22

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

$$\mathsf{GetInterpolatedValue}(I, x, y),$$

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,

$$\tilde{g}(x) \leftarrow g(u_x), \tag{22.1}$$

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,

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_22

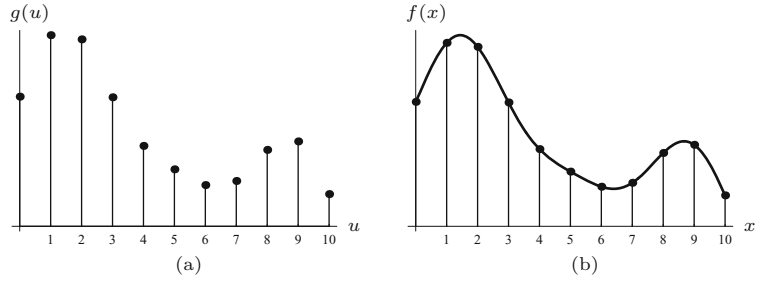


Fig. 22.1
Interpolating a discrete function in 1D. Given the discrete function values $g(u)$ (a), the goal is to estimate the original function $f(x)$ at arbitrary continuous positions $x \in \mathbb{R}$ (b).

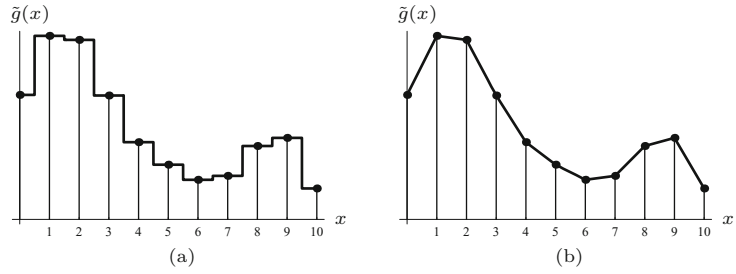


Fig. 22.2
Simple interpolation methods. The *nearest-neighbor interpolation* (a) simply selects the discrete sample $g(u)$ closest to the given continuous coordinate x as the interpolating value $\hat{g}(x)$. Under *linear interpolation* (b), the result is a piecewise linear function connecting adjacent samples $g(u)$ and $g(u+1)$.

$$\begin{aligned}\tilde{g}(x) &= g(u_x) + (x - u_x) \cdot \big(g(u_x + 1) - g(u_x)\big) \\ &= g(u_x) \cdot \big(1 - (x - u_x)\big) + g(u_x + 1) \cdot (x - u_x).\end{aligned} \tag{22.2}$$

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

$$\omega_s = 2 \cdot \pi \cdot f_s = 2 \cdot \pi \cdot \frac{1}{\tau_s} = 2 \cdot \pi \tag{22.3}$$

and thus the maximum signal frequency is $\omega_{\max} = \frac{\omega_s}{2} = \pi$. To isolate the frequency range $-\omega_{\max} \ldots \omega_{\max}$ in the corresponding (periodic)

Fourier spectrum, we multiply the spectrum $G(\omega)$ by a square windowing function $\Pi_\pi(\omega)$ of width $\pm\omega_{\max} = \pm\pi$,

$$\tilde{G}(\omega) \;=\; G(\omega) \cdot \Pi_\pi(\omega) \;=\; G(\omega) \cdot \begin{cases} 1 & \text{for } -\pi \le \omega \le \pi, \\ 0 & \text{otherwise.} \end{cases} \tag{22.4}$$

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

$$\mathrm{Sinc}(x) = \frac{\sin(\pi x)}{\pi x}, \tag{22.5}$$

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.

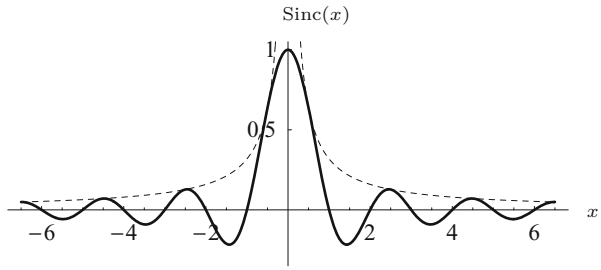


Fig. 22.3
Sinc function in 1D. The function $\mathrm{Sinc}(x)$ has the value 1 at the origin and zero values at all integer positions. The dashed line plots the amplitude $|\frac{1}{\pi x}|$ of the underlying sine function.

So theoretically $\mathrm{Sinc}(x)$ 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 $\mathrm{Sinc}(x)$ are *convolved*. The reconstructed value of the continuous function at position x_0 is thus

$$\tilde{g}(x_0) = [\mathrm{Sinc} * g]\,(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u), \tag{22.6}$$

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

$$\tilde{g}(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u \bmod N)\,. \tag{22.7}$$

[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).

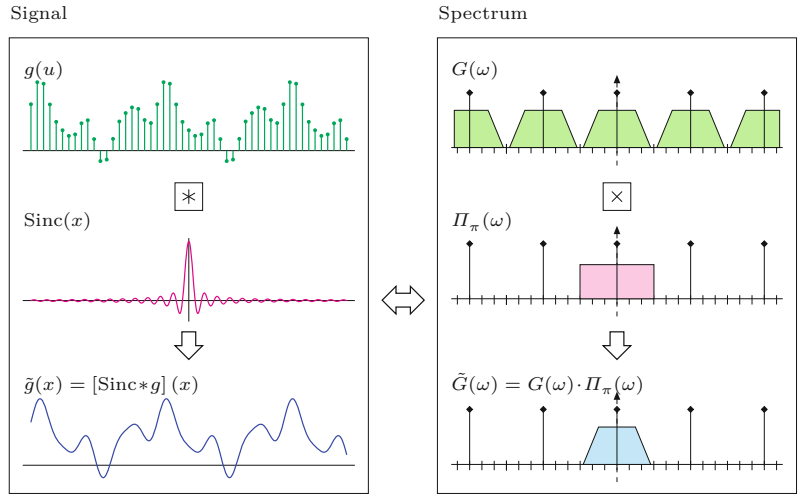


Fig. 22.4 Interpolation of a discrete signal—relation between signal and frequency space. The discrete signal $g(u)$ in signal space (left) corresponds to the periodic Fourier spectrum $G(\omega)$ in frequency space (right). The spectrum $\tilde{G}(\omega)$ of the continuous signal is isolated from $G(\omega)$ 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 Sinc(x).

