Chapter 5

Simple spatial operations

In the previous chapter we discussed grey value remapping operations of the type $x \longrightarrow g(x)$, where x is the grey value at a single pixel in the input image. Later we extended this to x being a vector containing the values of multiple images at a single pixel location. In *this* chapter, we will discuss operations that map grey values to new values considering a *neighborhood* of pixels, *i.e.*, we look at remapping of the type $x \longrightarrow g(N)$, where N represents the pixel values in some neighborhood around the current pixel.

Because we use a neighborhood of pixels, a spatial operation is sometimes referred to as *pixel group processing*. By using a certain neighborhood of pixels, we are now able to use the *spatial* characteristics around some pixel (which is not possible using one-pixel grey value remappings). Another synonym is therefore *spatial filtering*. The operation (or the convolution kernel used) is sometimes called the *filter*.

5.1 Discrete convolution

In chapter 3 we introduced the concept of (spatial) convolution:

$$(g * f)(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(a, b) f(x - a, y - b)\, da\, db, \tag{5.1}$$

where f is an image, and g some convolution kernel. A discrete version of convolution is defined by

$$(g * f)(x, y) = \sum_{(a,b)\in A} g(a, b) f(x - a, y - b), \tag{5.2}$$

where $A = \{(a,b)|g(a,b) \neq 0\}$, *i.e.,* the set of all pairs (a,b) with a non-zero value of $g(a,b)$. Alternatively, we define

$$(g \star f)(x,y) = \sum_{(a,b)\in A} g(a,b) f(x+a, y+b). \tag{5.3}$$

We use this last definition when we need to apply discrete convolution to an image because it is more intuitive than the formal definition of equation 5.2 in this case.

Intermezzo

If we take f and g to be digital images on a unit grid, the one-dimensional convolution equation

$$(g * f)(x) = \int_{-\infty}^{\infty} g(a) f(x-a)\, da.$$

can be written as

$$(g * f)(x) = \sum_{a\in A} g(a) f(x-a),$$

where A is a set containing all a's with a non-zero value $g(a)$, *i.e.,* $A = \{a|g(a) \neq 0\}$. This equation is a discrete version of convolution. The formula for discrete convolution for two-dimensional images analogously becomes:

$$(g * f)(x,y) = \sum_{(a,b)\in A} g(a,b) f(x-a, y-b), \tag{5.4}$$

where $A = \{(a,b)|g(a,b) \neq 0\}$. Note that $(g * f)(x,y)$ is not defined for values of x and y for which values of $f(x-a, y-b)$ do not exist. Equation 5.2 has the drawback that the minus signs work counter-intuitive in practice (we will show this in the example below). Therefore we usually define discrete convolution by

$$(g \star f)(x,y) = \sum_{(a,b)\in A} g(a,b) f(x+a, y+b).$$

When there can be no confusion, we term both the "$*$" and the "$\star$" operations convolutions. The $\star$ operator is sometimes called the correlation operator. Note that the relation $f * g = g * f$ is true, but that this relation does *not* hold for the $\star$ operator.

Having a formal definition, now what does a convolution *do?* From equation 5.3 we can see that the convolution result $(g \star f)(x, y)$ is a weighted sum of image values f around (x, y). Exactly how much each image value contributes to the convolution result is determined by the values in the convolution kernel g. We will work out an example:

Example

Suppose we have a convolution kernel g with

$$g(a,b) = \begin{cases} 1 & \text{if } (a,b) \in \{(-1,-1),(0,0),(1,0),(-1,1)\} \\ 0 & \text{otherwise} \end{cases}$$

i.e., $g =$

	-1	0	1
-1	1	0	0
0	0	1	1
1	1	0	0

.

Expanding equation 5.3 for this choice of g gives us:

$$(g \star f)(x,y) = f(x-1,y-1) + f(x,y) + f(x+1,y) + f(x-1,y+1).$$

So we see that the convolution result is exactly the sum of those image values f that occur in the neighborhood of (x, y) where the kernel g has value 1.

We see now that the effect of a convolution can easily be predicted if we represent the kernel in image form. It will now also be clear why we choose equation 5.3 for the working definition of discrete convolution, and not the formal equation 5.2. In the latter case, the intuitive relation between the convolution result and the pictorial representation of g is far less obvious.

Figure 5.1 shows the convolution process using a 3×3 kernel graphically.

Because discrete convolution allows us to make use of the spatial characteristics around a pixel, the number of possible applications is very large. In fact, virtually *all* image processing tasks that require spatial characteristics are implemented using convolution. In the next sections we will discuss a number of applications that require only simple kernels, generally not larger than 3×3.

Unless specified otherwise, the kernels presented are centered around $(0, 0)$.

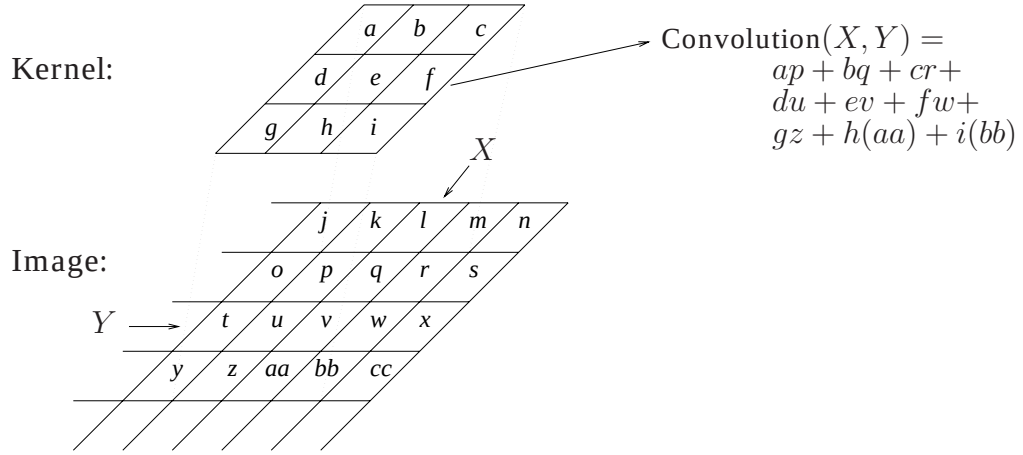


Figure 5.1 Graphical representation of discrete convolution using a 3×3 kernel.

5.1.1 Smoothing and low pass filtering

In chapter 3 we already saw that convolving with a rectangular kernel had a smoothing effect. The discrete version of such a kernel is, *e.g.*,

$$\begin{array}{|c|c|c|}\hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array} = \frac{1}{9} \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array},$$

for a 3×3 kernel. The second form is a short hand form for convenience. The effect of convolving an image with this kernel is that each pixel value is replaced by the average of itself and its eight direct neighbors. Convolving an image with this averaging kernel will have a smoothing effect on the image. It is used for noise reduction and low pass filtering. An example can be seen in figure 5.2. The general formula for an $N \times N$, with N odd, kernel that averages a pixel and all of its direct neighbors is

$$g(x,y) = \begin{cases} \frac{1}{N^2} & \text{if } x,y \in \{-\frac{N-1}{2}, \ldots, \frac{N-1}{2}\} \\ 0 & \text{otherwise.} \end{cases} \tag{5.5}$$

If N is even, we cannot place $(0,0)$ at the center of the kernel, so we cannot make the kernel symmetrical. Since symmetry is often a desired property, even-sized kernels are used far less than odd-sized kernels.

Many different kernels that compute local (*i.e.*, in a neighborhood) averages can be constructed, *e.g.*,

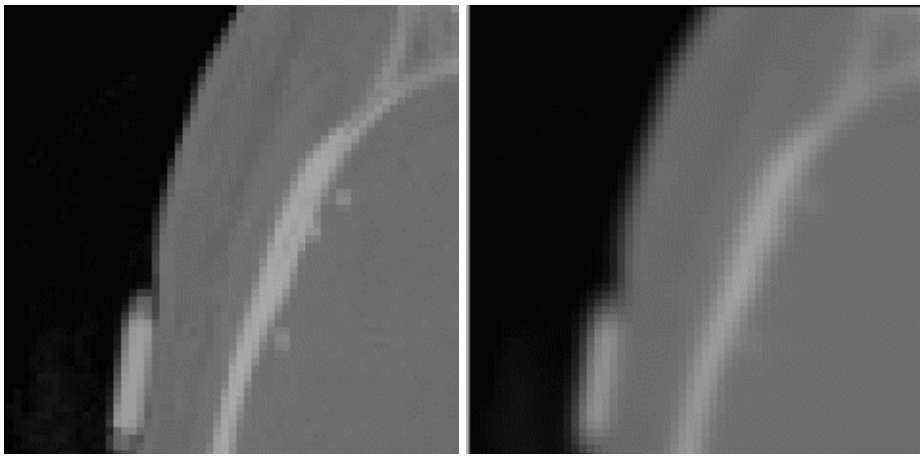


Figure 5.2 Example of convolving an image with a 3×3 averaging kernel.

$$\begin{array}{|c|c|c|}\hline \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \hline \frac{1}{8} & 0 & \frac{1}{8} \\ \hline \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline 0 & \frac{1}{5} & 0 \\ \hline \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \hline 0 & \frac{1}{5} & 0 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline 0 & 0 & 0 \\ \hline \end{array} .$$

With this last kernel it will be clear that its behavior is not isotropic, *i.e.*, it has a directional preference. In this case only pixel values in the x direction are averaged. But not even the kernels of equation 5.5 are isotropic; to achieve isotropy the kernel would have to be circular instead of square. We can only *approximate* a circular shape using square pixels, and the smaller the kernel, the worse the approximation will be.

Averaging kernels are also called *low pass* filters, since they remove high image frequencies (*i.e.*, the details), and let low frequencies "pass" through the filter. Image areas with constant pixel values (*i.e.*, only the zero frequency occurs) are not changed at all.

5.1.2 Sharpening and high pass filtering

An example of a detail enhancing kernel is

$$\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 9 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array} .$$

How does this filter work? Notice that all the factors in the kernel add up to one, so convolution with this filter will not change the pixel value in a part of the image that has constant grey values. But if the pixel has a higher or lower grey value than its neighbors, this contrast will be enlarged by the filter, see this example:

Example

Example of applying the above detail enhancing kernel to an image:

5	5	5	5	5	5	5
5	5	5	5	5	5	5
5	5	5	5	0	0	0
5	5	5	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

$\longrightarrow$

	5	5	10	15	20	
	5	10	20	-20	-15	
	20	25	-15	-5	0	
	-15	-10	-5	0	0	
	0	0	0	0	0	

Notice how the "constant" parts of the image (upper left and bottom right corner) are left unchanged, while the contrast in the parts where the 0's and 5's meet is greatly enhanced.

Figure 5.3 shows an example of this detail enhancing kernel applied to a real image.

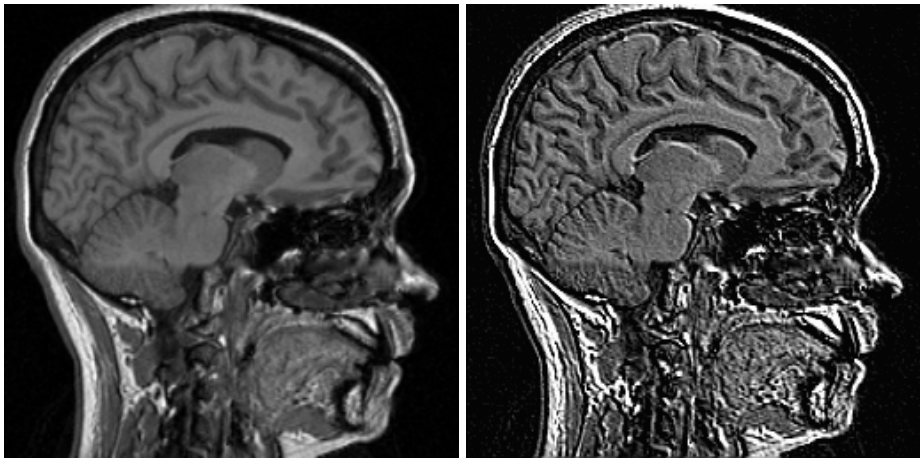


Figure 5.3 Example of applying a 3×3 detail enhancing kernel to a real image. See text for details.

