the resulting path, calculated as the sum of the individual segments, can be used to approximate the true length of the contour.

a number of other important codecs, including TIFF, GIF, and JPEG. In addition, RLE provides precomputed information about the image that can be used directly when computing certain properties of the image (for example, statistical moments; see Sec. 11.4.3).

11.3.3 Chain Codes

Regions can be represented not only using their interiors but also by their contours. Chain codes, which are often referred to as Freeman codes [35], are a classical method of contour encoding. In this encoding, the contour beginning at a given start point $\boldsymbol{x}_S$ is represented by the sequence of directional changes it describes on the discrete image raster (Fig. 11.14).

Absolute chain code

For a closed contour of a region $\mathcal{R}$, described by the sequence of points $\boldsymbol{c}_\mathcal{R} = [\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1}]$ with $\boldsymbol{x}_i = \langle u_i, v_i \rangle$, we create the elements of its chain code sequence $\boldsymbol{c}'_\mathcal{R} = [c'_0, c'_1, \ldots c'_{M-1}]$ by

$$c'_i = \textsc{Code}(\Delta u_i, \Delta v_i), \tag{11.1}$$

$$\text{where} \quad (\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \le i < M-1 \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases}$$

and $\textsc{Code}(\Delta u, \Delta v)$ being defined by the following table:[6]

[6] Assuming an 8-connected neighborhood.

A method that is often used [5,38] is to interpret the elements c''_i in the differential chain code as the digits of a number to the base b ($b = 8$ for an 8-connected contour or $b = 4$ for a 4-connected contour) and the numeric value

$$\begin{aligned}\textsc{Val}(\boldsymbol{c}''_{\mathcal{R}}) &= c''_0 \cdot b^0 + c''_1 \cdot b^1 + \ \ldots \ + c''_{M-1} \cdot b^{M-1} \\ &= \sum_{i=0}^{M-1} c''_i \cdot b^i. \end{aligned} \tag{11.3}$$

Then the sequence $\boldsymbol{c}''_{\mathcal{R}}$ is shifted cyclically until the numeric value of the corresponding number reaches a maximum. We use the expression $\boldsymbol{c}''_{\mathcal{R}} \rhd k$ to denote the sequence $\boldsymbol{c}''_{\mathcal{R}}$ being cyclically shifted by k positions to the right,[8] such as (for $k = 2$)

$$\begin{aligned}\boldsymbol{c}''_{\mathcal{R}} &= [\,0, 1, 3, 2, \ldots 5, 3, 7, 4\,] \\ \boldsymbol{c}''_{\mathcal{R}} \rhd 2 &= [\,7, 4, 0, 1, 3, 2, \ldots 5, 3\,]\end{aligned}$$

and

$$k_{\max} = \arg \max_{0 \le k < M} \textsc{Val}(\boldsymbol{c}''_{\mathcal{R}} \rhd k) \tag{11.4}$$

to denote the shift required to maximize the corresponding arithmetic value. The resulting code sequence or *shape number*,

$$\boldsymbol{s}_{\mathcal{R}} = \boldsymbol{c}''_{\mathcal{R}} \rhd k_{\max}, \tag{11.5}$$

is *normalized* with respect to the starting point and can thus be directly compared element by element with other normalized code sequences. Since the function Val() in Eqn. (11.3) produces values that are in general too large to be actually computed, in practice the relation

$$\textsc{Val}(\boldsymbol{c}''_1) > \textsc{Val}(\boldsymbol{c}''_2)$$

is determined by comparing the *lexicographic ordering* between the sequences $\boldsymbol{c}''_1$ and $\boldsymbol{c}''_2$ so that the arithmetic values need not be computed at all.

Unfortunately, comparisons based on chain codes are generally not very useful for determining the similarity between regions simply because rotations at arbitrary angles ($\neq 90°$) have too great of an impact (change) on a region's code. In addition, chain codes are not capable of handling changes in size (scaling) or other distortions. Section 11.4 presents a number of tools that are more appropriate in these types of cases.

Fourier descriptors

An elegant approach to describing contours are so-called Fourier descriptors, which interpret the two-dimensional contour $\boldsymbol{c}_{\mathcal{R}} = [\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots$

[8] $(\boldsymbol{c}''_{\mathcal{R}} \rhd k)[\,i\,] = \boldsymbol{c}''_{\mathcal{R}}[(i - k) \bmod M]$.

for the region that can be used for classification or comparison with other regions. The best features are those that are simple to calculate and are not easily influenced (robust) by irrelevant changes, particularly translation, rotation, and scaling.

11.4.2 Geometric Features

A region $\mathcal{R}$ of a binary image can be interpreted as a two-dimensional distribution of foreground points $\boldsymbol{x}_i = (u_i, v_i)$ within the discrete plane $\mathbb{Z}^2$,

$$\mathcal{R} = \{\boldsymbol{x}_0, \boldsymbol{x}_1 \ldots \boldsymbol{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1) \ldots (u_{N-1}, v_{N-1})\}.$$

Most geometric properties are defined in such a way that a region is considered to be a set of pixels that, in contrast to the definition in Sec. 11.1, does not necessarily have to be connected.

Perimeter

The perimeter (or circumference) of a region $\mathcal{R}$ is defined as the length of its outer contour, where $\mathcal{R}$ must be connected. As illustrated in Fig. 11.14, the type of neighborhood relation must be taken into account for this calculation. When using a 4-neighborhood, the measured length of the contour (except when that length is 1) will be larger than its actual length. In the case of 8-neighborhoods, a good approximation is reached by weighing vertical segments with 1 and diagonal segments with $\sqrt{2}$. Given an 8-connected chain code $\boldsymbol{c}'_{\mathcal{R}} = [c'_0, c'_1, \ldots c'_{M-1}]$, the perimeter of the region is arrived at by

$$\mathsf{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathrm{length}(c'_i) \tag{11.7}$$

$$\text{with} \quad \mathrm{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6, \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7. \end{cases}$$

However, with this conventional method of calculation,[9] the *real* perimeter ($P(\mathcal{R})$) is systematically overestimated. As a simple remedy, a general correction factor of 0.95 works satisfactory even for relatively small regions:

$$P(\mathcal{R}) \approx \mathsf{Perimeter}_{\mathrm{corr}}(\mathcal{R}) = 0.95 \cdot \mathsf{Perimeter}(\mathcal{R}). \tag{11.8}$$

[9] Note that the tools in ImageJ's Analyze→Measure menu use a different approach for computing a region's perimeter than the one described here.