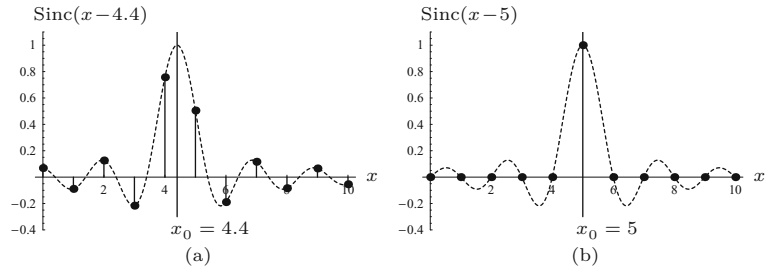


Fig. 22.5 Interpolation by convolving with the Sinc function. The Sinc function is shifted by aligning its origin with the interpolation points $x_0 = 4.4$ (a) and $x_0 = 5$ (b). The values of the shifted Sinc function (dashed curve) at the integral positions are the weights (coefficients) for the corresponding 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

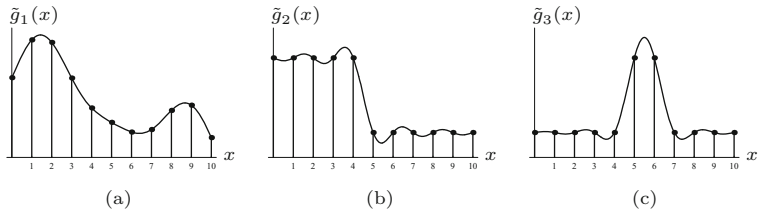


Fig. 22.6
Sinc interpolation applied to various signal types. The reconstructed function in (a) is identical to the continuous, band-limited original. The results for the step function (b) and the pulse function (c) show the strong ringing caused by Sinc (ideal low-pass) interpolation.

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

$$\tilde{g}(x_0) = [w * g](x_0) = \sum_{u=-\infty}^{\infty} w(x_0 - u) \cdot g(u). \tag{22.8}$$

The Sinc interpolation in Eqn. (22.6) is obviously only a special case with $w(x) = \mathrm{Sinc}(x)$. Similarly, the 1D *nearest-neighbor interpolation* (Eqn. (22.1), Fig. 22.2(a)) can be expressed as a linear convolution with the kernel

$$w_{\mathrm{nn}}(x) = \begin{cases} 1 & \text{for } -0.5 \le x < 0.5, \\ 0 & \text{otherwise,} \end{cases} \tag{22.9}$$

and the *linear interpolation* (see Eqn. (22.2), Fig. 22.2(b)) with the kernel

$$w_{\mathrm{lin}}(x) = \begin{cases} 1 - x & \text{for } |x| < 1, \\ 0 & \text{for } |x| \ge 1. \end{cases} \tag{22.10}$$

Both interpolation kernels $w_{\mathrm{nn}}(x)$ and $w_{\mathrm{lin}}(x)$ are shown in Fig. 22.7, and results for various function types are plotted in Fig. 22.8.

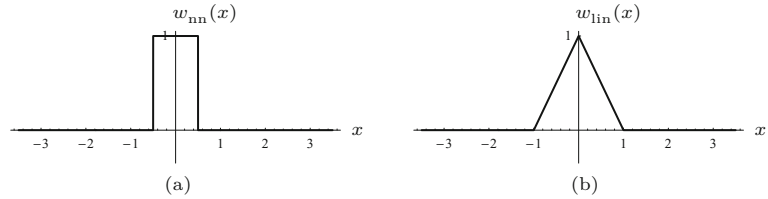


Fig. 22.7 Convolution kernels for the nearest-neighbor interpolation $w_{\text{nn}}(x)$ and the linear interpolation $w_{\text{lin}}(x)$.

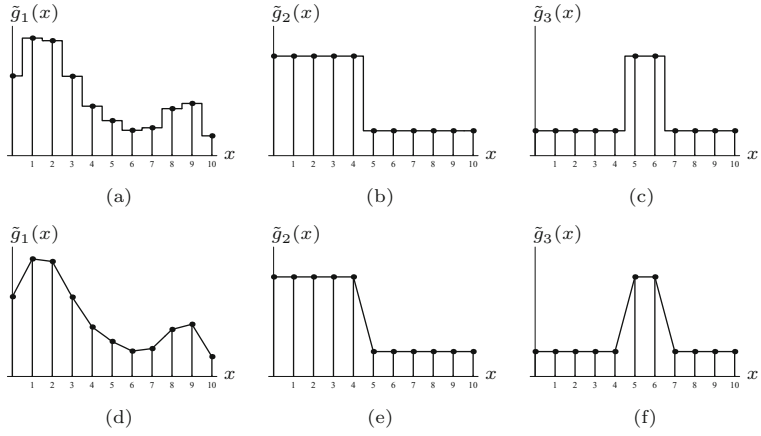


Fig. 22.8 Interpolation examples (1D): nearest-neighbor interpolation (a–c), linear interpolation (d–f).

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

$$w_{\text{cub}}(x,a) = \begin{cases} (-a+2)\cdot|x|^3 + (a-3)\cdot|x|^2 + 1 & \text{for } 0 \le |x| < 1, \\ -a\cdot|x|^3 + 5a\cdot|x|^2 - 8a\cdot|x| + 4a & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases} \tag{22.11}$$

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

$$w_{\text{cub}}(x) = \begin{cases} |x|^3 - 2\cdot|x|^2 + 1 & \text{for } 0 \le |x| < 1, \\ -|x|^3 + 5\cdot|x|^2 - 8\cdot|x| + 4 & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases} \tag{22.12}$$