It is common to call this detail enhancing kernel a high pass filter, although this is not strictly true. For one thing, constant areas are left unchanged by this filter, so this filter lets even the lowest frequency (the zero frequency) pass. More correctly, this filter is *high frequency enhancing*, while leaving the low frequencies relatively untouched.

We can construct a high pass filter by using the averaging low pass kernel from the previous section: if we *subtract* the low pass filtered image from the original, the result will contain only the high frequencies of the original image. We can use a single kernel to carry out this operation:

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}.$$

This kernel is very similar to our original detail enhancing filter, but the result is very different, as can be seen in figure 5.4. This filter acts truly as a high *pass* filter, whereas

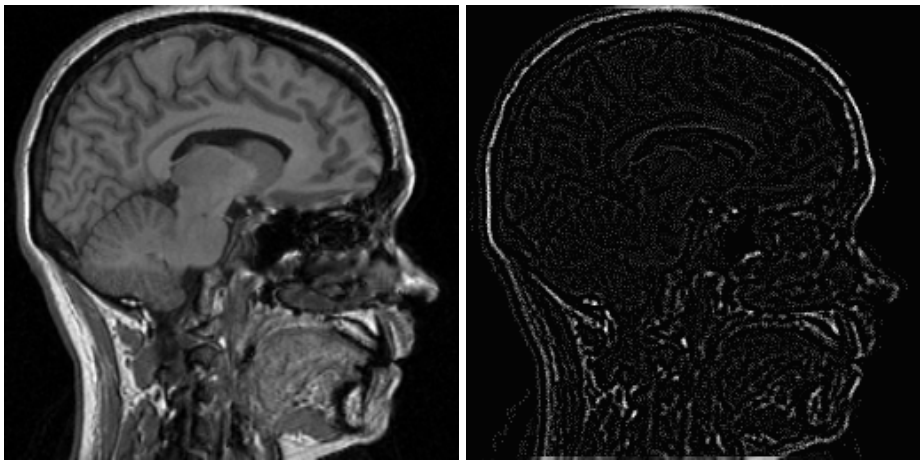


Figure 5.4 Example of high pass filtering an image. See text for details.

the original filter acted as a high frequency *enhancing* filter. This difference is explained by the different value of the center pixel in the kernel: in the case of the enhancing kernel there is a 9, so the kernel entries sum up to one. This means that constant areas remain unchanged. But the second kernel has an 8, which means that the entries sum up to 0, so constant areas are given pixel value 0.

Intermezzo

Enhancing high frequency details is something commonly done in printing processes, and is usually achieved by (1) adding an image proportional to a high pass filtered image to the original image, or (2) subtracting an image proportional to a low pass filtered image from the original. So

1. $f_{e_1}(x,y) = f(x,y) + \lambda_1 f_h(x,y)$
2. $f_{e_2}(x,y) = f(x,y) - \lambda_2 f_l(x,y)$,

where f is the original image, λ_1 and λ_2 are positive constants, f_h is a high pass filtered image, f_l is a low pass filtered image, and f_{e_1} and f_{e_2} are the enhanced images. This technique is called *unsharp masking*. Figure 5.5 shows an example using technique (1) with $\lambda_1 = \frac{1}{4}$ and using the 3×3 high pass kernel defined in the text above:

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}.$$

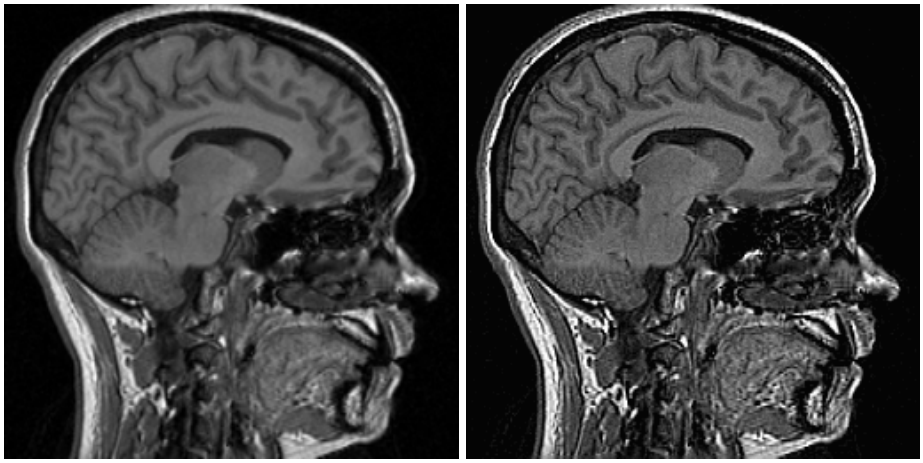


Figure 5.5 Example of unsharp masking. See text for details.

5.1.3 Derivatives

An important application of discrete convolution is the approximation of image *derivatives*. Derivatives play an important role in the mathematics of graphs: practically all of the "interesting" points of a graph (like extrema, inflection points, *etc.*) can be found by solving equations containing derivatives. This also holds true for images: practically all of the interesting points can be found using image derivatives.

A problem with computing derivatives of images is that images are not nice analytical functions for which we have a set of rules on how to compute derivatives. Digital images are nothing but a collection of discrete values on a discrete grid, so how do we compute derivatives? The best approach from the viewpoint of mathematical correctness uses *scale space* techniques and will be dealt with in a later chapter. In this chapter we use approximation techniques taken from numerical mathematics that can be implemented using discrete convolution. Before we do this, we introduce some elements from differential geometry that are frequently used in image processing.

5.1.3.1 Some differential geometry*

In this section we give some definitions and results from calculus and differential geometry. In all cases, f is a multiply differentiable and continuous function with $f : \mathbb{R}^n \rightarrow \mathbb{R}$

(*i.e.,* an n dimensional continuous image), with variables $x_1, x_2, \ldots, x_n$. a Is an interior point of f. For partial derivatives, we use the shorthand $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, $f_{xy} = \frac{\partial^2 f}{\partial x \partial y}$, *etc.*

The gradient and the Hamiltonian

The *nabla* or *gradient* operator ∇ is defined by

$$\nabla = (\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n}).$$

The *gradient* $\nabla f(a) = (\frac{\partial f}{\partial x_1}(a), \ldots, \frac{\partial f}{\partial x_n}(a))$ is a vector that points in the direction of steepest ascent of f starting from a. The length of the gradient $\|\nabla f(a)\|$ (*i.e.,* its norm; its magnitude) is a measure for the rate of ascent.

For $n = 2$, the *Hamiltonian* is the vector that is always perpendicular to the gradient: if $f = f(x, y)$, then the gradient $\nabla f = (f_x, f_y)$, and the Hamiltonian vector $= (f_y, -f_x)$.

With images, the integral curves[1] of the gradient and Hamiltonian are respectively called the image *flowlines* and *isophotes*. The image isophotes can also be defined in a much more intuitive way: they are the curves of constant grey value $f(x, y) = c$, comparable to iso-height lines on a map. See figure 5.6 for an example.

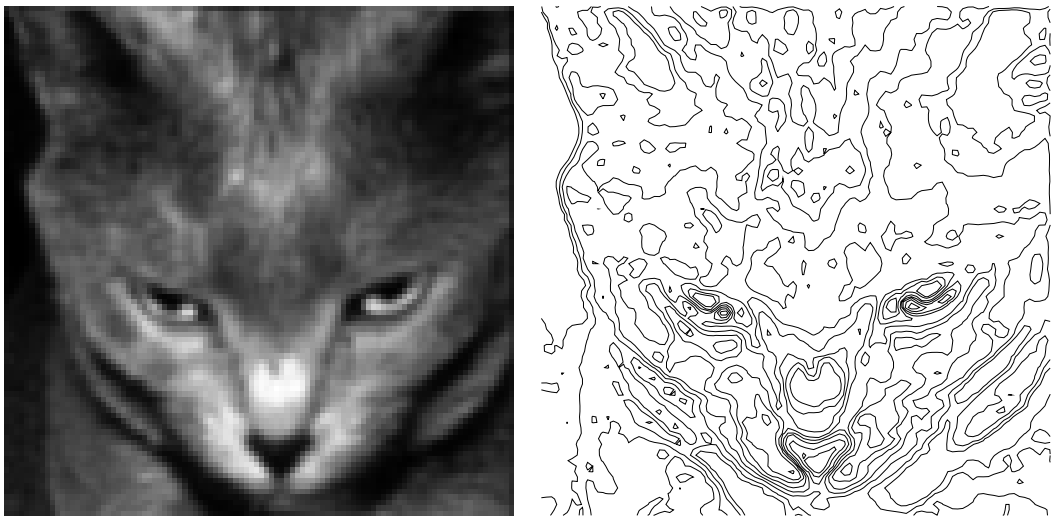


Figure 5.6 Example of an image and some of its isophotes; *i.e.,* some curves of equal image intensity. Note how the curves get closer together in areas of high image contrast.

[1]The integral curve of, *e.g.,* the gradient vector field is a curve whose tangent equals the local gradient vector everywhere.

Stationary points

A point a is called *stationary* if $\nabla f(a) = 0$, *i.e.,* if all of the partial derivatives $\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}$ are zero. An (interior) maximum, minimum, or saddle point is always a stationary point (but the reverse is not necessarily true). The second derivative in a can be used to establish the type of point. If $n = 1$, *i.e.,* $f = f(x)$, a stationary point a is a maximum if $f''(a) < 0$ or a minimum if $f''(a) > 0$. If $n > 1$, we can use the Hessian H —which is the determinant of the matrix containing all second order partial derivatives— to establish the point type. For instance, if $n = 2$, *i.e.,* $f = f(x, y)$:

$$H(a) = \begin{vmatrix} f_{xx}(a) & f_{xy}(a) \\ f_{xy}(a) & f_{yy}(a) \end{vmatrix} = f_{xx}(a) f_{yy}(a) - f_{xy}^2(a)$$

$$\begin{cases} \text{if} \quad H(a) > 0 \quad \text{and} \quad f_{xx}(a) > 0 \quad \text{then } f(a) \text{ is a local minimum} \\ \text{if} \quad H(a) > 0 \quad \text{and} \quad f_{xx}(a) < 0 \quad \text{then } f(a) \text{ is a local maximum} \\ \text{if} \quad H(a) < 0 \qquad\qquad\quad \text{then } f(a) \text{ is a saddle point.} \end{cases}$$

The Laplacian

The *Laplacian* Δf is defined by

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}.$$

For example, if $n = 2$, $f = f(x, y)$, then $\Delta f = f_{xx} + f_{yy}$.

Example

Consider $f(x, y) = -2x^2 - y^2$, see figure 5.7. The partial derivatives are $f_x = -4x$ and $f_y = -2y$, so the gradient is $\nabla f = (-4x, -2y)$, and the Hamiltonian is $(-2y, 4x)$. The length of the gradient equals $\sqrt{16x^2 + 4y^2}$, so it is zero at $(0, 0)$, and grows as we move away from $(0, 0)$. The Laplacian Δf equals -6.