Program 12.4
Converting an indexed image to a true color RGB image (ImageJ plugin).

```
1  // File Index_To_Rgb.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ColorProcessor;
6  import ij.process.ImageProcessor;
7  import java.awt.image.IndexColorModel;
8
9  public class Index_To_Rgb implements PlugInFilter {
10   static final int R = 0, G = 1, B = 2;
11
12   public int setup(String arg, ImagePlus im) {
13     return DOES_8C + NO_CHANGES; //does not alter original image
14   }
15
16   public void run(ImageProcessor ip) {
17     int w = ip.getWidth();
18     int h = ip.getHeight();
19
20     //retrieve the color table (palette) for R,G,B
21     IndexColorModel icm =
22         (IndexColorModel) ip.getColorModel();
23     int mapSize = icm.getMapSize();
24     byte[] Rmap = new byte[mapSize]; icm.getReds(Rmap);
25     byte[] Gmap = new byte[mapSize]; icm.getGreens(Gmap);
26     byte[] Bmap = new byte[mapSize]; icm.getBlues(Bmap);
27
28     //create new 24-bit RGB image
29     ColorProcessor cp = new ColorProcessor(w,h);
30     int[] RGB = new int[3];
31     for (int v = 0; v < h; v++) {
32       for (int u = 0; u < w; u++) {
33         int idx = ip.getPixel(u, v);
34         RGB[R] = 0xFF & Rmap[idx]; // treat maps as
35         RGB[G] = 0xFF & Gmap[idx]; // UNSIGNED byte!
36         RGB[B] = 0xFF & Bmap[idx];
37         cp.putPixel(u, v, RGB);  // putPixel() instead of set()
38       }
39     }
40     ImagePlus cimg = new ImagePlus("RGB Image",cp);
41     cimg.show();
42   }
43
44 } // end of class Index_To_Rgb
```

Creating indexed images

In ImageJ, no special method is provided for the creation of indexed images, so in almost all cases they are generated by converting an existing

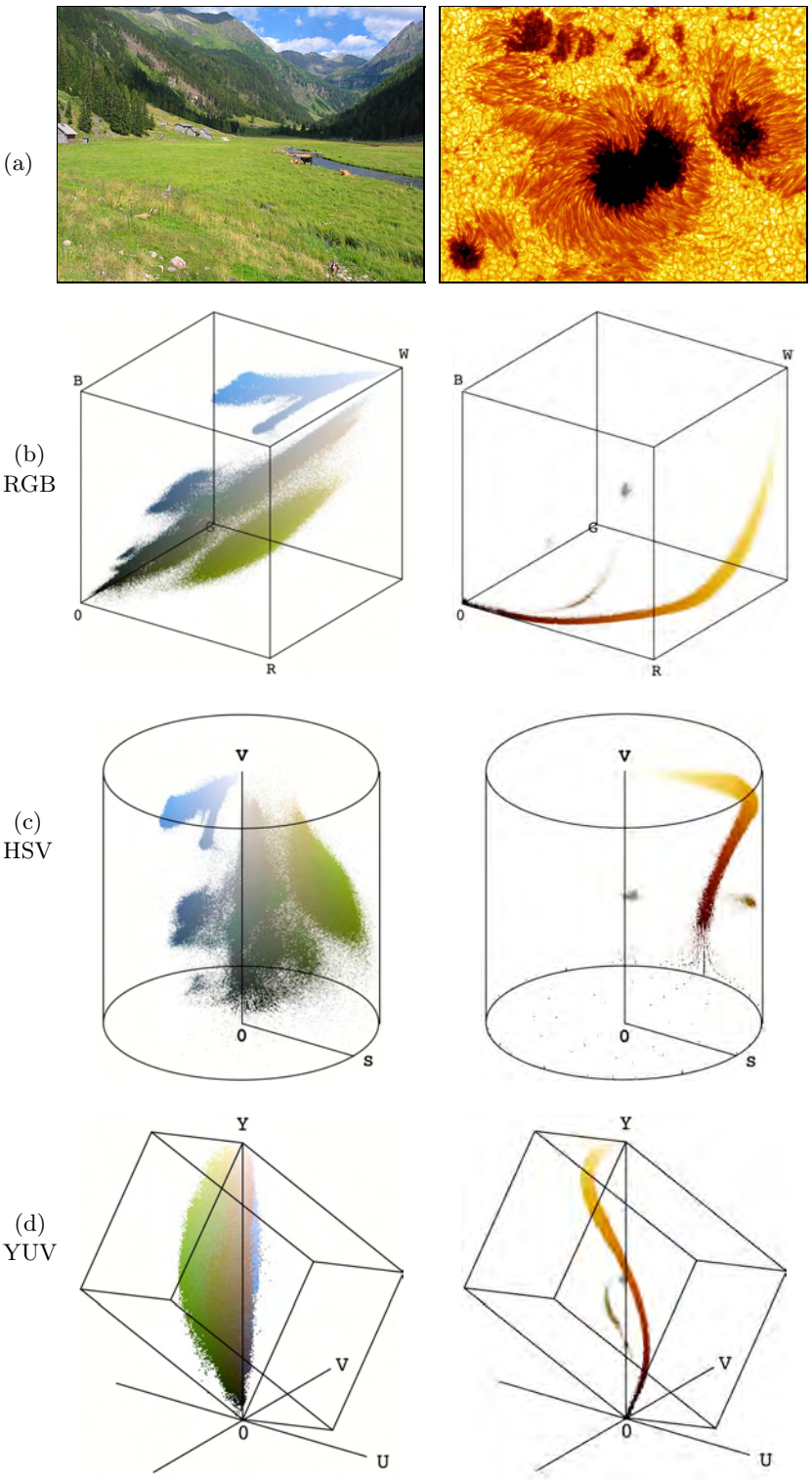


Fig. 12.9
Examples of the color distribution of natural images in three different color spaces. Original images: landscape photograph with dominant green and blue components and sun-spot image with rich red and yellow components (a). Color distribution in RGB- (b), HSV- (c), and YUV-space (d).

base between the two pyramids. Even though it is often portrayed in this intuitive way, mathematically the HLS space is again a cylinder (see Fig. 12.15).