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

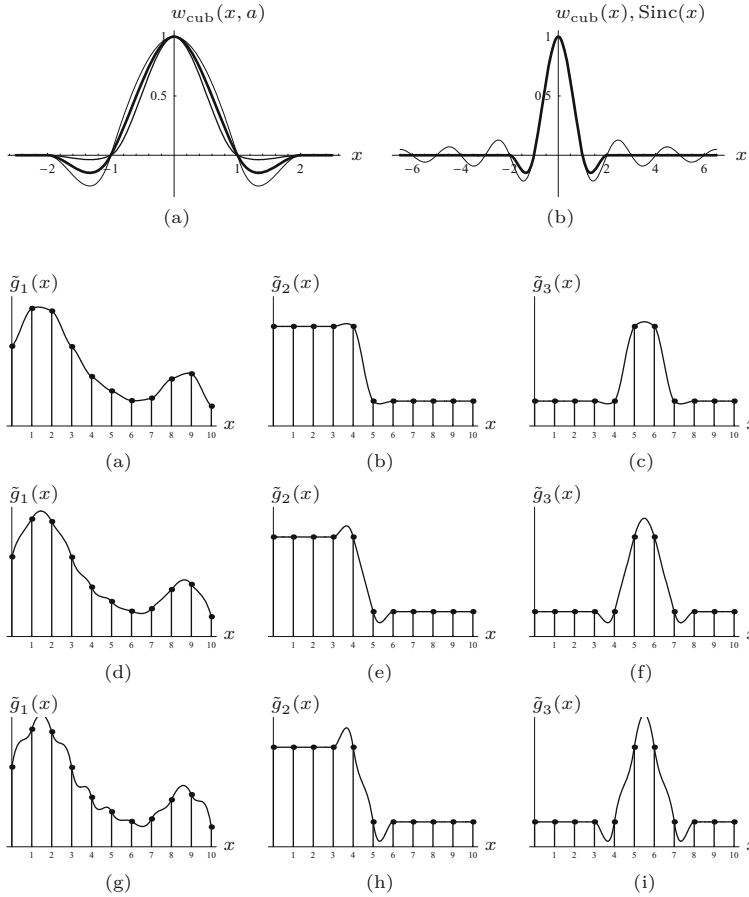


Fig. 22.9
Cubic interpolation kernel. Function $w_{\text{cub}}(x, a)$ with control parameter a set to $a = 0.25$ (dashed curve), $a = 1$ (continuous curve), and $a = 1.75$ (dotted curve) (a). Cubic function $w_{\text{cub}}(x)$ and Sinc function compared (b).

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,

$$g(u_0-1),\ g(u_0),\ g(u_0+1),\ g(u_0+2), \quad \text{with} \quad u_0 = \lfloor x_0 \rfloor.$$

This reduces the 1D cubic interpolation to the expression

$$\tilde{g}(x_0) = \sum_{u=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} w_{\text{cub}}(x_0 - u) \cdot g(u)\ . \tag{22.13}$$

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]

$$w_{\mathrm{cs}}(x, a, b) = \tag{22.14}$$
$$\frac{1}{6} \cdot \begin{cases} (-6a - 9b + 12) \cdot |x|^3 & \\ \quad + (6a + 12b - 18) \cdot |x|^2 - 2b + 6 & \text{for } 0 \le |x| < 1, \\ (-6a - b) \cdot |x|^3 + (30a + 6b) \cdot |x|^2 & \\ \quad + (-48a - 12b) \cdot |x| + 24a + 8b & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases}$$

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 $w_{\mathrm{cs}}(x, a, b)$ specifies a one-parameter family of so-called *cardinal splines* equivalent to the cubic interpolation function $w_{\mathrm{cub}}(x, a)$ in Eqn. (22.11),

$$w_{\mathrm{cs}}(x, a, 0) = w_{\mathrm{cub}}(x, a), \tag{22.15}$$

and for the standard setting $a = 1$ (Eqn. (22.12)) in particular

$$w_{\mathrm{cs}}(x, 1, 0) = w_{\mathrm{cub}}(x, 1) = w_{\mathrm{cub}}(x). \tag{22.16}$$

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).

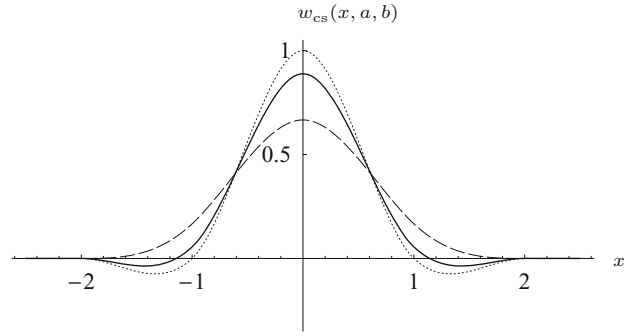


Fig. 22.11 Examples of cubic spline functions as defined in Eqn. (22.14): *Catmull-Rom* spline $w_{\mathrm{cs}}(x, 0.5, 0)$ (dotted line), *cubic B-spline* $w_{\mathrm{cs}}(x, 0, 1)$ (dashed line), and *Mitchell-Netravali* function $w_{\mathrm{cs}}(x, \frac{1}{3}, \frac{1}{3})$ (solid line).

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 $C \equiv a$.

$$w_{\text{crm}}(x) = w_{\text{cs}}(x, 0.5, 0) \tag{22.17}$$
$$= \frac{1}{2} \cdot \begin{cases} 3 \cdot |x|^3 - 5 \cdot |x|^2 + 2 & \text{for } 0 \le |x| < 1, \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases}$$

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

$$w_{\text{cbs}}(x) = w_{\text{cs}}(x, 0, 1) \tag{22.18}$$
$$= \frac{1}{6} \cdot \begin{cases} 3 \cdot |x|^3 - 6 \cdot |x|^2 + 4 & \text{for } 0 \le |x| < 1, \\ -|x|^3 + 6 \cdot |x|^2 - 12 \cdot |x| + 8 & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases}$$

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

$$w_{\text{mn}}(x) = w_{\text{cs}}\left(x, \tfrac{1}{3}, \tfrac{1}{3}\right) \tag{22.19}$$
$$= \frac{1}{18} \cdot \begin{cases} 21 \cdot |x|^3 - 36 \cdot |x|^2 + 16 & \text{for } 0 \le |x| < 1, \\ -7 \cdot |x|^3 + 36 \cdot |x|^2 - 60 \cdot |x| + 32 & \text{for } 1 \le |x| < 2, \\ 0 & \text{for } |x| \ge 2. \end{cases}$$

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.