The isophotes (the integral curves of the Hamiltonian) can be found by setting $f(x, y) = c$, which leads to $y = \pm\sqrt{-c - 2x^2}$.

There is only one stationary point: $(0, 0)$. Since $f_{xx} = -4$ and the Hessian $H = 8$, this is a maximum.

What is especially important for image processing in this section are the definitions of gradient, flowline, and isophote.

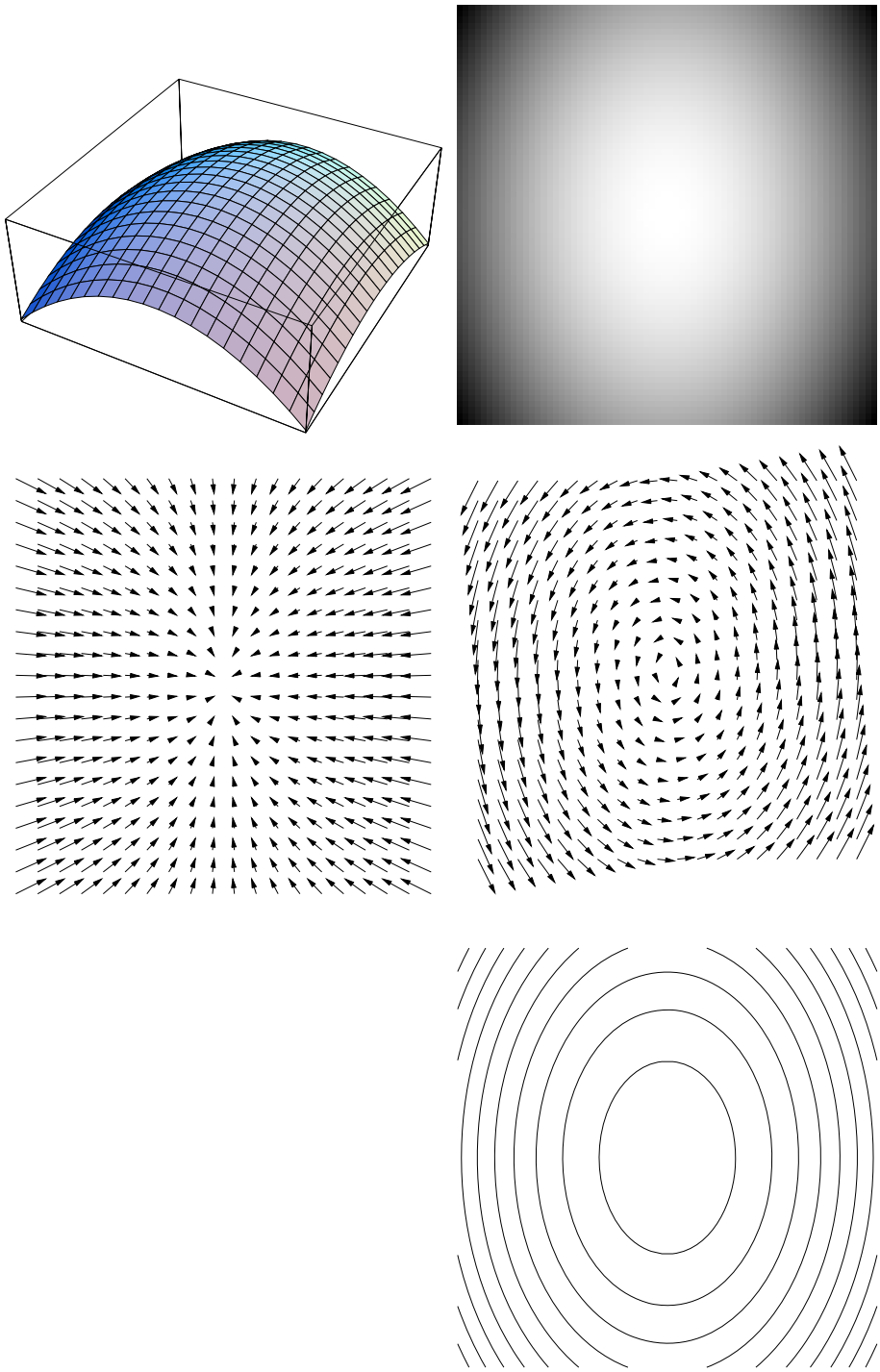


Figure 5.7 Example of basic differential geometry applied to $f(x, y) = -2x^2 - y^2$. Top left: original function. Top right: displayed as an image. Middle left: some vectors from the gradient vector field. Notice that they always point "uphill". Middle right: some vectors from the Hamiltonian vector field. Bottom right: some isophotes, *i.e.,* curves of equal intensity. Note how the isophotes are the integral curves of the Hamiltonian. The picture of the flowlines is missing because they cannot be computed analytically in this case.

5.1.3.2 Approximating derivatives

Now that we have some use for derivatives in terms of gradients, isophotes, and flowlines, we will introduce a way of approximating derivatives in images.

Suppose we have five equidistant samples of a one-dimensional function $f(x)$ and we wish to approximate the local first derivative $f'(x)$ using these five samples. Without loss of generality we assume the samples to be taken around zero with unit distance, *i.e.,* we have sampled $f(-2)$, $f(-1)$, $f(0)$, $f(1)$, and $f(2)$. Now we approximate the first derivative by f'_a, a linear weighted combination of our five samples, *i.e.,* $f'_a = g_{-2}f(-2) + g_{-1}f(-1) + \ldots + g_2 f(2) = \sum_{i=-2}^{2} g_i f(i)$. To find appropriate values for the weights g_i we demand that f'_a is exact if f equals one of the polynomials from the following set: $\{1, x, x^2, x^3, x^4\}$. This results in a system of five equations for the five unknowns g_i (each row corresponds to a polynomial, and the right hand side shows the correct value of the derivative at $x = 0$):

$$\left\{\begin{array}{rcrcrcrcrcl} 1g_{-2} & + & 1g_{-1} & + & 1g_0 & + & 1g_1 & + & 1g_2 & = & 0 \\ -2g_{-2} & + & -1g_{-1} & + & 0g_0 & + & 1g_1 & + & 2g_2 & = & 1 \\ 4g_{-2} & + & 1g_{-1} & + & 0g_0 & + & 1g_1 & + & 4g_2 & = & 0 \\ -8g_{-2} & + & -1g_{-1} & + & 0g_0 & + & 1g_1 & + & 8g_2 & = & 0 \\ 16g_{-2} & + & 1g_{-1} & + & 0g_0 & + & 1g_1 & + & 16g_2 & = & 0 \end{array}\right.$$

or equivalently

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \\ 16 & 1 & 0 & 1 & 16 \end{pmatrix} \begin{pmatrix} g_{-2} \\ g_{-1} \\ g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Solving and substituting gives:

$$f'_a(0) = \frac{1}{12}\Big(f(-2) - 8f(-1) + 0f(0) + 8f(1) - f(2)\Big).$$

This result can easily be translated to images: the x-derivative of an image can be approximated using convolution with the 5×1 kernel

$$\frac{1}{12} \cdot \boxed{\begin{array}{|c|c|c|c|c|} \hline 1 & -8 & 0 & 8 & -1 \\ \hline \end{array}},$$

and the y-derivative with the 1×5 kernel

$$\frac{1}{12} \cdot \begin{array}{|c|}\hline 1 \\ \hline -8 \\ \hline 0 \\ \hline 8 \\ \hline -1 \\ \hline \end{array}.$$

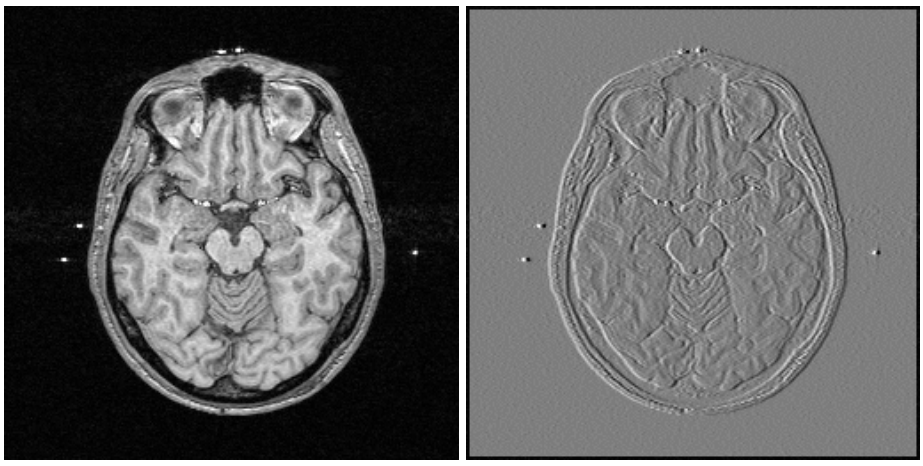


Figure 5.8 Example of the x-derivative f_x of an image f approximated by a 5×1 kernel as defined in the text.

Figure 5.8 shows an example of the x-derivative of an image approximated by the 5×1 kernel.

We can use the same technique to find kernels of different sizes for approximating derivatives, see the example below.

Example

If we want to approximate the first derivative using three pixels and demand it to be exact for the three polynomials $\{1, x, x^2\}$, we solve

$$\begin{cases} 1g_{-1} + 1g_0 + 1g_1 = 0 \\ -1g_{-1} + 0g_0 + 1g_1 = 1 \\ 1g_{-1} + 0g_0 + 1g_1 = 0 \end{cases},$$

or equivalently

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} g_{-1} \\ g_0 \\ g_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{5.6}$$

which leads to the kernel $\frac{1}{2}\cdot$ | -1 | 0 | 1 |. The second derivative can be approximated by setting the right hand side in equation 5.6 to $(0,0,2)^{\mathrm{T}}$, which leads to the kernel | 1 | -2 | 1 |.

Approximating cross-derivatives like f_{xy} takes a little more work, –since we cannot approximate these with flat one-dimensional kernels– but the principle remains the same, see the example below.

Example

Suppose we want to approximate f_{xy} using a 3×3 kernel g. As before, we center the kernel around $(0,0)$, and assume a unit distance between pixels. If we label our kernel entries as

$$g = \begin{array}{|c|c|c|}\hline g_1 & g_2 & g_3 \\ \hline g_4 & g_5 & g_6 \\ \hline g_7 & g_8 & g_9 \\ \hline \end{array},$$

and we demand the result to be exact for the nine functions

$$\{1, x, y, xy, x^2, y^2, x^2y, xy^2, x^2y^2\},$$

then the system to solve becomes

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ -1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \\ g_6 \\ g_7 \\ g_8 \\ g_9 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{5.7}$$

which after some work leads to

$$g = \frac{1}{4} \cdot \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 0 & 0 & 0 \\ \hline -1 & 0 & 1 \\ \hline \end{array}.$$

Note that we did not include the functions x^3 or y^3 in the list. This is because their corresponding rows would be the same as the rows corresponding to respectively x and y, which would degenerate the equations.

Also note that this approach with a 3×3 kernel gives consistent results with the 3×1 kernels of the previous example. For example, if we want to approximate f_x, we replace the right hand side of equation 5.7 with $(0, 1, 0, 0, 0, 0, 0, 0, 0)^{\mathrm{T}}$, which leads to

$$g = \frac{1}{2} \cdot \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline -1 & 0 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array},$$

which is equivalent to what we had already found before.