RGB→HSV

To convert from RGB to the HSV color space, we first find the *saturation* of the RGB color components $R, G, B \in [0, C_{\max}]$, with $C_{\max}$ being the maximum component value (typically 255), as

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{for } C_{\text{high}} > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{12.10}$$

and the luminance (*value*)

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\max}}\,, \tag{12.11}$$

with C_{high}, C_{low}, and C_{rng} defined as

$$\begin{aligned} C_{\text{high}} &= \max(R, G, B), \quad C_{\text{low}} = \min(R, G, B), \quad \text{and} \\ C_{\text{rng}} &= C_{\text{high}} - C_{\text{low}}. \end{aligned} \tag{12.12}$$

Finally, we need to specify the *hue* value H_{HSV}. When all three RGB color components have the same value ($R = G = B$), then we are dealing with an *achromatic* (gray) pixel. In this particular case $C_{\text{rng}} = 0$ and thus the saturation value $S_{\text{HSV}} = 0$, consequently the hue is undefined. To compute H_{HSV} when $C_{\text{rng}} > 0$, we first normalize each component using

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}}, \qquad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}}, \qquad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}}. \tag{12.13}$$

Then, depending on which of the three original color components had the maximal value, we compute a preliminary hue H' as

$$H' = \begin{cases} B' - G' & \text{if } R = C_{\text{high}} \\ R' - B' + 2 & \text{if } G = C_{\text{high}} \\ G' - R' + 4 & \text{if } B = C_{\text{high}}. \end{cases} \tag{12.14}$$

Since the resulting value for H' lies on the interval $[-1 \ldots 5]$, we obtain the final hue value by normalizing to the interval $[0, 1]$ as

$$H_{\text{HSV}} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{for } H' < 0 \\ H' & \text{otherwise.} \end{cases} \tag{12.15}$$

Hence all three components H_{HSV}, S_{HSV}, and V_{HSV} will lie within the interval $[0, 1]$. The hue value H_{HSV} can naturally also be computed in another angle interval, for example in the 0 to 360° interval using

array with the resulting H, S, V values in the interval $[0, 1]$. When an existing `float` array is passed as the argument ***hsv***, then the result is placed in it; otherwise (when ***hsv*** = `null`) a new array is created. Here is a simple usage example:

```
1 import java.awt.Color;
2 ...
3 float[] hsv = new float[3];
4 int red = 128, green = 255, blue = 0;
5 hsv = Color.RGBtoHSB (red, green, blue, hsv);
6 float h = hsv[0];
7 float s = hsv[1];
8 float v = hsv[2];
9 ...
```

A possible implementation of the Java method `RGBtoHSB()` using the definition in Eqns. (12.11)–(12.15) is given in Prog. 12.6.

HSV→RGB

To convert an HSV tuple $(H_{\text{HSV}}, S_{\text{HSV}}, V_{\text{HSV}})$, where H_{HSV}, S_{HSV}, and $V_{\text{HSV}} \in [0, 1]$, into the corresponding (R, G, B) color values, the appropriate color sector

$$H' = (6 \cdot H_{\text{HSV}}) \bmod 6 \tag{12.16}$$

$(0 \le H' < 6)$ is determined first, followed by computing the intermediate values

$$\begin{aligned} c_1 &= \lfloor H' \rfloor, & x &= (1 - S_{\text{HSV}}) \cdot V_{\text{HSV}}, \\ c_2 &= H' - c_1, & y &= (1 - (S_{\text{HSV}} \cdot c_2)) \cdot V_{\text{HSV}}, \\ & & z &= (1 - (S_{\text{HSV}} \cdot (1 - c_2))) \cdot V_{\text{HSV}}. \end{aligned} \tag{12.17}$$

Depending on the value of c_1, the normalized RGB values $R', G', B' \in [0, 1]$ are then computed from $v = V_{\text{HSV}}$, x, y, and z as follows:[8]

$$(R', G', B') = \begin{cases} (v, z, x) & \text{if } c_1 = 0 \\ (y, v, x) & \text{if } c_1 = 1 \\ (x, v, z) & \text{if } c_1 = 2 \\ (x, y, v) & \text{if } c_1 = 3 \\ (z, x, v) & \text{if } c_1 = 4 \\ (v, x, y) & \text{if } c_1 = 5. \end{cases} \tag{12.18}$$

The normalized RGB components $R', G', B' \in [0, 1]$ are scaled back to integer values R, G, B in the range $[0, 255]$ as follows:

$$\begin{aligned} R &= \min\big(\text{round}(255 \cdot R'), 255\big), \\ G &= \min\big(\text{round}(255 \cdot G'), 255\big), \\ B &= \min\big(\text{round}(255 \cdot B'), 255\big). \end{aligned} \tag{12.19}$$

[8] The variables x, y, z used here have no relation to those used in the CIEXYZ color space (Sec. 12.3.1).

Program 12.7
HSV→RGB conversion. This Java method takes the same arguments and returns identical results as the standard method `Color.HSBtoRGB()`. Lines 18–20 changed!

```
1  public static int HSVtoRGB (float h, float s, float v) {
2    // h, s, v ∈ [0,1]
3    float r = 0, g = 0, b = 0;
4    float hh = (6 * h) % 6;          // h' ← (6 · h) mod 6
5    int   c1 = (int) hh;             // c1 ← ⌊h'⌋
6    float c2 = hh - c1;
7    float x = (1 - s) * v;
8    float y = (1 - (s * c2)) * v;
9    float z = (1 - (s * (1 - c2))) * v;
10   switch (c1) {
11     case 0: r = v; g = z; b = x; break;
12     case 1: r = y; g = v; b = x; break;
13     case 2: r = x; g = v; b = z; break;
14     case 3: r = x; g = y; b = v; break;
15     case 4: r = z; g = x; b = v; break;
16     case 5: r = v; g = x; b = y; break;
17   }
18   int R = Math.min((int)(r * 255), 255);
19   int G = Math.min((int)(g * 255), 255); // Eqn. (12.19)
20   int B = Math.min((int)(b * 255), 255);
21   // create int-packed RGB-color:
22   int rgb = ((R & 0xff)<<16) | ((G & 0xff)< 8) | B & 0xff;
23   return rgb;
24 }
```

