Digital Image Processing
An algorithmic introduction using Java
Second Edition

ERRATA
Fig. 5.7
Border geometry. The filter can be applied only at locations where the kernel $H$ of size $(2K + 1) \times (2L + 1)$ is fully contained in the image (inner rectangle).

of key importance in practice: smoothing filters and difference filters (Fig. 5.8).

**Smoothing filters**

Every filter we have discussed so far causes some kind of smoothing. In fact, any linear filter with positive-only coefficients is a smoothing filter in a sense, because such a filter computes merely a weighted average of the image pixels within a certain image region.

**Box filter**

This simplest of all smoothing filters, whose 3D shape resembles a box (Fig. 5.8(a)), is a well-known friend already. Unfortunately, the box filter is far from an optimal smoothing filter due to its wild behavior in frequency space, which is caused by the sharp cutoff around its sides. Described in frequency terms, smoothing corresponds to low-pass filtering, that is, effectively attenuating all signal components above a given cutoff frequency (see also Chs. 18–19). The box filter, however, produces strong “ringing” in frequency space and is therefore not considered a high-quality smoothing filter. It may also appear rather ad hoc to assign the same weight to all image pixels in the filter region. Instead, one would probably expect to have stronger emphasis given to pixels near the center of the filter than to the more distant ones. Furthermore, smoothing filters should possibly operate “isotropically” (i.e., uniformly in each direction), which is certainly not the case for the rectangular box filter.

**Gaussian filter**

The filter matrix (Fig. 5.8(b)) of this smoothing filter corresponds to a 2D Gaussian function,

$$H^{G,\sigma}(x, y) = e^{-\frac{x^2 + y^2}{2\sigma^2}} = e^{-\frac{r^2}{2\sigma^2}}, \quad (5.14)$$

where $\sigma$ denotes the width (standard deviation) of the bell-shaped function and $r$ is the distance (radius) from the center. The pixel at the center receives the maximum weight (1.0, which is scaled to the integer value 9 in the matrix shown in Fig. 5.8(b)), and the remaining coefficients drop off smoothly with increasing distance from the center.
center. The Gaussian filter is isotropic if the discrete filter matrix is large enough for a sufficient approximation (at least $5 \times 5$). As a low-pass filter, the Gaussian is “well-behaved” in frequency space and thus clearly superior to the box filter. The 2D Gaussian filter is separable into a pair of 1D filters (see Sec. 5.3.3), which facilitates its efficient implementation.\footnote{See also Sec. E in the Appendix.}

**Difference filters**

If some of the filter coefficients are negative, the filter calculation can be interpreted as the difference of two sums: the weighted sum of all pixels with associated positive coefficients minus the weighted sum of pixels with negative coefficients in the filter region $R_H$, that is,

$$I'(u, v) = \sum_{(i,j) \in R^+} I(u+i, v+j) \cdot |H(i,j)| - \sum_{(i,j) \in R^-} I(u+i, v+j) \cdot |H(i,j)|,$$

where $R^+_H$ and $R^-_H$ denote the partitions of the filter with positive coefficients $H(i,j) > 0$ and negative coefficients $H(i,j) < 0$, respectively. For example, the $5 \times 5$ Laplace filter in Fig. 5.8(c) computes the difference between the center pixel (with weight 16) and the weighted sum of 12 surrounding pixels (with weights $-1$ or $-2$). The remaining 12 pixels have associated zero coefficients and are thus ignored in the computation.

While local intensity variations are smoothed by averaging, we can expect the exact contrary to happen when differences are taken: local intensity changes are enhanced. Important applications of difference filters thus include edge detection (Sec. 6.2) and image sharpening (Sec. 6.6).

### 5.3 Formal Properties of Linear Filters

In the previous sections, we have approached the concept of filters in a rather casual manner to quickly get a grasp of how filters are defined and used. While such a level of treatment may be sufficient for most practical purposes, the power of linear filters may not really
the pencil exhibits the least mass inertia (Fig. 10.17). As long as a region exhibits an orientation at all (i.e., \( \mu_{11}(R) \neq 0 \)), the direction \( \theta_R \) of the major axis can be found directly from the central moments \( \mu_{pq} \) as

\[
\tan(2 \theta_R) = \frac{2 \cdot \mu_{11}(R)}{\mu_{20}(R) - \mu_{02}(R)}
\]

(10.27)

and thus the corresponding angle is

\[
\theta_R = \frac{1}{2} \cdot \tan^{-1}\left(\frac{2 \cdot \mu_{11}(R)}{\mu_{20}(R) - \mu_{02}(R)}\right)
\]

(10.28)

\[
= \frac{1}{2} \cdot \text{ArcTan}\left(\mu_{20}(R) - \mu_{02}(R), 2 \cdot \mu_{11}(R)\right)
\]

(10.29)

The resulting angle \( \theta_R \) is in the range \([-\frac{\pi}{2}, \frac{\pi}{2}]\). Orientation measurements based on region moments are very accurate in general.