5.1.4 Edge detection

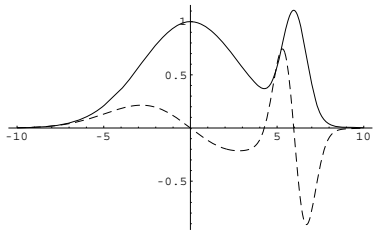
Edges are fundamental in many image processing tasks. Objects are defined by their edges, and hence edge *detection* plays an important role in finding and identifying objects.

Edges are easily located if they are "step" edges, as seen in figure 5.9, but step edges are rare occasions in many types of images. More often than not the edge of an object is not a step function, but a smoother transition of grey values across a number of pixels.

Edge pixels can be characterized as the pixels with a large local contrast, *i.e.*, regarding the image as a function, edge locations have a large slope, *i.e.*, a large (absolute) derivative. Edges are therefore often defined as the location where the derivative has a maximum or minimum, *i.e.*, where the second derivative is zero.

Example

The image below shows an example function (solid line) and its derivative (dashed line).



The sharp edge on the right side coincides with an extremum (minimum) of the derivative. On the left side of the graph the edge is much smoother, but the maximum of the derivative, *i.e.*, the locus of the steepest slope (at approximately -2.8), corresponds nicely to where most humans would place the edge point.

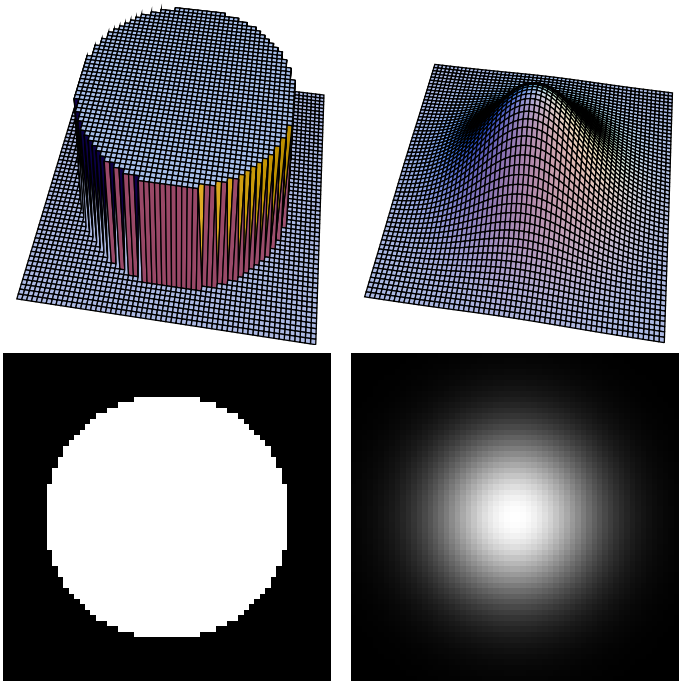


Figure 5.9 Example of a step object edge (left) and a smoother edge (right).

The derivative gives a good indication of the strength of the edge (the "edgeness"): sharp, steep edges have a large associated derivative value, and smooth edges have only a small derivative value.

For a two-dimensional image (say f), the slope of the grey value function in a pixel a depends on the direction in which you are looking: there is always a direction in which the slope is zero –the direction of the local isophote– and there is a direction in which the slope is maximum; the direction of the gradient $\nabla f(a) = (f_x(a), f_y(a))$. The maximal slope equals the norm of the gradient: $\|\nabla f(a)\| = \sqrt{f_x^2(a) + f_y^2(a)}$. An image showing the norm of the gradient[2] gives a good indication of edgeness, see figure 5.10. What we define as edge pixels can be extracted from this image by thresholding it.

We can compute the gradient image by using the techniques from the previous section, *i.e.*, f_x and f_y can be approximated using the kernels derived there, *e.g.*, the f_x image can be approximated by convolving the image with the kernel $\frac{1}{2}\cdot$ | -1 | 0 | 1 |.

The second derivative can also be used for finding edges: if we define an edge in a one-dimensional signal as the locus where the first derivative is maximum (or minimum),

[2] Usually –not entirely correctly– simply called a gradient image.

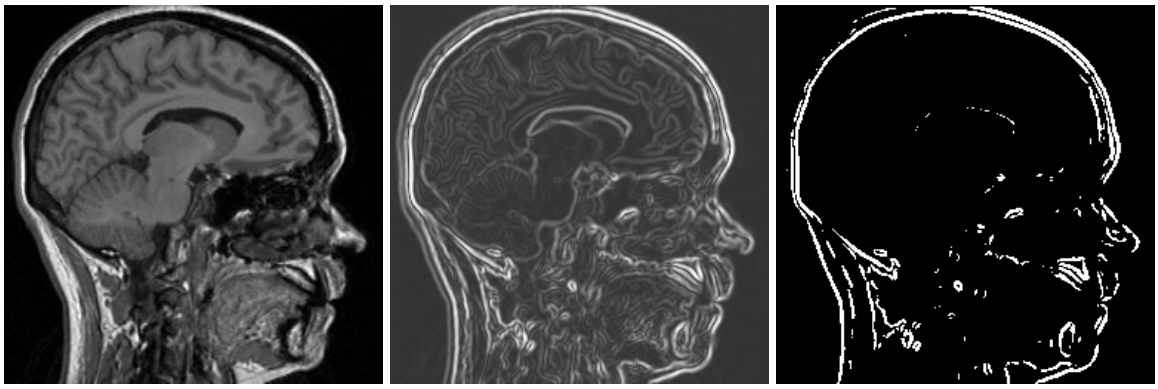


Figure 5.10 Example of edge detection. Left: original image. Middle: norm of the gradient. Right: after thresholding the gradient image.

then the second derivative will be zero at this locus. We can therefore locate edges by finding the zero-crossings of the second derivative. The two-dimensional equivalent of this for images is to find the zero-crossings of the Laplacian $\Delta f = f_{xx} + f_{yy}$. Figure 5.11 shows an example of this.

When computing edgeness images by approximation of the gradient norm $\|\nabla f\| = \sqrt{f_x^2 + f_y^2}$, we can often improve the results by using kernels slightly modified from the ones presented before for computing derivatives. For instance, consider these pairs of kernels:

	k_1	k_2
3×3 approximation:	$\frac{1}{2}\cdot$	$\frac{1}{2}\cdot$
Prewitt:		
Sobel:		
Isotropic:		

0	0	0
-1	0	1
0	0	0

0	-1	0
0	0	0
0	1	0

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

-1	0	1
$-\sqrt{2}$	0	$\sqrt{2}$
-1	0	1

-1	$-\sqrt{2}$	-1
0	0	0
1	$\sqrt{2}$	1

In all three cases, we can compute an edgeness image g of an image f by computing $g = \sqrt{(k_1 \star f)^2 + (k_2 \star f)^2}$. Note that with every kernel the entries sum up to zero, so there will be no response in flat image areas. Ignoring all multiplicative factors, the

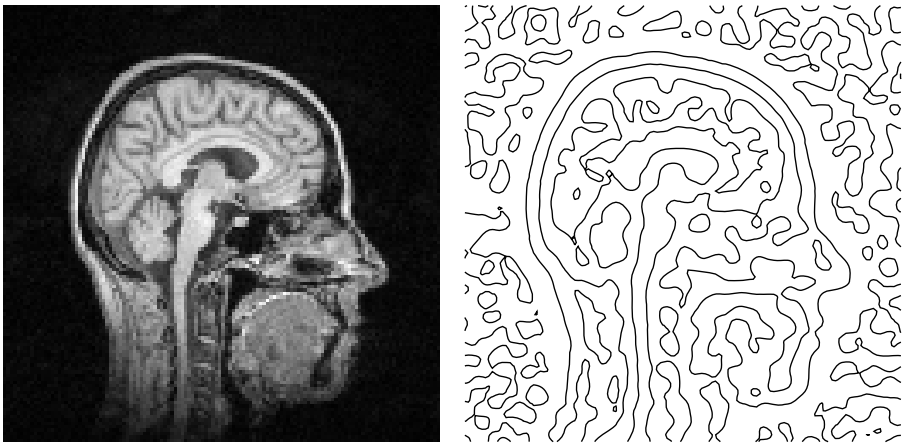


Figure 5.11 Example of edge detection. Left: original image. Right: zero-crossings of the Laplacian.

Prewitt kernels compute the average of the derivative across three image lines. The effect of this (relative to computing the derivatives by using the top two kernels) is that the result is less influenced by noise and very small structures. The *Sobel* kernels also compute the average derivative across three lines, but in a weighted way: the weight attached to the center line derivative is twice that of the weight attached to the other two lines. Figure 5.12 shows an example of an edgeness image computed using the Sobel kernels.

5.1.4.1 Edge orientation

The separate results $k_1 \star f = g_1$ and $k_2 \star f = g_2$ in the computation of edgeness using a pair of kernels form a vector (g_1, g_2) when combined. The magnitude $\sqrt{g_1^2 + g_2^2}$ of this vector is used for the edgeness image. The vector also has an orientation angle $\theta = \arctan(\frac{g_2}{g_1})$, as shown in figure 5.13. The orientation angle is a useful quantity in many image processing tasks where we are only interested in edges oriented in specific directions.

5.1.4.2 Compass operators

Edges can also be detected by using a *compass operator*. The result of a compass operator g on an image f is defined by

$$g = \max_{i \in \{0,\ldots,7\}} |g_i|,$$

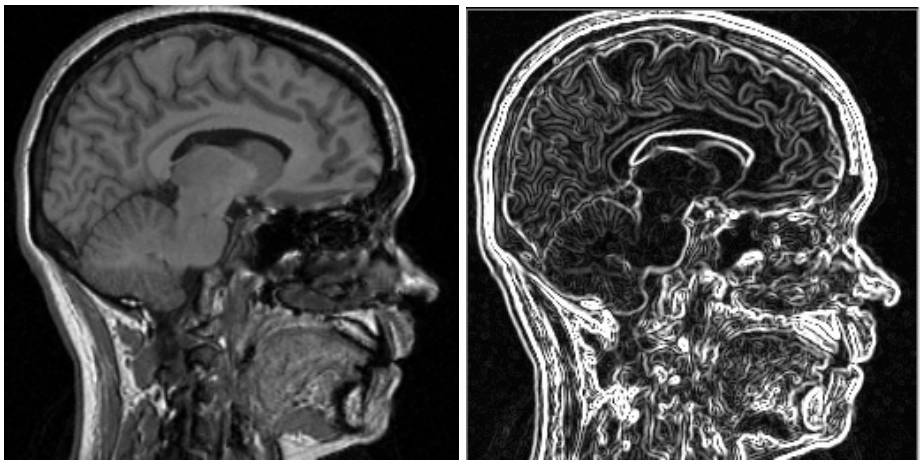


Figure 5.12 Example of edge detection. Left: original image. Right: Edgeness image computed using the Sobel kernels.

where $g_i = k_i \star f$, and k_i are rotated versions of a base kernel k. For example, if we choose a Prewitt kernel for a base kernel, the 8 rotated versions $k_0, k_1, \ldots, k_7$ would be

1	1	1
0	0	0
-1	-1	-1

1	1	0
1	0	-1
0	-1	-1

1	0	-1
1	0	-1
1	0	-1

0	-1	-1
1	0	-1
1	1	0

-1	-1	-1
0	0	0
1	1	1

-1	-1	0
-1	0	1
0	1	1

-1	0	1
-1	0	1
-1	0	1

0	1	1
-1	0	1
-1	-1	0

which measure edge strengths in the eight compass directions ($\uparrow, \nwarrow, \leftarrow, \swarrow, \downarrow, \searrow, \rightarrow, \nearrow$) respectively.