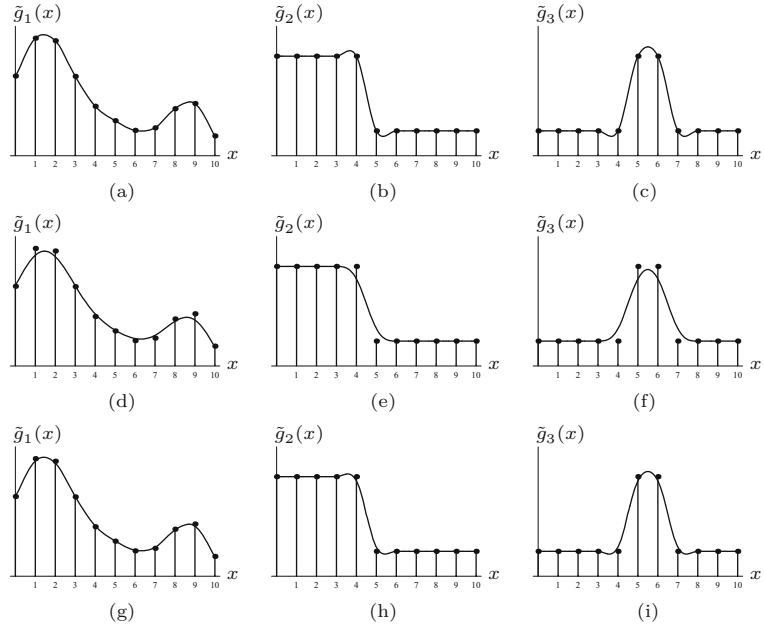


Fig. 22.12
Cardinal spline reconstruction examples: *Catmull-Rom* interpolation (a–c), *cubic B-spline* approximation (d–f), and *Mitchell-Netravali* approximation (g–i).

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 Lanczos[4] 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

$$w(x) = \psi(x) \cdot \mathrm{Sinc}(x)\,. \tag{22.20}$$

The particular window functions for the Lanczos interpolation are defined as

$$\psi_{\mathrm{L}n}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/n)}{\pi x/n} & \text{for } 0 < |x| < n, \\ 0 & \text{for } |x| \geq n, \end{cases} \tag{22.21}$$

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).

$$\psi_{\mathrm{L2}}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/2)}{\pi x/2} & \text{for } 0 < |x| < 2, \\ 0 & \text{for } |x| \geq 2, \end{cases} \tag{22.22}$$

$$\psi_{\mathrm{L3}}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ \frac{\sin(\pi x/3)}{\pi x/3} & \text{for } 0 < |x| < 3, \\ 0 & \text{for } |x| \geq 3. \end{cases} \tag{22.23}$$

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,

$$w_{\mathrm{L2}}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ 2 \cdot \frac{\sin(\pi x/2) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 2, \\ 0 & \text{for } |x| \geq 2, \end{cases} \tag{22.24}$$

and

$$w_{\mathrm{L3}}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ 3 \cdot \frac{\sin(\pi x/3) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 3, \\ 0 & \text{for } |x| \geq 3, \end{cases} \tag{22.25}$$

respectively. In general, for Lanczos interpolation of order n, we get

$$w_{\mathrm{L}n}(x) = \begin{cases} 1 & \text{for } |x| = 0, \\ n \cdot \frac{\sin(\pi x/n) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < n, \\ 0 & \text{for } |x| \geq n. \end{cases} \tag{22.26}$$

Figure 22.13(c, d) shows the resulting interpolation kernels together with the original Sinc function. The function $w_{\mathrm{L2}}(x)$ is quite similar to the Catmull-Rom kernel $w_{\mathrm{crm}}(x)$ (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 $w_{\mathrm{L3}}(x)$ 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) *1*D signals from discrete samples. Images are *2*D signals but, as we

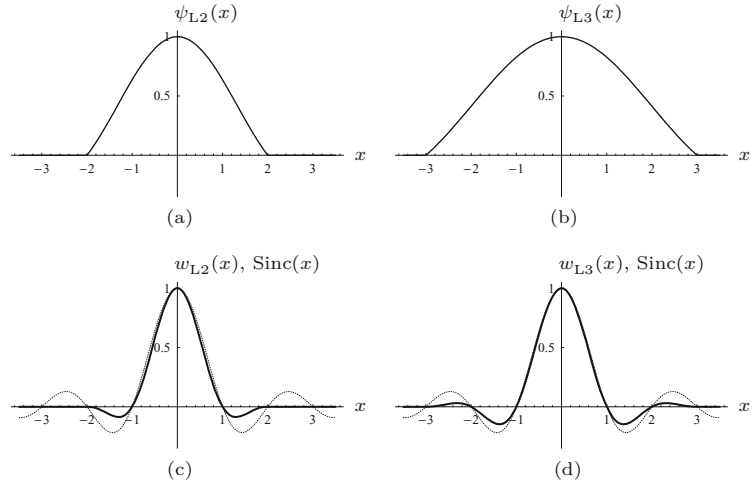


Fig. 22.13 1D Lanczos interpolation kernels. Lanczos window functions ψ_{L2} (a), ψ_{L3} (b), and the corresponding interpolation kernels w_{L2} (c), w_{L3} (d). The original Sinc function (dotted curve) is shown for comparison.

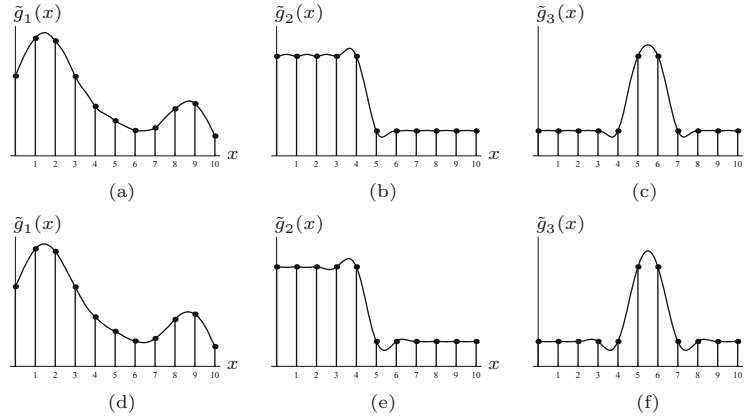


Fig. 22.14 Lanczos interpolation examples: Lanczos-2 (a–c), Lanczos-3 (d–f). Note the ringing in the flat (constant) regions caused by Lanczos-2 interpolation in the left part of (b). The Lanczos-3 interpolator shows less ringing (e) but produces steeper edges at the cost of increased overshoot (e, f).