RGB→HLS

In the HLS model, the *hue* value H_{HLS} is computed in the same way as in the HSV model (Eqns. (12.13)–(12.15)), i. e.,

$$H_{\text{HLS}} = H_{\text{HSV}}. \tag{12.20}$$

The other values, L_{HLS} and S_{HLS}, are computed as follows (with C_{high}, C_{low}, $C_{\text{rng}} \in [0, 255]$, as defined in Eqn. (12.12)):

$$L_{\text{HLS}} = \frac{(C_{\text{high}} + C_{\text{low}})/255}{2}, \tag{12.21}$$

$$S_{\text{HLS}} = \begin{cases} 0 & \text{for } L_{\text{HLS}} = 0 \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{L_{\text{HLS}}} & \text{for } 0 < L_{\text{HLS}} \le 0.5 \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{1-L_{\text{HLS}}} & \text{for } 0.5 < L_{\text{HLS}} < 1 \\ 0 & \text{for } L_{\text{HLS}} = 1. \end{cases} \tag{12.22}$$

Figure 12.14 shows the individual HLS components of the test image as grayscale images. Using the above definitions, the unit cube in the RGB space is again mapped to a cylinder with height and length 1 (Fig. 12.15).

$$
\begin{aligned}
c_1 &= \lfloor H' \rfloor & d &= \begin{cases} S_{\text{HLS}} \cdot L_{\text{HLS}} & \text{for } L_{\text{HLS}} \le 0.5 \\ S_{\text{HLS}} \cdot (1 - L_{\text{HLS}}) & \text{for } L_{\text{HLS}} > 0.5 \end{cases} \\
c_2 &= H' - c_1 & & \\
w &= L_{\text{HLS}} + d & y &= w - (w - x) \cdot c_2 \\
x &= L_{\text{HLS}} - d & z &= x + (w - x) \cdot c_2.
\end{aligned}
\tag{12.25}
$$

The assignment of the RGB values is done similarly to Eqn. (12.18), i. e.,

$$
(R', G', B') = \begin{cases} (w, z, x) & \text{if } c_1 = 0 \\ (y, w, x) & \text{if } c_1 = 1 \\ (x, w, z) & \text{if } c_1 = 2 \\ (x, y, w) & \text{if } c_1 = 3 \\ (z, x, w) & \text{if } c_1 = 4 \\ (w, x, y) & \text{if } c_1 = 5. \end{cases}
\tag{12.26}
$$

Finally, the normalized color components $R', G', B' \in [0, 1]$ are scaled back to the $[0, 255]$ integer range as described in Eqn. (12.19).

Java implementation (RGB↔HLS)

Currently there is no method in either the standard Java API or ImageJ for converting color values between RGB and HLS. Program 12.8 gives one possible implementation of the RGB→HLS conversion that follows the definitions in Eqns. (12.20)–(12.22). The HLS→RGB conversion is given in Prog. 12.9.

Comparing HSV and HLS

Despite the gross similarity between the two color spaces, as Fig. 12.16 illustrates, substantial differences in the V/L and S components do exist. The essential difference between the HSV and HLS spaces is the ordering of the colors that lie between the white point **W** and the "pure" colors (**R**, **G**, **B**, **Y**, **C**, **M**), which consist of at most two primary colors, at least one of which is completely saturated.

The difference in how colors are distributed in RGB, HSV, and HLS space is readily apparent in Fig. 12.17. The starting point was a distribution of 1331 ($11 \times 11 \times 11$) color tuples obtained by uniformly sampling the RGB space at an interval of 0.1 in each dimension.

Both the HSV and HLS color spaces are widely used in practice; for instance, for selecting colors in image editing and graphics design applications. In digital image processing, they are also used for *color keying* (that is, isolating objects according to their *hue*) on a homogeneously colored background where the brightness is not necessarily constant.

Program 12.8
RGB→HLS conversion (Java method).

```
1  static float[] RGBtoHLS (int R, int G, int B) {
2    // R, G, B assumed to be in [0,255]
3    int cHi = Math.max(R, Math.max(G, B)); // highest color value
4    int cLo = Math.min(R, Math.min(G, B)); // lowest color value
5    int cRng = cHi - cLo;          // color range
6
7    // Calculate hue H (same as in HSV):
8    float H = 0;
9
10   if (cHi > 0 && cRng > 0) {     // a color pixel
11     float r = (float)(cHi - R) / cRng;
12     float g = (float)(cHi - G) / cRng;
13     float b = (float)(cHi - B) / cRng;
14     float h;
15     if (R == cHi)                  // R is largest component
16       h = b - g;
17     else if (G == cHi)             // G is largest component
18       h = r - b + 2.0f;
19     else                           // B is largest component
20       h = g - r + 4.0f;
21     if (h < 0)
22       h = h + 6;
23     H = h / 6;
24   }
25
26   // Calculate lightness L
27   float L = ((cHi + cLo) / 255f) / 2; // Eqn. (12.21)
28
29   // Calculate saturation S
30   float S = 0;
31   if (0 < L && L < 1) {
32     float d = (L <= 0.5f) ? L : (1 - L);
33     S = 0.5f * (cRng / 255f) / d;    // Eqn. (12.22)
34   }
35
36   return new float[] { H, L, S };
37 }
```

12.2.4 TV Color Spaces—YUV, YIQ, and YC_bC_r

These color spaces are an integral part of the standards surrounding the recording, storage, transmission, and display of television signals. YUV and YIQ are the fundamental color-encoding methods for the analog NTSC and PAL systems, and YC_bC_r is a part of the international standards governing digital television [45]. All of these color spaces have in common the idea of separating the luminance component Y from two chroma components and, instead of directly encoding colors, encoding color differences. In this way, compatibility with legacy black and white systems is maintained while at the same time the bandwidth of the sig-