**Calculating orientation vectors**

When visualizing region properties, a frequent task is to plot the region’s orientation as a line or arrow, usually anchored at the center of gravity \( \bar{x} = (\bar{x}, \bar{y})^T \); for example, by a parametric line of the form

\[
x = \bar{x} + \lambda \cdot x_d = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_R) \\ \sin(\theta_R) \end{pmatrix}
\]

(10.30)

with the normalized orientation vector \( x_d \) and the length variable \( \lambda > 0 \). To find the unit orientation vector \( x_d = (\cos \theta, \sin \theta)^T \), we could first compute the inverse tangent to get \( 2 \theta \) (Eqn. (10.28)) and then compute the cosine and sine of \( \theta \). However, the vector \( x_d \) can also be obtained without using trigonometric functions as follows. Rewriting Eqn. (10.27) as

\[
\tan(2 \theta_R) = \frac{2 \cdot \mu_{11}(R)}{\mu_{20}(R) - \mu_{02}(R)} = \frac{\sin(2 \theta_R)}{\cos(2 \theta_R)}
\]

(10.31)

we get (by Pythagora’s theorem)

\[^9\] See Sec. A.1 in the Appendix for the computation of angles with the \text{ArcTan}() (inverse tangent) function and Sec. F.1.6 for the corresponding Java method \text{Math.atan2()}.
In practice, the logarithm of these quantities (that is, \( \log(\phi_k) \)) is used since the raw values may have a very large range. These features are also known as moment invariants since they are invariant under translation, rotation, and scaling. While defined here for binary images, they are also applicable to parts of grayscale images; examples can be found in [88, p. 517].

**Flusser’s invariant moments**

It was shown in [72, 73] that Hu’s moments, as listed in Eqn. (10.47), are partially redundant and incomplete. Based on so-called complex moments \( c_{pq} \in \mathbb{C} \), Flusser designed an improved set of 11 rotation and scale-invariant features \( \psi_1, \ldots, \psi_{11} \) (see Eqn. (10.51)) for characterizing 2D shapes. For grayscale images (with \( I(u, v) \in \mathbb{R} \)), the complex moments of order \( p, q \) are defined as

\[
c_{p,q}(R) = \sum_{(u,v) \in R} I(u,v) \cdot (u + i \cdot v)^p \cdot (u - i \cdot v)^q, \quad (10.48)
\]

with centered positions \( x = u - \bar{x} \) and \( y = v - \bar{y} \), and \((\bar{x}, \bar{y})\) being the centroid of \( R \) (it denotes the imaginary unit). In the case of binary images (with \( I(u, v) \in \{0, 1\} \)) Eqn. (10.48) simplifies to

\[
c_{p,q}(R) = \sum_{(u,v) \in R} (u + i \cdot v)^p \cdot (u - i \cdot v)^q. \quad (10.49)
\]

Analogous to Eqn. (10.26), the complex moments can be scale-normalized to

\[
\hat{c}_{p,q}(R) = \frac{1}{A^{(p+q+2)/2}} \cdot c_{p,q}(R), \quad (10.50)
\]

with \( A \) being the area of \( R \) [74, p. 29]. Finally, the derived rotation and scale invariant region moments of 2nd to 4th order are\(^{15}\)