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

$$\text{Sinc}(x, y) = \text{Sinc}(x) \cdot \text{Sinc}(y) = \frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi y)}{\pi y}, \tag{22.27}$$

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,

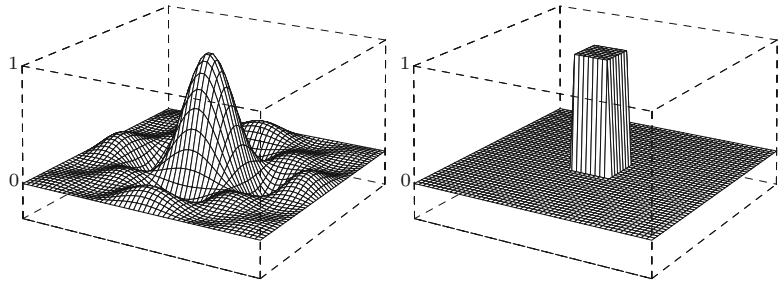


Fig. 22.15
Interpolation kernels in 2D. Sinc kernel $\text{SINC}(x, y)$ (a) and nearest-neighbor kernel $W_{\text{nn}}(x, y)$ (b) for $-3 \le x, y \le 3$.

$$\tilde{I}(x, y) = I(u_x, v_y), \tag{22.28}$$

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

$$W_{\text{nn}}(x, y) = \begin{cases} 1 & \text{for } -0.5 \le x, y < 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{22.29}$$

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.

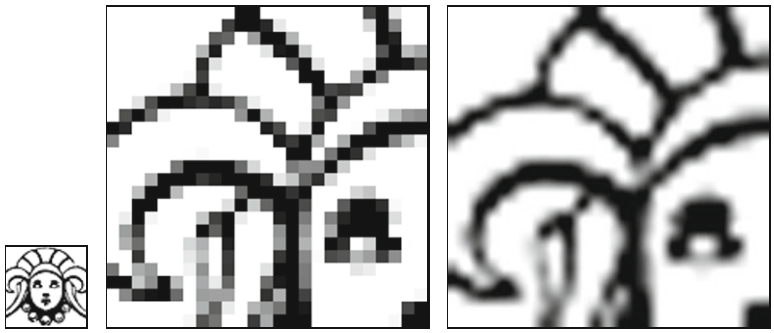


Fig. 22.16
Image enlargement example. Original (a); 8× enlargement using nearest-neighbor interpolation (b) and bicubic interpolation (c).

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 vcalues,

$$\begin{aligned} A &= I(u_x, v_y), & B &= I(u_x + 1, v_y), \\ C &= I(u_x, v_y + 1), & D &= I(u_x + 1, v_y + 1), \end{aligned} \tag{22.30}$$

[5] Not to be confused with the bilinear *mapping* (transformation) described in Chapter 21, Sec. 21.1.5.

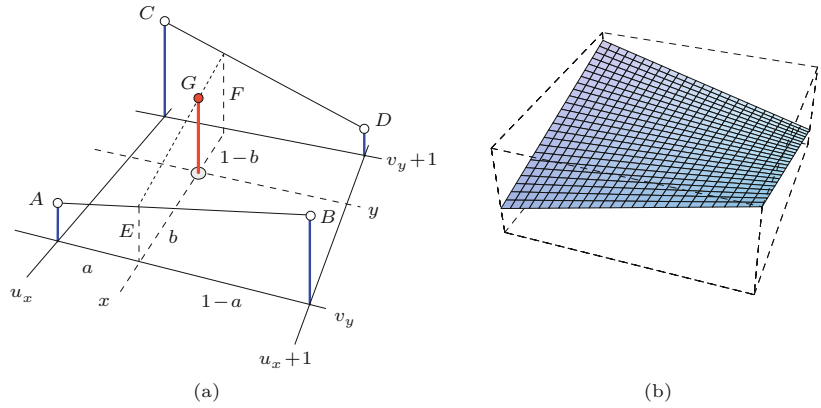


Fig. 22.17 Bilinear interpolation. For a given position (x, y), the interpolated value is computed from the values A, B, C, D of the four closest pixels in two steps (a). First the intermediate values E and F are computed by linear interpolation in the horizontal direction between A, B and C, D, respectively, 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

$$E = A + (x - u_x) \cdot (B - A) = A + a \cdot (B - A), \tag{22.31}$$

$$F = C + (x - u_x) \cdot (D - C) = C + a \cdot (D - C), \tag{22.32}$$

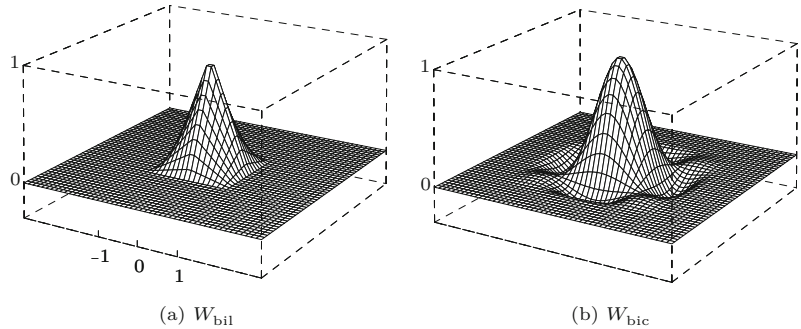
and the final interpolation value G is computed from the vertical distance $b = y_0 - v_y$ as

$$\begin{aligned} \tilde{I}(x, y) = G &= E + (y - v_y) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a-1)(b-1)\, A + a(1-b)\, B + (1-a)\, b\, C + a\, b\, D\,. \end{aligned} \tag{22.33}$$

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,

$$\begin{aligned} W_{\text{bilin}}(x, y) &= w_{\text{lin}}(x) \cdot w_{\text{lin}}(y) \\ &= \begin{cases} 1 - x - y + x \cdot y & \text{for } 0 \le |x|, |y| < 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{22.34}$$

In this function (plotted in Fig. 22.18), we can recognize the bilinear term that gives this method its name.