Program 12.9 HLS→RGB conversion (Java method). Lines 24–28 changed!

```
1  static int HLStoRGB (float H, float L, float S) {
2    // H, L, S assumed to be in [0, 1]
3    float r = 0, g = 0, b = 0;
4    if (L <= 0)      // black
5      r = g = b = 0;
6    else if (L >= 1)  // white
7      r = g = b = 1;
8    else {
9      float hh = (6 * H) % 6;
10     int  c1 = (int) hh;
11     float c2 = hh - c1;
12     float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
13     float w = L + d, x = L - d;
14     float y = w - (w - x) * c2, z = x + (w - x) * c2;
15     switch (c1) {
16       case 0: r=w; g=z; b=x; break;
17       case 1: r=y; g=w; b=x; break;
18       case 2: r=x; g=w; b=z; break;
19       case 3: r=x; g=y; b=w; break;
20       case 4: r=z; g=x; b=w; break;
21       case 5: r=w; g=x; b=y; break;
22     }
23   }
24   int R = Math.min(Math.round(r * 255), 255);
25   int G = Math.min(Math.round(g * 255), 255); // Eqn. (12.19)
26   int B = Math.min(Math.round(b * 255), 255);
27   int rgb = ((R&0xff)<<16) | ((G&0xff)<<8) | B&0xff;
28   return rgb;
29 }
```

nal can be optimized by using different transmission bandwidths for the brightness and the color components. Since the human visual system is not able to perceive detail in the color components as well as it does in the intensity part of a video signal, the amount of information, and consequently bandwidth, used in the color channel can be reduced to approximately 1/4 of that used for the intensity component. This fact is also used when compressing digital still images and is why, for example, the JPEG codec converts RGB images to YC_bC_r. That is why these color spaces are important in digital image processing, even though raw YIQ or YUV images are rarely encountered in practice.

YUV

YUV is the basis for the color encoding used in analog television in both the North American NTSC and the European PAL systems. The luminance component Y is computed, just as in Eqn. (12.6), from the RGB components as

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{12.27}$$

$$
\begin{aligned}
Y &= w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B,\\
C_b &= \frac{0.5}{1 - w_B} \cdot (B - Y),\\
C_r &= \frac{0.5}{1 - w_R} \cdot (R - Y),
\end{aligned} \tag{12.32}
$$

and the inverse transformation from $\mathrm{YC}_b\mathrm{C}_r$ to RGB is

$$
\begin{aligned}
R &= Y + \frac{1 - w_R}{0.5} \cdot C_r,\\
G &= Y - \frac{w_B \cdot (1 - w_B) \cdot C_b + w_R \cdot (1 - w_R) \cdot C_r}{0.5 \cdot (1 - w_B - w_R)},\\
B &= Y + \frac{1 - w_B}{0.5} \cdot C_b.
\end{aligned} \tag{12.33}
$$

The ITU[9] recommendation BT.601 [56] specifies the values $w_R = 0.299$ and $w_B = 0.114$ ($w_G = 1 - w_B - w_R = 0.587$). Using these values, the transformation becomes

$$
\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{12.34}
$$

and the inverse transformation becomes

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}. \tag{12.35}
$$

Different weights are recommended based on how the color space is used; for example, ITU-BT.709 [55] recommends $w_R = 0.2125$ and $w_B = 0.0721$ to be used in digital HDTV production. The values of U, V, I, Q, and C_b, C_r may be both positive or negative. To encode C_b, C_r values to digital numbers, a suitable offset is typically added to obtain positive-only values, e. g., $128 = 2^7$ in case of 8-bit components.

Figure 12.18 shows the three color spaces YUV, YIQ, and $\mathrm{YC}_b\mathrm{C}_r$ together for comparison. The U, V, I, Q, and C_b, C_r values in the right two frames have been offset by 128 so that the negative values are visible. Thus a value of zero is represented as medium gray in these images. The $\mathrm{YC}_b\mathrm{C}_r$ encoding is practically indistinguishable from YUV in these images since they both use very similar weights for the color components.

12.2.5 Color Spaces for Printing—CMY and CMYK

In contrast to the *additive* RGB color scheme (and its various color models), color printing makes use of a *subtractive* color scheme, where

[9] International Telecommunication Union (www.itu.int).

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. (13.20)) and the inverse transformation $\mathcal{F}^{-1}$ (Eqn. (13.21)) 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.

13.1.5 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 $G(\omega)$ constitute a "transform pair",

$$g(x) \circ\!\!-\!\!\bullet\, G(\omega).$$

Table 13.1 lists the transform pairs for some selected analytical functions, which are also shown graphically in Figs. 13.3 and 13.4.

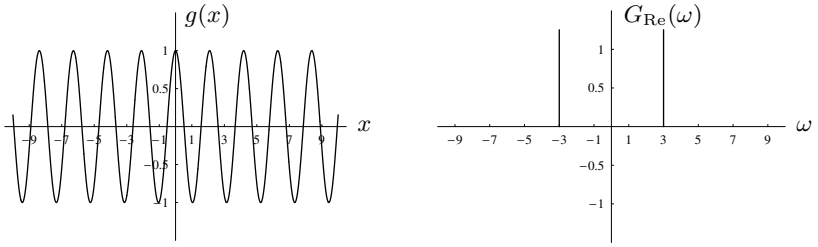
Table 13.1
Fourier transforms of selected analytical functions; $\delta()$ denotes the "impulse" or *Dirac* function (see Sec. 13.2.1).

Function	*Transform Pair* $g(x) \circ\!\!-\!\!\bullet\, G(\omega)$	*Figure*
Cosine function with frequency ω_0	$g(x) = \cos(\omega_0 x)$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}\omega_0) + \delta(\omega{-}\omega_0)\big)$	13.3 (a, c)
Sine function with frequency ω_0	$g(x) = \sin(\omega_0 x)$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}\omega_0) - \delta(\omega{-}\omega_0)\big)$	13.3 (b, d)
Gaussian function of width σ	$g(x) = \frac{1}{\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$ $G(\omega) = e^{-\frac{\sigma^2\omega^2}{2}}$	13.4 (a, b)
Rectangular pulse of width $2b$	$g(x) = \Pi_b(x) = \begin{cases} 1 & \text{for } \lvert x\rvert \le b \\ 0 & \text{otherwise} \end{cases}$ $G(\omega) = \frac{2b\sin(b\omega)}{\sqrt{2\pi}\omega}$	13.4 (c, d)

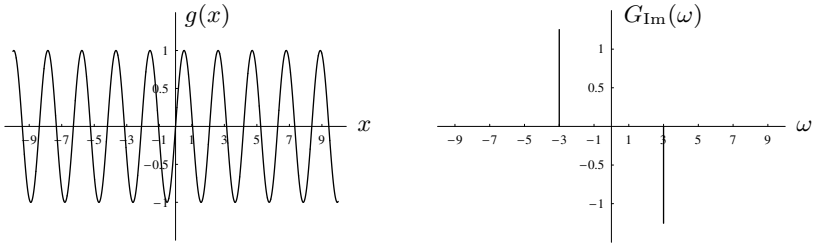
The Fourier spectrum of a *cosine function* $\cos(\omega_0 x)$, for example, consists of two separate thin pulses arranged symmetrically at a distance ω_0 from the origin (Fig. 13.3 (a, c)). Intuitively, this corresponds to our physical understanding of a spectrum (e. g., if we think of a pure

[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.