\[
\begin{align*}
\psi_1 &= \text{Re}(\hat{c}_{1,1}), & \psi_2 &= \text{Re}(\bar{c}_{2,1} \cdot \hat{c}_{1,2}), & \psi_3 &= \text{Re}(\bar{c}_{2,0} \cdot \hat{c}_{1,2}), \\
\psi_4 &= \text{Im}(\bar{c}_{2,0} \cdot \hat{c}_{1,2}), & \psi_5 &= \text{Re}(\bar{c}_{3,0} \cdot \hat{c}_{1,2}), & \psi_6 &= \text{Im}(\bar{c}_{3,0} \cdot \hat{c}_{1,2}), \\
\psi_7 &= \text{Re}(\bar{c}_{2,2}), & \psi_8 &= \text{Re}(\bar{c}_{3,1} \cdot \hat{c}_{1,2}), & \psi_9 &= \text{Im}(\bar{c}_{3,1} \cdot \hat{c}_{1,2}), \\
\psi_{10} &= \text{Re}(\bar{c}_{4,0} \cdot \hat{c}_{1,2}), & \psi_{11} &= \text{Im}(\bar{c}_{4,0} \cdot \hat{c}_{1,2}).
\end{align*} \quad (10.51)
\]

Table 10.1 lists the normalized Flusser moments for five binary shapes taken from the Kimia dataset [134].

**Shape matching with region moments**

One obvious use of invariant region moments is shape matching and classification. Given two binary shapes \( A \) and \( B \), with associated moment ("feature") vectors

\[
\mathbf{f}_A = (\psi_1(A), \ldots, \psi_{11}(A)) \quad \text{and} \quad \mathbf{f}_B = (\psi_1(B), \ldots, \psi_{11}(B)),
\]

respectively, one approach could be to simply measure the difference between shapes by the Euclidean distance of these vectors in the form\(^{15}\)

\[
\text{In Eqn. (10.51), the use of } \text{Re}(\cdot)\text{ for the quantities } \psi_1, \psi_2, \psi_7 \text{ (which are real-valued per se) is redundant.}
\]

\[242\]
12.1 RGB Color Images

Fig. 12.2
A color image and its corresponding RGB channels. The fruits depicted are mainly yellow and red and therefore have high values in the $R$ and $G$ channels. In these regions, the $B$ content is correspondingly lower (represented here by darker gray values) except for the bright highlights on the apple, where the color changes gradually to white. The tabletop in the foreground is purple and therefore displays correspondingly higher values in its $B$ channel.

individual color components. In the next sections we will examine the difference between true color images, which utilize colors uniformly selected from the entire color space, and so-called palleted or indexed images, in which only a select set of distinct colors are used. Deciding which type of image to use depends on the requirements of the application.

Duplicate text removed.

True color images

A pixel in a true color image can represent any color in its color space, as long as it falls within the (discrete) range of its individual color components. True color images are appropriate when the image contains many colors with subtle differences, as occurs in digital photography and photo-realistic computer graphics. Next we look at two methods of ordering the color components in true color images: component ordering and packed ordering.
Colorimetric Color Spaces in Chapter 12, Sec. 12.2.1, where we had simply ignored the issue of possible nonlinearities. As one may have guessed, however, the variables $R$, $G$, $B$, and $Y$ in Eqn. (12.10) on p. 305,

$$ Y = 0.2125 \cdot R + 0.7154 \cdot G + 0.072 \cdot B $$

implicitly refer to linear color and gray values, respectively, and not the raw sRGB values! Based on Eqn. (14.37), the correct grayscale conversion from raw (nonlinear) sRGB components $R'$, $G'$, $B'$ is

$$ Y' = f_1(0.2125 \cdot f_2(R') + 0.7154 \cdot f_2(G') + 0.0721 \cdot f_2(B')), $$

with $f_1()$ and $f_2()$ as defined in Eqns. (14.33) and (14.35), respectively. The result ($Y'$) is again a nonlinear, sRGB-compatible gray value; that is, the sRGB color tuple ($Y'$, $Y'$, $Y'$) should have the same perceived luminance as the original color ($R'$, $G'$, $B'$).

Note that setting the components of an sRGB color pixel to three arbitrary but identical values $Y'$,

$$(R', G', B') \leftarrow (Y', Y', Y')$$

always creates a gray (colorless) pixel, despite the nonlinearities of the sRGB space. This is due to the fact that the gamma correction (Eqns. (14.33) and (14.35)) applies evenly to all three color components and thus any three identical values map to a (linearized) color on the straight gray line between the black point $S$ and the white point $W$ in XYZ space (cf. Fig. 14.1(b)).