(a) W_{bil} (b) W_{bic}

Fig. 22.18 2D interpolation kernels. bilinear kernel $W_{\text{bil}}(x, y)$ (a) and bicubic kernel $W_{\text{bic}}(x, y)$ (b) for $-3 \le x, y \le 3$.

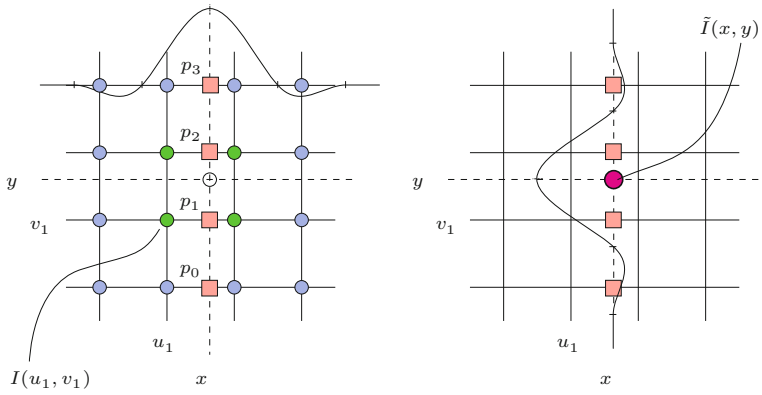


Fig. 22.19
Bicubic interpolation in two steps. The discrete image I (pixel positons correspond to raster lines) is to be interpolated at some continuous position (x, y). In step 1 (left), a 1D interpolation is performed in the horizontal direction with $w_{\text{cub}}(x)$ over four pixels $I(u_i, v_j)$ in four lines. One intermediate result p_j (marked □) 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 $p_0, \ldots, p_3$. In total, $16 + 4 = 20$ interpolation steps are required.

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)),

$$W_{\text{bic}}(x, y) = w_{\text{cub}}(x) \cdot w_{\text{cub}}(y). \tag{22.35}$$

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

$$\tilde{I}(x, y) = \sum_{\substack{v=\\ \lfloor y \rfloor - 1}}^{\lfloor y \rfloor + 2} \left[\sum_{\substack{u=\\ \lfloor x \rfloor - 1}}^{\lfloor x \rfloor + 2} I(u, v) \cdot W_{\text{bic}}(x - u, y - v) \right] \tag{22.36}$$

$$= \sum_{j=0}^{3} \left[w_{\text{cub}}(y - v_j) \cdot \underbrace{\sum_{i=0}^{3} I(u_i, v_j) \cdot w_{\text{cub}}(x - u_i)}_{p_j} \right], \tag{22.37}$$

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.

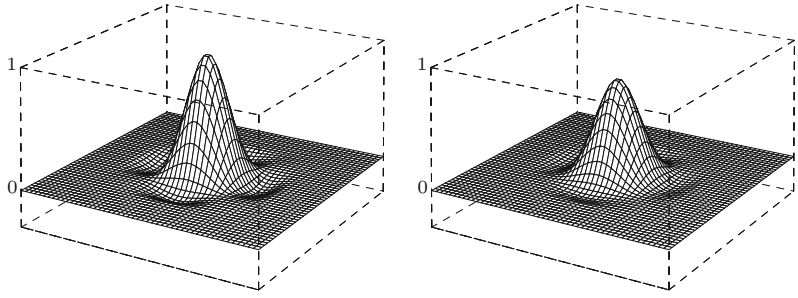
This method, which is summarized in Alg. 22.1, can be used to implement any x/y-separable 2D interpolation kernel of size 4×4, such as the 2D *Catmull-Rom* interpolation (Eqn. (22.17)) with

$$W_{\text{crm}}(x, y) = w_{\text{crm}}(x) \cdot w_{\text{crm}}(y) \tag{22.38}$$

or the *Mitchell-Netravali* interpolation (Eqn. (22.19)) with

$$W_{\text{mn}}(x, y) = w_{\text{mn}}(x) \cdot w_{\text{mn}}(y). \tag{22.39}$$

The corresponding 2D kernels are shown in Fig. 22.20. For interpolation with separable kernels of larger size see the general procedure in Alg. 22.2.



(a) W_{crm} (b) W_{mn}

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 $-3 \le x, y \le 3$.

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.

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).

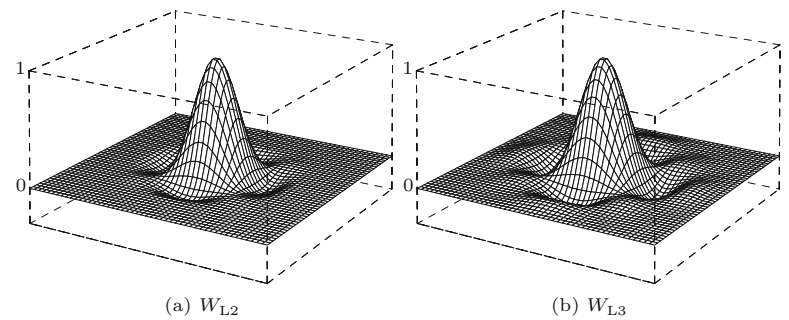
$q \leftarrow 0$
for $j \leftarrow 0, \ldots, 3$ **do** ▷ iterate over 4 lines
  $v \leftarrow \lfloor y \rfloor - 1 + j$
  $p \leftarrow 0$
  for $i \leftarrow 0, \ldots, 3$ **do** ▷ iterate over 4 columns
    $u \leftarrow \lfloor x \rfloor - 1 + i$
    $p \leftarrow p + I(u, v) \cdot w_{\text{cub}}(x - u, a)$ ▷ see Eq. 22.11
  $q \leftarrow q + p \cdot w_{\text{cub}}(y - v, a)$
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,

$$W_{\text{L}n}(x, y) = w_{\text{L}n}(x) \cdot w_{\text{L}n}(y) \,. \tag{22.40}$$

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 $-2 \le x, y \le 2$, 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}).



(a) W_{L2} (b) W_{L3}

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 $-3 \le x, y \le 3$.