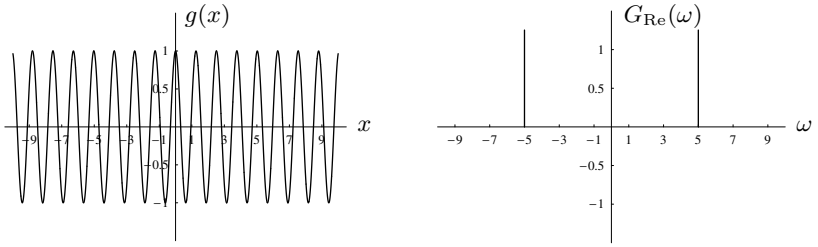
Fig. 13.3
Fourier transform pairs—cosine and sine functions.



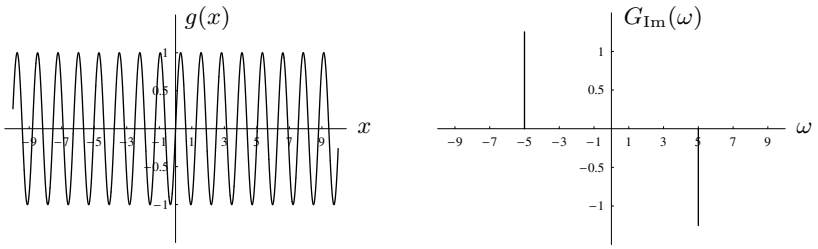
(a) cosine ($\omega_0 = 3$): $g(x) = \cos(3x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}3) + \delta(\omega{-}3)\big)$



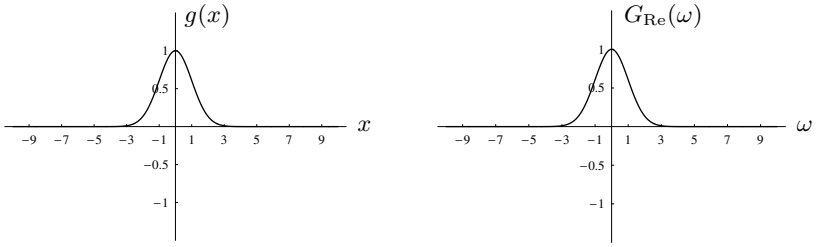
(b) sine ($\omega_0 = 3$): $g(x) = \sin(3x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}3) - \delta(\omega{-}3)\big)$



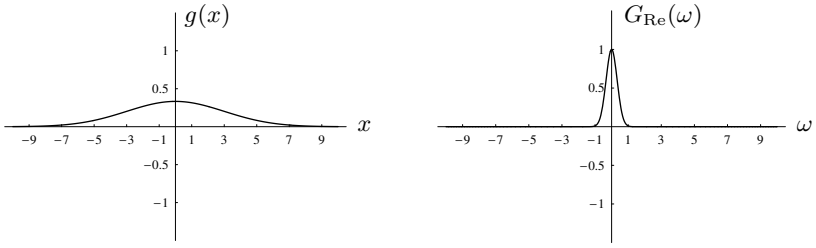
(c) cosine ($\omega_0 = 5$): $g(x) = \cos(5x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}5) + \delta(\omega{-}5)\big)$



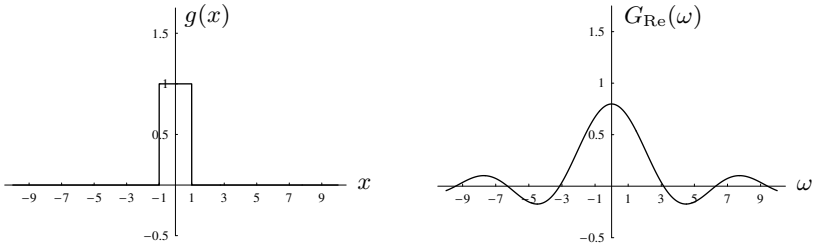
(d) sine ($\omega_0 = 5$): $g(x) = \sin(5x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega{+}5) - \delta(\omega{-}5)\big)$



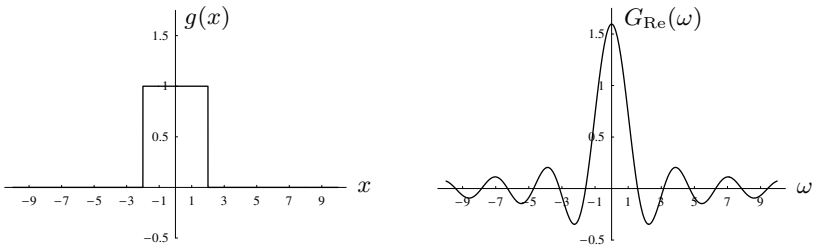
(a) Gauss ($\sigma=1$): $g(x) = e^{-\frac{x^2}{2}}$ ⟜⊸ $G(\omega) = e^{-\frac{\omega^2}{2}}$



(b) Gauss ($\sigma=3$): $g(x) = \frac{1}{3} \cdot e^{-\frac{x^2}{2 \cdot 9}}$ ⟜⊸ $G(\omega) = e^{-\frac{9\omega^2}{2}}$



(c) rect. pulse ($b=1$): $g(x) = \Pi_1(x)$ ⟜⊸ $G(\omega) = \frac{2\sin(\omega)}{\sqrt{2\pi}\omega}$



(d) rect. pulse ($b=2$): $g(x) = \Pi_2(x)$ ⟜⊸ $G(\omega) = \frac{4\sin(2\omega)}{\sqrt{2\pi}\omega}$

Fig. 13.4
Fourier transform pairs—Gaussian functions and rectangular pulses.