A Prewitt or Sobel kernel is often used as the base kernel for a compass operator. Another one used frequently is the *Kirsch* operator, which uses for a base kernel:

5	5	5
-3	0	-3
-3	-3	-3

.

5.1.4.3 Frei and Chen operator

Sometimes an edge detection method will erroneously indicate a high edgeness value at pixels where there is no edge to be found by human definitions. The Frei and Chen

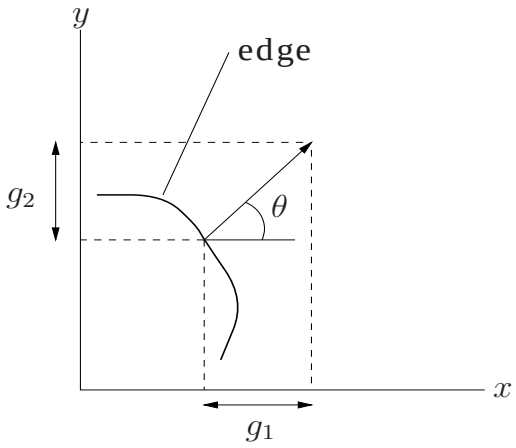


Figure 5.13 The orientation of a gradient vector as characterized by an angle θ.

operator can often remedy this. This operator uses the isotropic kernels to detect edges, but also requires that a number of other kernels have a *low* response. Nine kernels are used:

k_1

-1	0	1
$-\sqrt{2}$	0	$\sqrt{2}$
-1	0	1

k_2

-1	$-\sqrt{2}$	-1
0	0	0
1	$\sqrt{2}$	1

k_3

0	-1	$\sqrt{2}$
1	0	-1
$-\sqrt{2}$	1	0

k_4

$\sqrt{2}$	-1	0
-1	0	1
0	1	$-\sqrt{2}$

k_5

0	1	0
-1	0	-1
0	1	0

k_6

-1	0	1
0	0	0
1	0	-1

k_7

1	-2	1
-2	4	-2
1	-2	1

k_8

-2	1	-2
1	4	1
-2	1	-2

k_9

1	1	1
1	1	1
1	1	1

The kernels k_1 and k_2 are the isotropic gradient kernels we saw before. The original Frei and Chen operator plotted the results of the convolution with each kernel k_i in a nine-dimensional space and then compared the results as projected in the "edge"subspace ($i \in \{1, 2, 3, 4\}$) with the results as projected in the non-edge subspace ($i \in \{5, 6, 7, 8, 9\}$). Only if the ratio was large enough was a pixel considered an edge pixel. The method is more commonly used in a simplified way: the kernels k_3 and k_4 are not used, because their response at edges is very small compared to the k_1 and k_2 responses.

5.1.4.4 Other edge detecting methods

From the large number of presented kernels it will be clear that the ideal edge-detecting kernel does not exist. In many cases, the Sobel operator is a good choice, since it performs reasonably well while requiring little computer operations and is easy to imple-

ment. If more subtle edges are to be found the Frei and Chen operator is usually the operator of choice. For specific applications, some of the kernels and operators presented in this chapter perform better than others, but *all* of them perform badly in the presence of noise[3]. For many applications it is necessary to use more sophisticated approaches to edge detection. One such approach is the *Marr-Hildreth* operator. This operator subtracts two images that are smoothed by Gaussian kernels of different width:

$$r = (f * g_{\sigma_1}) - (f * g_{\sigma_2}),$$

where r is the result of the operation, f the original image, and g_{σ_i} is a (2D) Gaussian function with parameter σ_i ($\sigma_2 > \sigma_1$). The effect of this operator is very similar to computing the Laplacian, but it is much less sensitive to noise. A drawback is that the two parameters σ_i will have to be carefully tuned to the task at hand. Figure 5.14 shows an example of applying the Marr-Hildreth operator. Other sophisticated approaches to

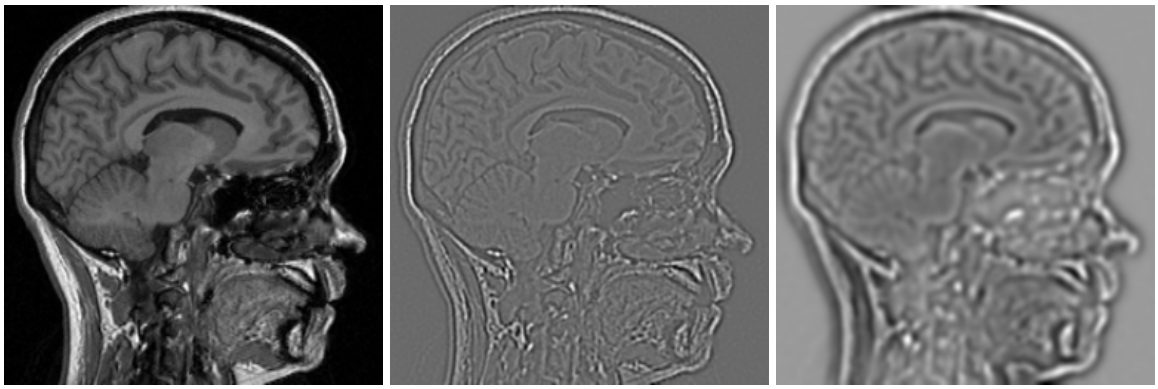


Figure 5.14 Example of the Marr-Hildreth operator compared to the Laplacian. Left: original image. Middle: Laplacian. Right: Marr-Hildreth operator.

edge detection will be covered in later chapters.

5.1.4.5 Line and isolated pixel detection

Lines can be detected using the compass operator:

k_1

-1	-1	-1
2	2	2
-1	-1	-1

k_2

-1	-1	2
-1	2	-1
2	-1	-1

k_3

-1	2	-1
-1	2	-1
-1	2	-1

k_4

2	-1	-1
-1	2	-1
-1	-1	2

[3]Even though the Laplacian is usually most sensitive and the Frei and Chen operator least sensitive to noise.

The kernels can be used like the edge detecting compass kernels:

$$g = \max_{i \in \{1,\ldots,4\}} |g_i|,$$

where $g_i = k_i \star f$. Note that the entries of each kernel sum up to zero, so there is no response in flat image areas. The highest kernel response will occur if a one-pixel thick line is oriented along the row of twos in the kernels, but an edge (oriented along the twos) will also give a positive response. Figure 5.15 shows an example of applying this line operator.

Isolated points in images –*i.e.*, pixels that have a grey value significantly different from their neighbors' grey values– can be detected by (thresholding of) the convolution of an image with the kernel

-1	-1	-1
-1	8	-1
-1	-1	-1

.

However, this kernel also responds in other image areas; it has already been presented as a high pass kernel. A better approach is to use the following kernel as the base kernel for a compass operator:

0	-1	0
0	1	0
0	0	0

,

and ensure all of the rotated kernel responses have large absolute values.

5.1.5 Approximating continuous convolution

The approximation of continuous convolution by a discrete convolution is a frequently occurring task in image processing. For example, it is used in simulating image acquisition, and approximating Gaussian convolution, which is useful for noise suppression and the basis of Gaussian scale space (see chapter 9). We will use the approximation of Gaussian convolution as a working example in this section.

Figure 5.16 shows a discretization of the Gaussian kernel $g(x) = \frac{1}{2\sqrt{\pi}} e^{-\frac{x^2}{4}}$ using nine pixels. These nine values $\{g(-4), g(-3), \ldots, g(4)\}$ (approximated to three decimals) are

Figure 5.15 Example of a line detecting compass operator. Left: original image. Right: after applying the compass operator.

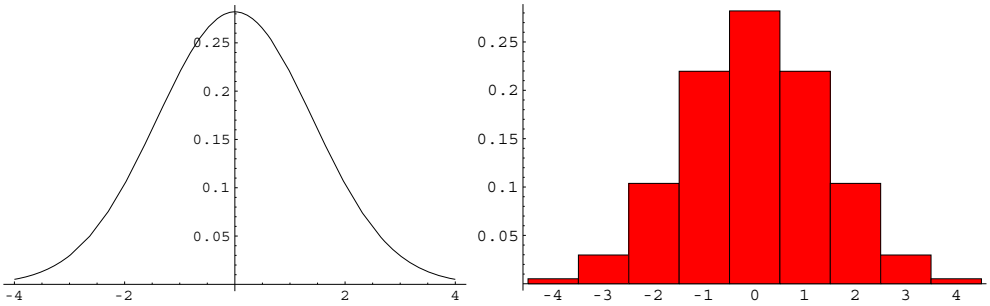


Figure 5.16 Example of a continuous Gaussian kernel $g(x) = \frac{1}{2\sqrt{\pi}} e^{-\frac{x^2}{4}}$ (left) and the same kernel discretized using 9 pixels (right).

0.005	0.030	0.104	0.220	0.282	0.220	0.104	0.030	0.005

.

These values nicely capture the shape of the Gaussian, but we cannot use them directly as a discrete kernel, because we would then loose an important property of the Gaussian kernel, namely that the area under the curve equals one. This property ensures us that flat image areas are not changed by Gaussian convolution. For the discrete kernel to have the same property we must ensure its entries sum up to one, *i.e.*, normalize it. This is achieved by including a factor equal to the inverse of the sum of entries, *i.e.*,

$\frac{1}{0.999} \cdot$

0.005	0.030	0.104	0.220	0.282	0.220	0.104	0.030	0.005

.

In this case, the factor is nearly one. Should we have used only the pixel values at $\{-2, -1, 0, 1, 2\}$, then the factor would have been 0.929.

Two types of error are involved in this kind of approximation. First a discretization error caused by the fact that we approximate (in this case) each unit length of the Gaussian curve by a single pixel value. This error can be reduced by denser sampling of the continuous pixels, *i.e.,* by using more pixels per unit length[4]. Second a truncation error because values of the kernel more than four units from the origin are not taken into account. This error can be reduced by extending the range in which the discrete kernel is sampled. In this case, the truncation is already very small: less than 0.5% of the area under the Gaussian curve is below the 'tail' ends farther than four pixels from the origin.

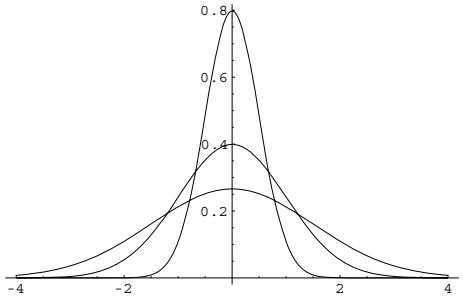


Figure 5.17 Three Gaussian kernels with $\sigma \in \{0.5, 1, 1.5\}$. The narrowest kernel has $\sigma = 0.5$.

Figure 5.17 shows three Gaussian curves with values of σ of 0.5, 1, and 1.5 respectively. If σ is small, the truncation error will be small even for small discrete kernels because the curve rapidly tends to zero as we move away from the origin. Using one sample per pixel, the discretization error will be large, because the shape of the kernel cannot be adequately captured using a few pixels around the origin. If σ is large, the discretization error will be relatively small because the shape of the kernel can be captured better. The truncation error can be kept small by choosing a large discrete kernel size. In practice, the kernel size is often chosen depending on the value of σ, *e.g.,* the kernel size is at least 3σ.

The approximation technique in the above is easily extended to two and more dimensions, as the example below shows

Example

Suppose we want to approximate convolution with the 2D Gaussian function

$$g(x,y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)}$$

with $\sigma = 2$ using a 9×9 discrete kernel. Filling the kernel is done by computing the values of $g(x,y)$ for $(x,y) \in \{-4,-3,\ldots,4\} \times \{-4,-3,\ldots,4\}$, which leads to (using two digits of accuracy):

[4]With digital images using discrete convolution, however, there is not much sense in sampling denser than the pixel size, *i.e.,* to use more than one sample per pixel.