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 $n = 2$).

```
1:  SeparableInterpolation(I, x, y, w, n)
        Input: I, original image; x, y ∈ ℝ, continuous position; w, a 1D
        interpolation kernel of extent ±n (n ≥ 1).
        Returns the interpolated image value at position (x, y) using the
        composite interpolation kernel W(x, y) = w(x) · w(y).
2:      q ← 0
3:      for j ← 0, ..., 2n−1 do                    ▷ iterate over 2n lines
4:          v ← ⌊y⌋ − n + 1 + j                    ▷ = v_j
5:          p ← 0
6:          for i ← 0, ..., 2n−1 do                ▷ iterate over 2n columns
7:              u ← ⌊x⌋ − n + 1 + i                ▷ = u_i
8:              p ← p + I(u, v) · w(x − u)
9:          q ← q + p · w(y − v)
10:     return q
```

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

$$\tilde{I}(x,y) = \sum_{\substack{v=\\ \lfloor y\rfloor-2}}^{\lfloor y\rfloor+3} \left[\sum_{\substack{u=\\ \lfloor x\rfloor-2}}^{\lfloor x\rfloor+3} I(u,v) \cdot W_{\text{L3}}(x-u, y-v) \right] \tag{22.41}$$

$$= \sum_{j=0}^{5} \left[w_{\text{L3}}(y-v_j) \cdot \sum_{i=0}^{5} I(u_i,v_j) \cdot w_{\text{L3}}(x-u_i) \right], \tag{22.42}$$

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 Ln of arbitrary order $n \geq 1$ is

$$\tilde{I}(x,y) = \sum_{\substack{v=\\ \lfloor y\rfloor-n+1}}^{\lfloor y\rfloor+n} \left[\sum_{\substack{u=\\ \lfloor x\rfloor-n+1}}^{\lfloor x\rfloor+n} \big[I(u,v) \cdot W_{\text{Ln}}(x-u, y-v) \big] \right] \tag{22.43}$$

$$= \sum_{j=0}^{2n-1} \left[w_{\text{Ln}}(y-v_j) \cdot \sum_{i=0}^{2n-1} \big[I(u_i,v_j) \cdot w_{\text{Ln}}(x-u_i) \big] \right], \tag{22.44}$$

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 $W_{\mathrm{L}2}$ 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 $W_{\mathrm{L}3}$ (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 $I'(u', v')$.

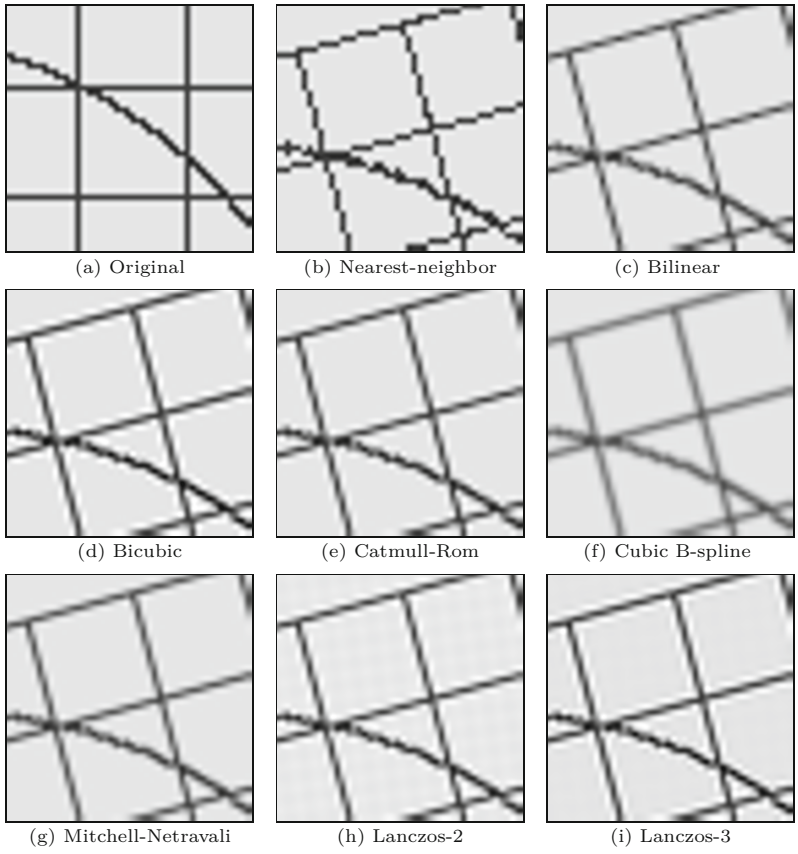


Fig. 22.22
Image interpolation methods compared (line art).

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).

(a) Original (b) Nearest-neighbor (c) Bilinear

(d) Bicubic (e) Catmull-Rom (f) Cubic B-spline

(g) Mitchell-Netravali (h) Lanczos-2 (i) Lanczos-3

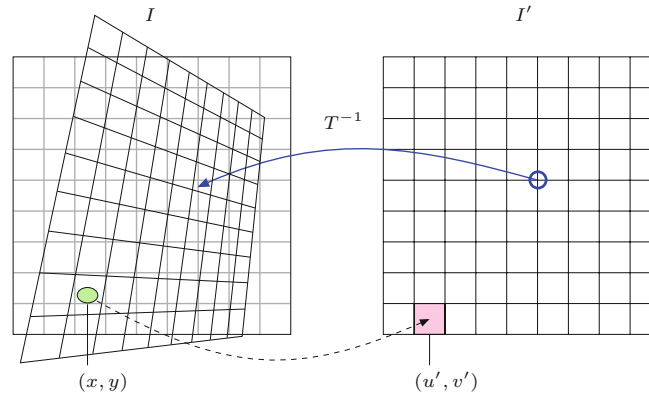


Fig. 22.24
Sampling errors in geometric operations. If the geometric transformation T leads to a local contraction of the image (which corresponds to a local enlargement by T^{-1}), the distance between adjacent sample points in I is increased. This reduces the local sampling frequency and thus the maximum 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