Definitions:

$$r_u = \frac{u - M/2}{M/2} = \frac{2u}{M} - 1 \qquad r_v = \frac{v - N/2}{N/2} = \frac{2v}{N} - 1 \qquad r_{u,v} = \sqrt{r_u^2 + r_v^2}$$

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

Table 14.1
2D windowing functions. 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.

$$\boldsymbol{A}_1 = \begin{pmatrix} 3.3\dot{3} & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.3\dot{3} & -0.50 & 1.00 \end{pmatrix} \quad \text{and} \quad \boldsymbol{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}.$$

Concatenating the inverse mapping $\boldsymbol{A}_1^{-1}$ with $\boldsymbol{A}_2$ (by matrix multiplication), we get the complete mapping $\boldsymbol{A} = \boldsymbol{A}_2 \cdot \boldsymbol{A}_1^{-1}$ with

$$\boldsymbol{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \text{and} \quad \boldsymbol{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}.$$

The Java method `makeMapping()` in class `ProjectiveMapping` (p. 420) shows an implementation of this two-step technique.

16.1.5 Bilinear Mapping

Similar to the projective transformation (Eqn. (16.17)), the bilinear mapping function

$$\begin{aligned} T_x: \quad x' &= a_1 x + a_2 y + a_3 xy + a_4, \\ T_y: \quad y' &= b_1 x + b_2 y + b_3 xy + b_4, \end{aligned} \tag{16.35}$$

is specified with four pairs of corresponding points and has eight parameters ($a_1 \ldots a_4$, $b_1 \ldots b_4$). The transformation is nonlinear because of the mixed term xy 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 $(\boldsymbol{x}_1, \boldsymbol{x}'_1) \ldots (\boldsymbol{x}_4, \boldsymbol{x}'_4)$. In the general case, for a bilinear mapping between arbitrary quadrilaterals, the coefficients $a_1 \ldots a_4$, $b_1 \ldots b_4$ (Eqn. (16.35)) are found as the solution of two separate systems of equations, each with four unknowns:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \\ x_4 & y_4 & x_4 \cdot y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \quad \text{or} \quad \boldsymbol{x} = \boldsymbol{M} \cdot \boldsymbol{a}, \tag{16.36}$$

$$\begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \\ x_4 & y_4 & x_4 \cdot y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \quad \text{or} \quad \boldsymbol{y} = \boldsymbol{M} \cdot \boldsymbol{b}. \tag{16.37}$$

These equations can again be solved using standard numerical techniques, as described on page 381. A sample implementation of this computation is shown by the Java method `makeInverseMapping()` inside the class `BilinearMapping` on page 422.

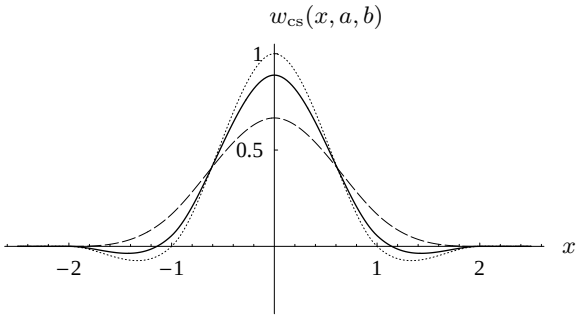


Fig. 16.21
Examples of cardinal spline functions $w_{cs}(x, a, b)$ as specified by Eqn. (16.56): *Catmull-Rom* spline $w_{cs}(x, 0.5, 0)$ (dotted line), *cubic B-spline* $w_{cs}(x, 0, 1)$ (dashed line), and *Mitchell-Netravali* function $w_{cs}(x, \frac{1}{3}, \frac{1}{3})$ (solid line).

spline in computer graphics. In its general form, this function takes not only one but *two* control parameters (a, b) [65],[6]

$$w_{cs}(x, a, b) = \frac{1}{6} \cdot \begin{cases} (-6a - 9b + 12) \cdot |x|^3 \\ \quad + (6a + 12b - 18) \cdot |x|^2 - 2b + 6 & \text{for } 0 \le |x| < 1 \\ (-6a - b) \cdot |x|^3 + (30a + 6b) \cdot |x|^2 \\ \quad + (-48a - 12b) \cdot |x| + 24a + 8b & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.56}$$

Equation (16.56) describes a family of C1-continuous functions; i. e., the functions and their first derivatives are continuous everywhere and thus their trajectories exhibit no discontinuities or corners. For $b = 0$, the function $w_{cs}(x, a, b)$ specifies a one-parameter family of so-called *cardinal splines* equivalent to the cubic interpolation function $w_{cub}(x, a)$ in Eqn. (16.53),

$$w_{cs}(x, a, 0) = w_{cub}(x, a),$$

and for the standard setting $a = 1$ (Eqn. (16.54)) in particular

$$w_{cs}(x, 1, 0) = w_{cub}(x, 1) = w_{cub}(x).$$

Figure 16.21 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 below. The actual computation of the interpolated signal follows exactly the same scheme as used for the cubic interpolation described in Eqn. (16.55).

Catmull-Rom interpolation

With the control parameters set to $a = 0.5$ and $b = 0$, the function in Eqn. (16.56) is a *Catmull-Rom spline* [19], as already mentioned in Sec. 16.3.4:

[6] In [65], the parameters a and b were originally named C and B, respectively, with $B \equiv b$ and $C \equiv a$.

$$w_{\text{crm}}(x) = w_{\text{cs}}(x, 0.5, 0)$$
$$= \frac{1}{2} \cdot \begin{cases} 3 \cdot |x|^3 - 5 \cdot |x|^2 + 2 & \text{for } 0 \le |x| < 1 \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \quad (16.57)$$

Examples of signals interpolated with this kernel are shown in Fig. 16.22 (a–c). The results are similar to ones produced by cubic interpolation (with $a = 1$, see Fig. 16.20) with regard to sharpness, but the Catmull-Rom reconstruction is clearly superior in smooth signal regions (compare, e. g., Fig. 16.20 (d) vs. Fig. 16.22 (a)).

Cubic B-spline approximation

With parameters set to $a = 0$ and $b = 1$, Eqn. (16.56) corresponds to a cubic B-spline function [10] of the form

$$w_{\text{cbs}}(x) = w_{\text{cs}}(x, 0, 1)$$
$$= \frac{1}{6} \cdot \begin{cases} 3 \cdot |x|^3 - 6 \cdot |x|^2 + 4 & \text{for } 0 \le |x| < 1 \\ -|x|^3 + 6 \cdot |x|^2 - 12 \cdot |x| + 8 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \quad (16.58)$$

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. 16.22 (d–f)). Notice also 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.