For many applications, however, the following approximation to the exact grayscale conversion in Eqn. (14.38) is sufficient. It works without converting the sRGB values (i.e., directly on the nonlinear $R'$, $G'$, $B'$ components) by computing a linear combination

$$ Y' \approx w'_R \cdot R' + w'_G \cdot G' + w'_B \cdot B', $$

with a slightly different set of weights; for example, $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$, as proposed in [188]. The resulting quantity from Eqn. (14.39) is sometimes called luma (compared to luminance in Eqn. (14.37)).

14.5 Adobe RGB

A distinct weakness of sRGB is its relatively small gamut, which is limited to the range of colors reproducible by ordinary color monitors. This causes problems, for example, in printing, where larger gamuts are needed, particularly in the green regions. The “Adobe RGB (1998)” [1] color space, developed by Adobe as their own standard, is based on the same general concept as sRGB but exhibits a significantly larger gamut (Fig. 14.3), which extends its use particularly to print applications. Figure 14.7 shows the noted difference between the sRGB and Adobe RGB gamuts in 3D CIEXYZ color space.

The neutral point of Adobe RGB corresponds to the D65 standard (with $x = 0.3127$, $y = 0.3290$), and the gamma value is 2.199
17.2 Bilateral Filter

\[
I'(u, v) \leftarrow \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I(u + m, v + n) \cdot H_d(m, n) \tag{17.15}
\]

\[
= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot H_d(i - u, j - v), \tag{17.16}
\]

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

17.2.2 Range Filter

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

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

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

17.2.3 Bilateral Filter—General Idea

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

\[
I'(u, v) = \frac{1}{W_{u,v}} \cdot \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot H_d(i - u, j - v) \cdot H_r(I(i, j) - I(u, v)), \tag{17.18}
\]

\(^5\) In Eqn. (17.16), functions \( I \) and \( H_d \) are assumed to be zero outside their domains of definition.
17.3 Anisotropic Diffusion Filters

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

\[
I^{(n)}(u) \left\{ \begin{array}{ll}
I(u) & \text{for } n = 0, \\
I^{(n-1)}(u) + \alpha \cdot [\nabla^2 I^{(n-1)}(u)] & \text{for } n > 0,
\end{array} \right.
\tag{17.45}
\]

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

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

\[
\nabla^2 I \approx I \ast H_L, \quad \text{with } H_L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix},
\tag{17.46}
\]

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

\[
I^{(n)} \equiv I \ast H^{G, \sigma_n},
\tag{17.47}
\]

with the normalized Gaussian kernel

\[
H^{G, \sigma_n}(x, y) = \frac{1}{2\pi\sigma^2_n} e^{-\frac{x^2+y^2}{2\sigma^2_n}},
\tag{17.48}
\]

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

\[11\] See also Chapter 6, Sec. 6.6.1 and Sec. C.3.1 in the Appendix.
Fig. 21.2
Affine mapping. An affine 2D transformation is uniquely specified by three pairs of corresponding points, for example, \((x_1, y_1'), (x_2, y_2'), (x_3, y_3'), (x_1, y_1), (x_2, y_2), (x_3, y_3)\).

six transformation parameters \(a_{00}, \ldots, a_{12}\) are derived by solving the system of linear equations

\[
\begin{align*}
 x'_0 &= a_{00} \cdot x_0 + a_{01} \cdot y_0 + a_{02}, \\
 y'_0 &= a_{10} \cdot x_0 + a_{11} \cdot y_0 + a_{12}, \\
 x'_1 &= a_{00} \cdot x_1 + a_{01} \cdot y_1 + a_{02}, \\
 y'_1 &= a_{10} \cdot x_1 + a_{11} \cdot y_1 + a_{12}, \\
 x'_2 &= a_{00} \cdot x_2 + a_{01} \cdot y_2 + a_{02}, \\
 y'_2 &= a_{10} \cdot x_2 + a_{11} \cdot y_2 + a_{12},
\end{align*}
\] (21.25)

provided that the points (vectors) \(x_0, x_1, x_2\) are linearly independent (i.e., that they do not lie on a common straight line). Since Eqn. (21.25) consists of two independent sets of linear \(3 \times 3\) equations for \(x'_i\) and \(y'_i\), the solution can be written in closed form as