0.00073	0.0017	0.0033	0.0048	0.0054	0.0048	0.0033	0.0017	0.00073
0.0017	0.0042	0.0078	0.011	0.013	0.011	0.0078	0.0042	0.0017
0.0033	0.0078	0.015	0.021	0.024	0.021	0.015	0.0078	0.0033
0.0048	0.011	0.021	0.031	0.035	0.031	0.021	0.011	0.0048
0.0054	0.013	0.024	0.035	0.04	0.035	0.024	0.013	0.0054
0.0048	0.011	0.021	0.031	0.035	0.031	0.021	0.011	0.0048
0.0033	0.0078	0.015	0.021	0.024	0.021	0.015	0.0078	0.0033
0.0017	0.0042	0.0078	0.011	0.013	0.011	0.0078	0.0042	0.0017
0.00073	0.0017	0.0033	0.0048	0.0054	0.0048	0.0033	0.0017	0.00073

Summing the values found amounts to 0.95456, so the multiplication factor for this kernel should be $\frac{1}{0.95456}$.

The truncation error is still fairly large using a 9×9 kernel, since $\int_{-4}^{4}\int_{-4}^{4} g(x,y)\,dxdy \approx 0.911$, so about 9% of the area under the kernel is outside of the window.

Note that we can make use of the symmetry of the Gaussian in filling the discrete kernel; we only have to compute the values in one quarter of the kernel, the rest of the values can be filled in symmetrically.

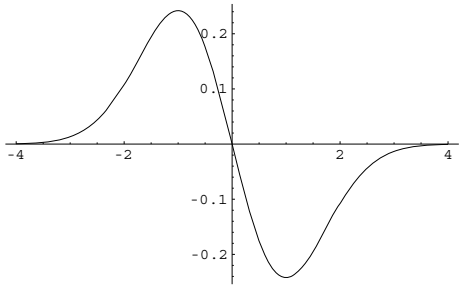
Most convolution with "acquisition type" kernels –*i.e.*, those that have an area of one under their curve and tend to zero away from the origin– can be approximated by discrete convolution in the same way as we did in the above for Gaussian convolution, *i.e.*, by sampling values in a finite discrete region, normalizing the values found, and placing the results in a discrete kernel. The discretization and truncation errors should be re-assessed for each new type of kernel.

Many "feature-extracting" kernels have the property that the area under the kernel is zero, *i.e.*, they don't respond in flat image areas. If the kernel is odd[5] very similar techniques as before can be used, see the example below. A difficulty only arises when a correct normalization factor must be found.

Example

Suppose we have the following feature-extracting kernel

$$f'(x) = -\frac{x}{\sigma^3\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x^2}{\sigma^2}\right)},$$



[5] *i.e.*, $g(-x) = -g(x)$; the curve is point-symmetric in the origin.

i.e., the derivative f' of the Gaussian

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x^2}{\sigma^2}\right)}.$$

Suppose we wish to approximate a continuous convolution with the kernel f' with $\sigma = 1$ by a discrete convolution using a 9×1 kernel. We can find the appropriate kernel entries $f_a(x)$ by evaluating f' at $x \in \{-4, -3, \ldots, 4\}$ (using two digits of precision):

0.00054	0.013	0.11	0.24	0	-0.24	-0.11	-0.013	-0.00054

.

The kernel entries sum up to zero, which is a desired property. This means that the area under the curve of f' (which is zero) is preserved regardless of a normalization factor or the size of the discrete kernel. Because the "outer" kernel entries are very small the truncation error is very small.

Finding an appropriate normalization factor is not straightforward because the property of preservation of area under the kernel is already met. In these cases it is common to extent the property $\int f(x)\,dx = 1$ to the kernel $f'(x)$: by partial integration we find

$$\int f(x)\,dx = \int f(x) \cdot 1\,dx = [f(x)x] - \int xf'(x) = \int -xf'(x)\,dx,$$

where the term $[f(x)x]$ is zero because $f(x)x$ is an odd function. In other words, we can find a good value for the normalization factor p by setting $\int -xf'(x)\,dx = 1$. The discrete equivalent of this in our example would be

$$p\Big((4f_a(-4)) + (3f_a(-3)) + \ldots + (-4f_a(4))\Big) = 1,$$

Which leads to $p = 1.00007$. Below are some values of p for kernels of different sizes.

size	p
3	2.06637
5	1.09186
7	1.00438
9	1.00007

5.1.6 More on kernels

Convolution has many properties similar to multiplication properties:

Commutative law:	$f * g = g * f$
Associative law:	$(f * g) * h = f * (g * h)$
Distributive law:	$(g + h) * f = g * f + h * f.$

But this is where the similarity ends. For instance, $f * f$ can have negative values, and $f * 1$ does not generally equal f. Note that these laws do *not* apply to the $\star$ operation, except for the distributive law.

The distributive law is useful in image applications where we need to compute two convolutions of one input image, and then add (or subtract) the results. According to the distributive law, we can get the same result by convolving once with the sum (or difference) of the original convolution kernels, which is usually a much cheaper operation.

Example

Given an image f, we can compute

$$\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 5&0&0\\\hline 0&4&0\\\hline\end{array} \star f + \begin{array}{|c|c|c|}\hline 2&3&4\\\hline 0&0&5\\\hline 0&0&6\\\hline\end{array} \star f$$

in one go using

$$\begin{array}{|c|c|c|}\hline 3&5&7\\\hline 5&0&5\\\hline 0&4&6\\\hline\end{array} \star f.$$

The associative law is useful in those cases where a succession of *continuous* convolutions needs to be computed of an image. It is often cheaper to first convolve all the convolution kernels with each other, and then convolve the image with the resultant kernel.

Example

If we wish to compute

$$f * g_{\sigma_1} * g_{\sigma_2},$$

where $g_{\sigma_i} = \frac{1}{\sigma_i\sqrt{2\pi}}e^{-\frac{1}{2}\frac{x^2}{\sigma_i^2}}$ are Gaussian kernels, and f is an image, then we can compute the same result with one image convolution by

$$f * h,$$

where

$$h = g_{\sigma_1} * g_{\sigma_2} = \frac{1}{\sqrt{2\pi}\sqrt{\sigma_1^2+\sigma_2^2}}e^{-\frac{1}{2}\frac{x^2}{\sigma_1^2+\sigma_2^2}}.$$

The last example shows that the convolution of two Gaussians is again a Gaussian (with σ equal to the square root of the sum of squares of the original two sigmas). This relation is not so easy to establish using standard calculus. In chapter 7 the *Fourier transform* will be introduced, a technique that greatly facilitates the computation of continuous convolutions. However, for discrete kernels, the convolution of two kernels can often be easily computed, as the example below shows.

Example

Suppose we wish to compute $g \star (g \star f)$ using only one image convolution, with f an image, and $g = \boxed{0}\boxed{1}\boxed{2}$. We cannot directly use the associative law, since it applies to the $*$ operator, and not directly to the $\star$ operator. First we need to write our expression in a form using $*$. This can be done by using the mirror relation

$$g_m * f = g \star f,$$

where g_m is g mirrored in the origin, *i.e.*, $g_m = \boxed{2}\boxed{1}\boxed{0}$. So $g \star (g \star f) = g \star (g_m * f) = g_m * g_m * f$. To compute this last expression using only one image convolution, we need to compute $g_m * g_m$. Using the mirror relation again: $g_m * g_m = g \star g_m$. We can compute this last expression (using zero padding, *i.e.*, assuming unknown values to be zero, and always taking the center pixel of the kernel for the origin):

$$\boxed{0}\boxed{1}\boxed{2} \star \boxed{2}\boxed{1}\boxed{0} = \boxed{4}\boxed{4}\boxed{1}\boxed{0}\boxed{0}.$$

Using the mirror relation one last time:

$$\boxed{4}\boxed{4}\boxed{1}\boxed{0}\boxed{0} * f = \boxed{0}\boxed{0}\boxed{1}\boxed{4}\boxed{4} \star f.$$

So the operation $g \star (g \star f)$ can be carried out using one convolution with the formula

$$\boxed{0}\boxed{0}\boxed{1}\boxed{4}\boxed{4} \star f.$$

The above example illustrates how to arrive at the associative relationship for the $\star$ operator using the mirror relation $g_m * f = g \star f$:

$$h \star (g \star f) = (h \star g_m)_m \star f.$$

This example *en passant* shows that the convolution of two kernels generally leads to a larger kernel (or –at least– to more non-zero kernel values). Although this makes perfect sense from a mathematical point of view, it may be surprising here, as it is *not* common practice to enlarge *images* when applying a convolution operation. In the case of convolving two kernels it is usually a necessity, to avoid the loss of information.

For a second example of combining two convolutions into one convolution, we will show how a high-frequency enhancing operation and a derivative can be combined.

Example

If we want to compute $h \star (g \star f)$, where f is an image, g is the high-frequency enhancing kernel

-1	-1	-1
-1	9	-1
-1	-1	-1

, and h is the x-derivative kernel $\frac{1}{2}$

-1	0	1

, we can use one image convolution by computing $(h \star g_m)_m \star f$, *i.e.,* (note that $g_m = g$)

$$\left(\frac{1}{2} \cdot \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline \end{array} \star \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 9 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array} \right)_m \star f,$$

which equals

$$\frac{1}{2} \cdot \begin{array}{|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & -1 & -1 \\ \hline 1 & -9 & 0 & 9 & -1 \\ \hline 1 & 1 & 0 & -1 & -1 \\ \hline 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \star f.$$

In general it is *not* cheaper to combine successive *discrete* convolutions as one convolution. For example, when computing $h \star g \star f$ using two successive convolutions, with h and g 3×3 kernels, we require $2 \times 9 = 18$ multiplications and additions per pixel. If we used one 5×5 kernel $(h \star g_m)_m$, we would need 25 multiplications and additions per pixel. It is therefore useful to see if a discrete convolution operation can be carried using smaller kernels.

Example

The averaging kernel $g = \begin{array}{|c|c|c|}\hline 1&1&1\\\hline 1&1&1\\\hline 1&1&1\\\hline\end{array}$ can be decomposed as $g = g_1 \star g_2$, where $g_1 = \begin{array}{|c|c|c|}\hline 1&1&1\\\hline\end{array}$ and $g_2 = \begin{array}{|c|}\hline 1\\\hline 1\\\hline 1\\\hline\end{array}$. Computing $g_2 \star (g_1 \star f)$ takes less multiplications and additions per pixel (6) than computing $g \star f$ does (9).

Separable kernels are kernels that can be separated into functions of a lower dimension, *e.g.*, $g(x, y) = g_1(x)g_2(y)$. Convolution with a separable kernel can be split into separate one-dimensional convolutions. These one-dimensional convolutions together are generally considerably cheaper than computing one multi-dimensional convolution. For instance, in the 2D case:

$$\begin{aligned} g * f &= \iint f(a,b)g(x-a,y-b)\,da\,db = \\ &\iint f(a,b)g_1(x-a)g_2(y-b)\,da\,db = \\ &\int \left[\int f(a,b)g_1(x-a)\,da\right] g_2(y-b)\,db. \end{aligned}$$

Since a large number of kernels occurring in practical and theoretical image processing are separable, computing a convolution by separate one-dimensional convolutions is a much-used technique.