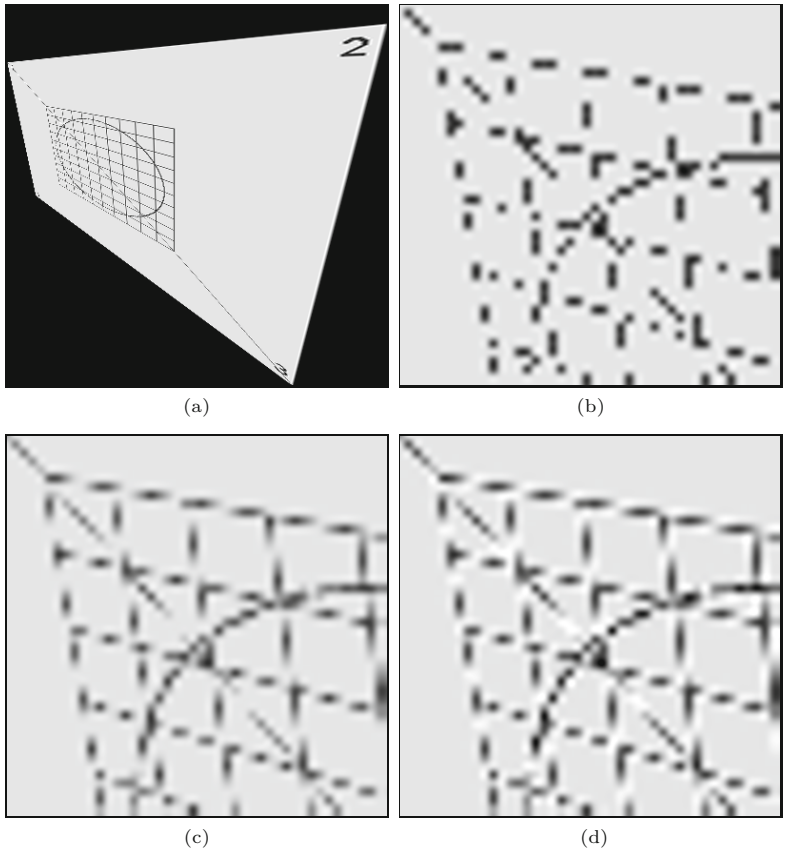


Fig. 22.25
Aliasing caused by local image contraction. Aliasing is caused by a violation of the sampling criterion and is largely unaffected by the interpolation method used: complete transformed image (a), detail using nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

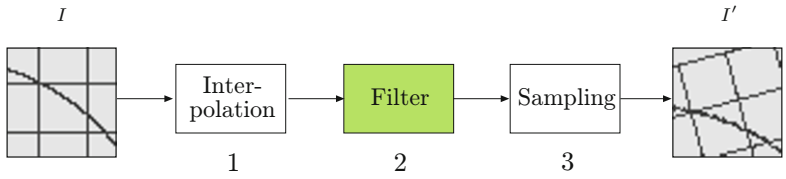


Fig. 22.26
Low-pass filtering to avoid aliasing in geometric operations. After interpolation (step 1), the reconstructed image function is subjected to low-pass filtering (step 2) before being resampled (step 3).

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`	`NearestNeighborInterpolator()`
`Bilinear`	`BilinearInterpolator()`
`Bicubic`	`BicubicInterpolator(1.00)`
`BicubicSmooth`	`BicubicInterpolator(0.25)`
`BicubicSharp`	`BicubicInterpolator(1.75)`
`CatmullRom`	`SplineInterpolator(0.5, 0.0)`
`CubicBSpline`	`SplineInterpolator(0.0, 1.0)`
`MitchellNetravali`	`SplineInterpolator(1.0/3, 1.0/3)`
`Lanzcos2`	`LanczosInterpolator(2)`
`Lanzcos3`	`LanczosInterpolator(3)`
`Lanzcos4`	`LanczosInterpolator(4)`

[6] Package `imagingbook.lib.interpolation`.

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)).

```
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   public BicubicInterpolator(double a) {
15     this.a = a;
16   }
17
18   public float getInterpolatedValue(
19           ImageAccessor.Scalar ia, double x, double y) {
20     final int u0 = (int) Math.floor(x);
21     final int v0 = (int) Math.floor(y);
22     double q = 0;
23     for (int j = 0; j <= 3; j++) {
24       int v = v0 - 1 + j;
25       double p = 0;
26       for (int i = 0; i <= 3; i++) {
27         int u = u0 - 1 + i;
28         float pixval = ia.getVal(u, v);
29         p = p + pixval * w_cub(x - u, a);
30       }
31       q = q + p * w_cub(y - v, a);
32     }
33     return (float) q;
34   }
35
36   private final double w_cub(double x, double a) {
37     if (x < 0)
38       x = -x;
39     double z = 0;
40     if (x < 1)
41       z = (-a + 2) * x * x * x + (a - 3) * x * x + 1;
42     else if (x < 2)
43       z = -a * x * x * x + 5 * a * x * x
44           - 8 * a * x + 4 * a;
45     return z;
46   }
47 }
```

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.

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 }
```

22.8 Exercises

Exercise 22.1. The 1D interpolation function by Mitchell and Natravali $w_{\mathrm{mn}}(x)$ is defined as a general spline function $w_{\mathrm{cs}}(x, a, b)$ (Eqn. (22.19)). Show that this function can be expressed as the weighted sum of a Catmull-Rom function $w_{\mathrm{crm}}(x)$ (Eqn. (22.17)) and a cubic B-spline $w_{\mathrm{cbs}}(x)$ (Eqn. (22.18)) in the form

$$\begin{aligned} w_{\mathrm{mn}}(x) &= w_{\mathrm{cs}}\big(x, \tfrac{1}{3}, \tfrac{1}{3}\big) \\ &= \tfrac{1}{3} \cdot \big[2 \cdot w_{\mathrm{cs}}(x, 0.5, 0) + w_{\mathrm{cs}}(x, 0, 1)\big] \\ &= \tfrac{1}{3} \cdot \big[2 \cdot w_{\mathrm{crm}}(x) + w_{\mathrm{cbs}}(x)\big]. \end{aligned} \tag{22.45}$$

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

$$w_{\mathrm{L4}} = \begin{cases} 4 \cdot \frac{\sin(\pi \frac{x}{4}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 \le |x| < 4, \\ 0 & \text{for } |x| \ge 4. \end{cases} \tag{22.46}$$

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).