\[
\begin{align*}
 a_{00} &= \frac{1}{d} \left[ y_0 (x'_1 - x'_2) + y_1 (x'_2 - x'_0) + y_2 (x'_0 - x'_1) \right], \\
 a_{01} &= \frac{1}{d} \left[ x_0 (x'_0 - x'_2) + x_1 (x'_1 - x'_0) + x_2 (x'_0 - x'_1) \right], \\
 a_{10} &= \frac{1}{d} \left[ y_0 (y'_1 - y'_2) + y_1 (y'_2 - y'_0) + y_2 (y'_0 - y'_1) \right], \\
 a_{11} &= \frac{1}{d} \left[ x_0 (y'_0 - y'_2) + x_1 (y'_1 - y'_2) + x_2 (y'_0 - y'_1) \right], \\
 a_{02} &= \frac{1}{d} \left[ x_0 (y_2 y'_1 - y_1 y'_2) + x_1 (y_0 y'_2 - y_2 y'_0) + x_2 (y_0 y'_0 - y_0 y'_1) \right], \\
 a_{12} &= \frac{1}{d} \left[ x_0 (y_0 y'_1 - y_1 y'_0) + x_1 (y_0 y'_2 - y_2 y'_0) + x_2 (y_0 y'_0 - y_0 y'_1) \right],
\end{align*}
\] (21.26)

with \(d = x_0 (y_2 - y_1) + x_1 (y_0 - y_2) + x_2 (y_1 - y_0)\).

**Inverse affine mapping**

The inverse of the affine transformation, which is often required in practice (see Sec. 21.2.2), can be calculated by simply applying the inverse of the transformation matrix \(A_{\text{affine}}\) (Eqn. (21.20)) in homogeneous coordinate space, that is,

\[
\begin{align*}
 \mathbf{z} &= A_{\text{affine}}^{-1} \cdot \mathbf{z}' \\
 \mathbf{x} &= \text{hom}^{-1} \left[ A_{\text{affine}}^{-1} \cdot \text{hom}(\mathbf{z}') \right]
\end{align*}
\] (21.27)
for $a, b \in \mathbb{R}$. This type of operator or library method was not available in the standard Java API until recently.\(^2\) The following Java method implements the mod operation according to the definition in Eqn. (F.1):\(^3\)

```java
int Mod(int a, int b) {
    if (b == 0)
        return a;
    if (a * b >= 0 || a % b == 0) { ← error fixed!
        return a - b * (a / b);
    } else
        return a - b * (a / b - 1);
}
```

Note that the remainder operator $\%$, defined as

$$a \% b \equiv a - b \cdot \text{truncate}(a/b), \quad \text{for } b \neq 0, \quad \text{(F.2)}$$

is often used in this context, but yields the same results only for positive operands $a \geq 0$ and $b > 0$. For example,

$$
\begin{align*}
13 \mod 4 &= 1 & 13 \mod -4 &= -3 & \text{vs.} & 13 \mod 4 &= 1 & 13 \mod -4 &= -1 \\
-13 \mod 4 &= 3 & -13 \mod 4 &= -1 \\
-13 \mod -4 &= -1 & -13 \mod -4 &= -1
\end{align*}
$$

F.1.3 Unsigned Byte Data

Most grayscale and indexed images in Java and ImageJ are composed of pixels of type `byte`, and the same holds for the individual components of most color images. A single byte consists of eight bits and can thus represent $2^8 = 256$ different bit patterns or values, usually mapped to the numeric range $0, \ldots, 255$. Unfortunately, Java (unlike C and C++) does not provide a suitable “unsigned” 8-bit data type. The primitive Java type `byte` is “signed”, using one of its eight bits for the $\pm$ sign, and is intended to hold values in the range $-128, \ldots, +127$.

Java’s `byte` data can still be used to represent the values $0$ to $255$, but conversions must take place to perform proper arithmetic computations. For example, after execution of the statements

```java
int a = 200;
byte b = (byte) p;
```

the variables $a$ (32-bit `int`) and $b$ (8-bit `byte`) contain the binary patterns

$$
\begin{align*}
a &= 00000000000000000000000000011001000 \\
b &= 11001000
\end{align*}
$$

Interpreted as a (signed) `byte` value, with the leftmost bit\(^4\) as the sign bit, the variable $b$ has the decimal value $-56$. Thus after the statement

\[^2\] Starting with Java version 1.8 the mod operation (as defined in Eqn. (F.1)) is implemented by the standard method `Math.floorMod(a, b)`.
\[^3\] The definition in Eqn. (F.1) is not restricted to integer operands.
\[^4\] Java uses the standard “2’s-complement” representation, where a sign bit $= 1$ stands for a negative value.