5.2 Simple non-linear filtering

Most of the filters we have seen so far in this chapter are *linear* filters; the filter result r at a location is a linear function of the grey values f_i in a neighborhood:

$$r = \sum_i w_i f_i,$$

where w_i are weights, *i.e.*, the entries in a discrete convolution kernel. Other filters, such as the compass filters, use discrete convolution, but the final result of the operator is not a linear function of the grey values. Non-linear filters can by definition not (fully) be

implemented using discrete convolution. An important class of non-linear filters makes use of the order of grey values in a neighborhood, *e.g.*, filters that take the maximum, minimum, or median value of grey values in a neighborhood. Such filters that first order the neighborhood grey values and then select a value at a specific place in the list are called *rank* filters. Filters that use the maximum and minimum grey values in a neighborhood are discussed in depth in chapter 6 on mathematical morphology. The result of the *median filter* equals the median of the ordered grey values in a neighborhood.

Example

Examples of the effect of a 3×3 median filter on simple images:

0	0	0	0	0
0	0	0	1	1
0	0	1	1	1
0	0	1	20	1
0	0	1	1	1

$\longrightarrow$

	0	0	1	
	0	1	1	
	0	1	1	

0	1	2	3	4
5	6	7	8	9
5	5	5	9	9
5	5	5	9	9
5	5	5	9	9

$\longrightarrow$

	5	5	7	
	5	6	9	
	5	5	9	

The median filter is often used to remove speckle noise from an image. See for instance the first example above, where the noisy outlier value of 20 is removed from the resultant image. However, the median filter also removes other small details. An example of applying a median filter to a real image is shown in figure 5.18. An interesting effect of the median filter is that edges are not shifted, even if the median filter is repeated a large number of times. Figure 5.18 shows that repeated application of the median filter removes small structures, but that larger structures are preserved. Another effect is that structures are homogenized; the number of contained grey values is reduced.

5.3 Handling image borders

In the examples in the above sections the results of discrete convolution and other neighborhood operations “lose their border”. This is of course undesired in many applications. The border loss is caused by the fact that convolution results at border pixels requires knowledge of grey values at pixel locations that are outside of the image:

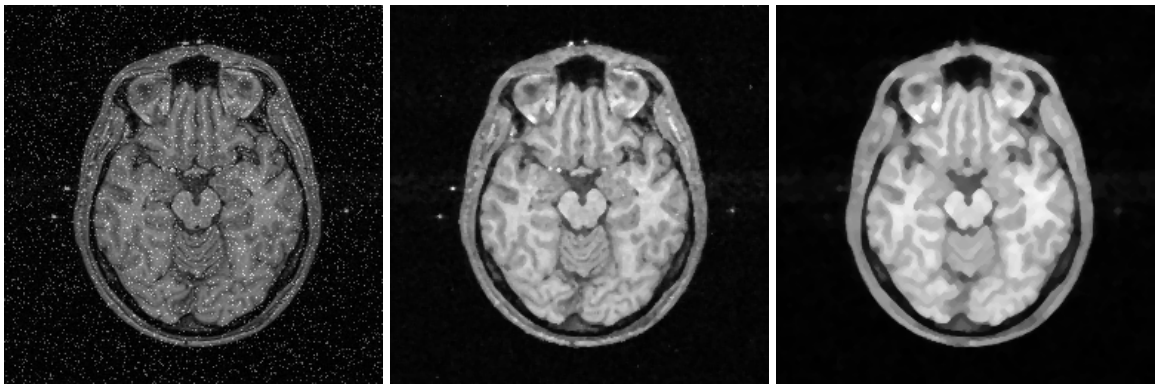
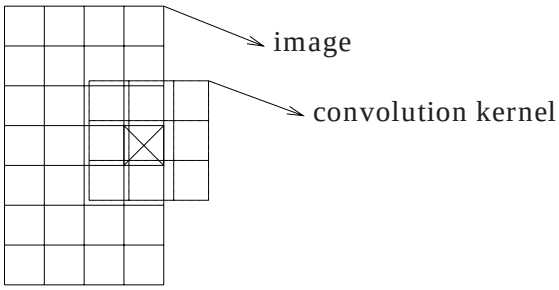


Figure 5.18 Example of applying a 3×3 median filter. The left image shows a real image with added noise. The middle and right image show this image after applying the median filter once and twelve times respectively.

Example



In this example, the convolution result at the marked pixel requires three pixel values that are located outside of the image.

The border loss increases with larger kernels: one pixel thick with a 3×3 kernel, two pixels with a 5×5 kernel, *etc.*

Although there is no way of establishing the unknown pixel values, we can avoid border loss by *guessing* these values. Three basic approaches exist for this:

- **Padding.** In this case the image is padded at the borders with extra pixels with a fixed grey value. In other words, a fixed value is substituted for the unknown grey values. Usually this is a value common to the image background. In some cases of non-linear filtering it may be useful to pad with a value of $+\infty$ or $-\infty$.
- **Extrapolation.** In this case a missing grey value is estimated by extrapolating it from the grey values near the border. In the simplest case the nearest known pixel value is substituted for the unknown one. In more complex cases a line, a surface,

or a more complex structure is fitted through values near the border, and unknown values are estimated by extrapolating this structure.

- **Mirroring.** In this case a mirror image of the known image is created with the border for a mirroring axis.

Example

Suppose we have an image like the one below, and we need to guess the values in the two empty spaces on the right.

0	1	2		
0	1	2		
0	1	2		

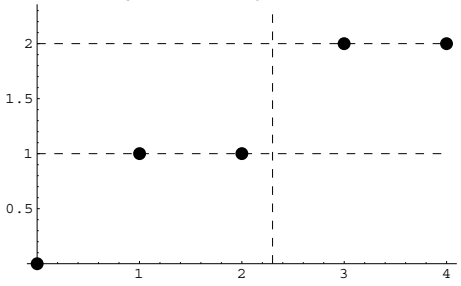
Using zero-padding these values are 0. Simple extrapolation gives values of 2. Extrapolation by fitting a line perpendicular to the border through the nearest image values would give values of 3 and 4 respectively. Mirroring the nearest grey values in the borderline gives values of 2 and 1.

5.4 Interpolation

Many image processing applications require knowledge of grey values at non-pixel locations, *i.e.*, at locations between the grid positions with known grey values. These values need to be estimated (interpolated) from the grey values in a neighborhood. Two common interpolation techniques are *nearest neighbor* and *linear* interpolation. Nearest neighbor interpolation assigns to a location the grey value of the nearest pixel location. Linear interpolation uses a weighted combination of the grey values of the nearest pixels, where a weight w is related to the distance d to a pixel location by $w = 1 - d$.

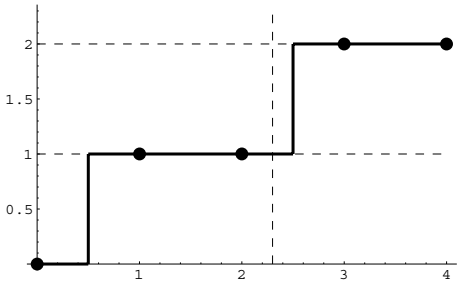
Example: nearest neighbor interpolation

Consider the signal with values $\{0, 1, 1, 2, 2\}$ at the locations $x \in \{0, 1, 2, 3, 4\}$:



Suppose we wish to interpolate the value at 2.3 (dashed vertical line) by nearest neighbor interpolation, then this value would be 1, since the nearest point ($x = 2$) has value 1.

This graph shows the values at $x \in [0, 4]$ as obtained by nearest neighbor interpolation:



For a 2D image example, consider this image:

0	1	2	3
4	5	6	7
8	9	8	7
6	5	4	3

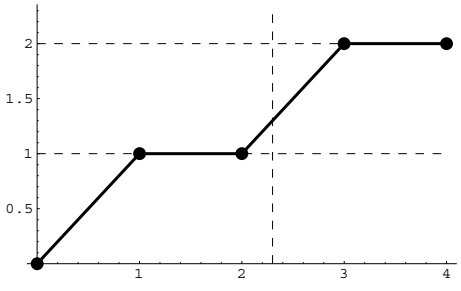
If we choose the origin $(0, 0)$ to be in the center of the top left pixel, then the grey values represent the values at the pixel locations $\{(0, 0), (1, 0), \ldots, (3, 3)\}$. The interpolated value at (*e.g.*) $(0.8, 0.8)$ would be 5, since the nearest pixel, $(1, 1)$, has grey value 5.

Example: linear interpolation

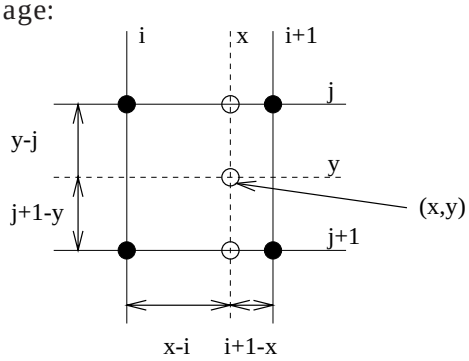
Using the same signal from the previous example, the value v at $x = 2.3$ by linear interpolation can be determined from the nearest points ($x = 2$ and $x = 3$) by a weighted average with weights $(1 - d)$, where d is the distance to the appropriate point:

$$v(2.3) = (1 - 0.3)v(2) + (1 - 0.7)v(3) = 1.3$$

This graph shows the values as obtained by linear interpolation:



For 2D images, we carry out this interpolation twice; once for each direction. Consider this part of an image:



Suppose we wish to find an interpolated value v for (x, y) which is located in the square of pixels $\{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$ as shown above. First we interpolate in the x-direction (top and bottom open circles):

$$\begin{cases} v(x,j) & = & (1-(x-i))v(i,j) & + & (1-(i+1-x))v(i+1,j) \\ v(x,j+1) & = & (1-(x-i))v(i,j+1) & + & (1-(i+1-x))v(i+1,j+1) \end{cases}$$

Second, we interpolate the values found in the y-direction:

$$v(x,y) \quad = \quad (1-(y-j))v(x,j) \quad + \quad (1-(j+1-y))v(x,j+1).$$

These two steps can be combined in a single formula for $v(x, y)$. Linear interpolation in two dimensions is usually called *bilinear* interpolation. Figure 5.19 shows an example of bilinear interpolation applied to a real image.

The linear interpolation result shows a graph or an image that is continuous, but is not smooth at the grid points, *i.e.*, the derivative is not continuous. By fitting higher order curves to the grid points and / or demanding the derivative (and possibly higher order derivatives) to be continuous, we can obtain a smoother interpolation result. Examples of curves that satisfy these conditions are the so-called *B-spline* and *thin-plate spline* functions.

5.4.1 Interpolation by convolution

For theoretical –and sometimes practical– purposes it is useful to model interpolation by a convolution process. This model also shows the relation between nearest neighbor and linear interpolation.

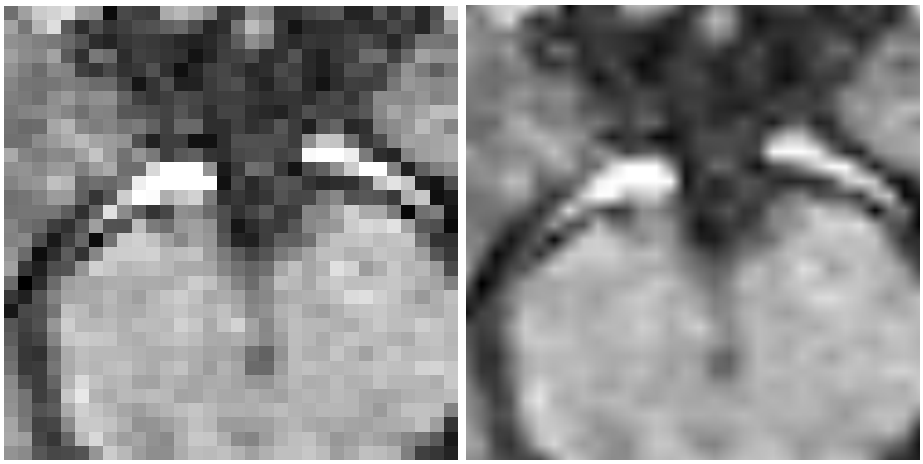


Figure 5.19 Example of bilinear interpolation: the left image, with a 30×30 resolution has been bilinearly interpolated on a 256×256 grid (right image).