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 [72] proposed a cubic interpolation kernel as described in Eqn. (16.56) with parameter settings $a = \frac{1}{3}$ and $b = \frac{1}{3}$, and the resulting interpolation function

$$w_{\text{mn}}(x) = w_{\text{cs}}\left(x, \tfrac{1}{3}, \tfrac{1}{3}\right)$$
$$= \frac{1}{18} \cdot \begin{cases} 21 \cdot |x|^3 - 36 \cdot |x|^2 + 16 & \text{for } 0 \le |x| < 1 \\ -7 \cdot |x|^3 + 36 \cdot |x|^2 - 60 \cdot |x| + 32 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2. \end{cases} \quad (16.59)$$

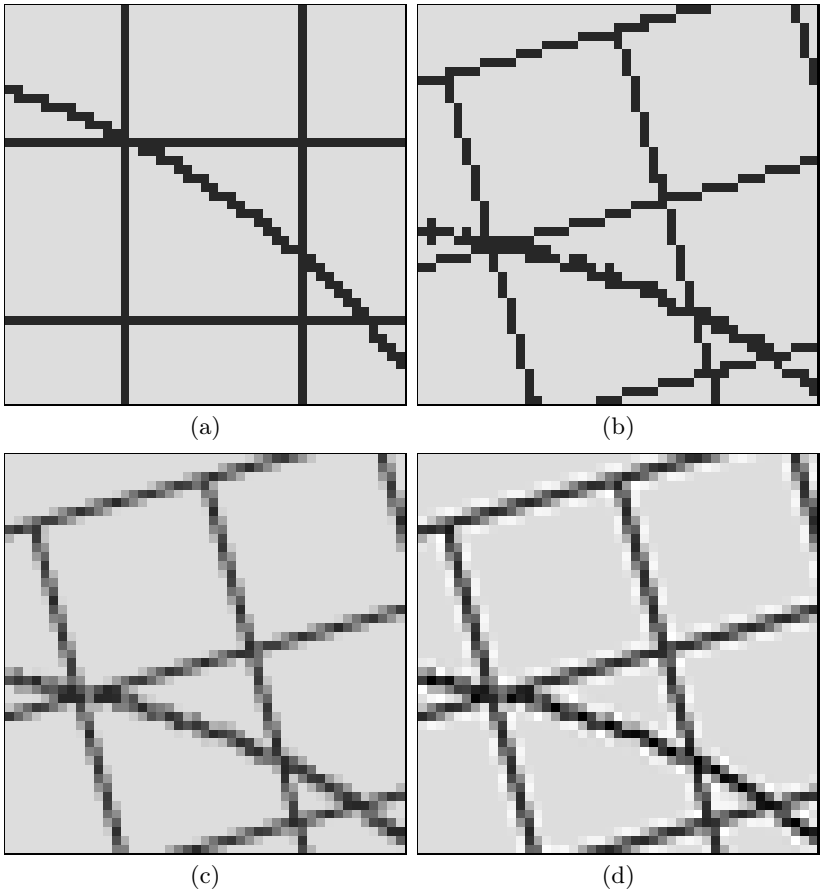


Fig. 16.31
Image interpolation methods compared: part of the original image (a), which is subsequently rotated by 15°, nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

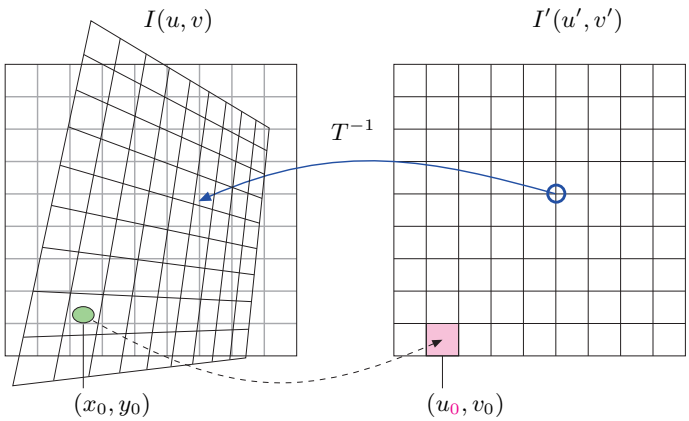


Fig. 16.32
Sampling errors in geometric operations. If the geometric transformation T leads to a local contraction of the image (which corresponds to a local enlargement by T^{-1}), the distance between adjacent sample points in I is increased. This reduces the local sampling frequency and thus the maximum signal frequency allowed in the source image, which eventually leads to aliasing.

C.4.3 ByteProcessor (Class)

`ByteProcessor (Image img)`
Constructor method: creates a new `ByteProcessor` object from an 8-bit image *img* of type `java.awt.Image`.

`ByteProcessor (int width, int height)`
Constructor method: creates a blank `ByteProcessor` object of size *width* × *height*.

`ByteProcessor (int width, int height, byte[] pixels, ColorModel cm)`
Constructor method: creates a new `ByteProcessor` object of the specified size and the color model *cm* (of type `java.awt.image.ColorModel`), with the pixel values taken from the one-dimensional `byte` array *pixels*.

C.4.4 ColorProcessor (Class)

`ColorProcessor (Image img)`
Constructor method: creates a new `ColorProcessor` object from the RGB image *img* of type `java.awt.Image`.

`ColorProcessor (int width, int height)`
Constructor method: creates a blank `ColorProcessor` object of size *width* × *height*.

`ColorProcessor (int width, int height, int[] pixels)`
Constructor method: creates a new `ColorProcessor` object of the specified size with the pixel values taken from the one-dimensional `int` array *pixels*.

C.4.5 FloatProcessor (Class)

`FloatProcessor (float[][] pixels)`
Constructor method: creates a new `FloatProcessor` object from the two-dimensional `float` array *pixels*, which is assumed to store the image data as *pixels*`[u][v]` (i. e., in column-first order).

`FloatProcessor (int[][] pixels)`
Constructor method: creates a new `FloatProcessor` object from the two-dimensional `int` array *pixels*; otherwise the same as above.

`FloatProcessor (int width, int height)`
Constructor method: creates a blank `FloatProcessor` object of size *width* × *height*.

`FloatProcessor (int width, int height, double[] pixels)`
Constructor method: creates a new `FloatProcessor` object of the specified size with the pixel values taken from the one-dimensional `double` array *pixels*. The resulting image uses the default grayscale color model.