A two-dimensional digital image f only has known values at the grid points (i, j) with $(i, j) \in \mathbb{N} \times \mathbb{N}$. We can model the continuous interpolated image $f(x, y)$ by a convolution

$$f(x, y) = \sum_i \sum_j f(i, j) g(x - i, y - j),$$

where g is a *continuous* kernel. We can use a summation instead of an integration because f is only defined at grid points. To simplify matters, we assume that the kernel g is separable, $g(x, y) = h(x)h(y)$ (an assumption that is almost always true in practice), so we only need to examine the one-dimensional case

$$f(x) = \sum_i f(i) h(x - i).$$

A desired property for interpolating kernels is:

$$h(i) = \begin{cases} 0 & \text{if} \quad i \in \mathbb{Z} \setminus \{0\} \\ 1 & \text{if} \quad i = 0. \end{cases}$$

This property ensures that the value of $f(x)$ is not altered (equals $f(i)$) at grid points.

In the case of nearest neighbor interpolation, h equals the kernel h_1 with

$$h_1(x) = \begin{cases} 1 & \text{if} \quad |x| < \frac{1}{2} \\ 0 & \text{elsewhere.} \end{cases}$$

In the case of linear interpolation, h equals the kernel h_2 with

$$h_2(x) = \begin{cases} 1 - |x| & \text{if } |x| < 1 \\ 0 & \text{elsewhere.} \end{cases}$$

There is a relationship between h_1 and h_2: $h_2 = h_1 * h_1$.[6] It is therefore interesting to examine the kernels $h_3 = h_1 * h_1 * h_1$, $h_4 = h_1 * h_1 * h_1 * h_1$, *etc.* Figure 5.20 shows h_i with $i \in \{1, 2, 3, 4\}$. In formulae:

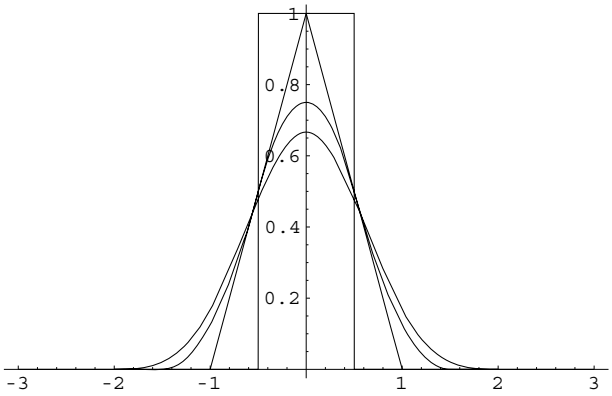


Figure 5.20 The interpolation kernels $h_i(x)$. See text for details.

$$h_3(x) = \begin{cases} \frac{1}{2}x^2 - \frac{3}{2}|x| + \frac{9}{8} & \text{if } \frac{1}{2} \leqslant |x| < \frac{3}{2} \\ -x^2 + \frac{3}{4} & \text{if } |x| < \frac{1}{2} \\ 0 & \text{elsewhere} \end{cases}$$

$$h_4(x) = \begin{cases} -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3} & \text{if } 1 \leqslant |x| < 2 \\ \frac{1}{2}|x|^3 - x^2 + \frac{2}{3} & \text{if } |x| < 1 \\ 0 & \text{elsewhere} \end{cases}$$

These kernels h_k are known as *B-spline* interpolation kernels with k basis functions. As k approaches infinity, h_k approaches a Gaussian function, which is why the Gaussian can also be used as an interpolation kernel. Even though the h_k functions are frequently used interpolation kernels –especially the *cubic* B-spline kernel k_4– we can see that they are not ideal since they do not meet the above mentioned desired property for $k > 2$.

The ideal kernel does in fact exist; the *sinc* function $h(x)$ defined as

$$h(x) = \frac{\sin(\pi x)}{\pi x}$$

[6] Note that, since we are dealing with symmetrical kernels, the $*$ sign may be replaced by $\star$ everywhere in this section.

is an ideal kernel in the sense that it retains all of the spatial frequencies that are present in a digital image. Note that this kernel has the desired property[7]. In practice, we need to truncate or approximate this kernel, since we cannot work with infinitely-sized kernels. Hence, the ideal character cannot be retained in practice.

5.5 Geometric transformation

Geometric transformation is necessary when an image needs to be, *e.g.*, optimized for some viewing position, aligned with another image, or when an image needs distortion correction. Such a transformation consists of, *e.g.*, translation, rotation, scaling, skewing, elastic deformation, *etc.*

A geometric transformation is a function f that maps each image coordinate pair (x, y) to a new location (x', y'). A geometric image transformation is called *rigid*, when only translations and rotations[8] are allowed. If the transformation maps parallel lines onto parallel lines it is called *affine*. If it maps lines onto lines, it is called *projective*. Finally, if it maps lines onto curves, it is called *curved* or *elastic*. Each type of transformation contains as special cases the ones described before it, *e.g.*, the rigid transformation is a special kind of affine transformation. A composition of more than one transformation can be categorized as a single transformation of the most complex type in the composition, *e.g.*, a composition of a projective and an affine transformation is a projective transformation, and a composition of rigid transformations is again a rigid transformation. Figure 5.21 shows examples of each type of transformation.

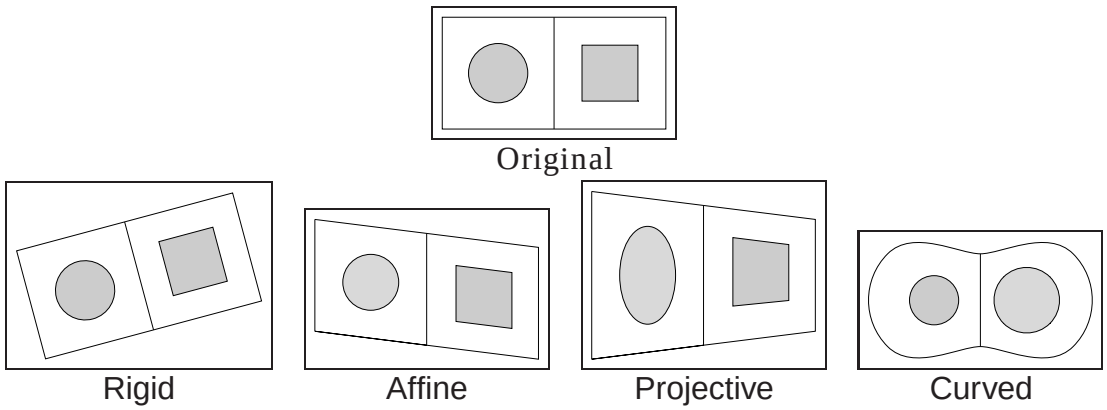


Figure 5.21 Examples of geometric transformations.

[7] Although the sinc function is singular at $x = 0$, the limit value $\lim_{x \to 0} h(x) = 1$ is commonly substituted for $h(0)$.

[8] and –technically– reflections.

Translation
Translation of an image by a vector $\begin{pmatrix} t_x \\ t_y \end{pmatrix}$ can be achieved by adding this vector to each coordinate pair:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

Rotation
Positive (anticlockwise) rotation of an image by an angle α around the origin is described by:

$$\begin{cases} x' &= x\cos\alpha &- &y\sin\alpha \\ y' &= x\sin\alpha &+ &y\cos\alpha \end{cases},$$

or in matrix notation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Rigid transformation
We can easily combine translation and rotation in one formula:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

This is commonly written as a single matrix multiplication:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & t_x \\ \sin\alpha & \cos\alpha & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine transformation
Affine transformation can be described with a similar matrix as the one used for rigid transformations:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

except now the constants a, b, c, and d can be any number.

Scaling
Scaling is a special type of affine transformation defined by

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

The parameters a and d are the scaling parameters in the x and y-direction respectively.

Skewing
Skewing is a special type of affine transformation defined by

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \tan\alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

The parameter α is called the skewing angle. Figure 5.22 shows some examples of the mentioned transformations.

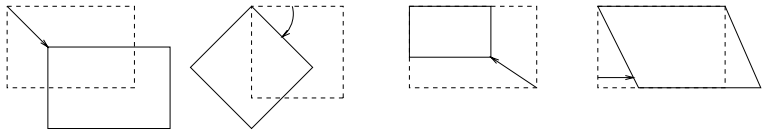


Figure 5.22 Example of (from left to right) a translation, a rotation, a scaling, and a skewing.

Example

Rotation around the image center

The rotation operation as described above does a rotation around the image *origin*. The origin is usually chosen to be at the top left corner of the image, or at the center of the top left pixel. We can rotate the image around its *center* pixel by using two additional translations to shift the origin.

Suppose we have a 256×256 pixel image, with the origin located at the top left of the image, and we wish to rotate it by 30^o around the image center. The center of the image is located at $(128, 128)$. (Note: we are working with continuous coordinates here, not integer pixel locations. The origin is at the top left of the image, so

the center of the top left pixel has coordinates $(0.5, 0.5)$.) First we translate the image so that the intended rotation center becomes the image origin, *i.e.*, we translate by a vector $\begin{pmatrix} -128 \\ -128 \end{pmatrix}$:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -128 \\ -128 \end{pmatrix}.$$

We can now do the rotation:

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} \cos 30^o & -\sin 30^o \\ \sin 30^o & \cos 30^o \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}.$$

Finally, we translate the origin back to its original location:

$$\begin{pmatrix} x''' \\ y''' \end{pmatrix} = \begin{pmatrix} x'' \\ y'' \end{pmatrix} + \begin{pmatrix} 128 \\ 128 \end{pmatrix}.$$

By substitution we can find explicit formulas for x''' and y''' directly in terms of x and y.

Projective transformation

The general transformation that describes the projection of a *3D* scene with coordinates (x, y, z) to a *2D* image with coordinates (x', y') is given by

$$\alpha \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Since this type of projective transformation occurs in conjunction with 3D images, we will mention here only the special case of the *central perspective* transformation which relates two 2D images that are different projections from the same object, with a point for a light source, as described in figure 5.23. This transformation is described by

$$\begin{cases} x' &= \frac{ax+by+c}{gx+hy+1} \\ y' &= \frac{dx+ey+f}{gx+hy+1}. \end{cases}$$

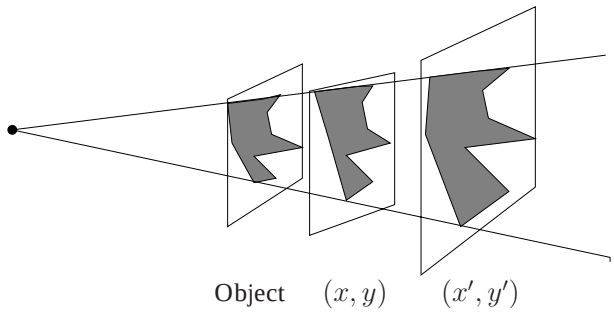


Figure 5.23 The two projection images with coordinates (x, y) and (x', y') are related by a central perspective transformation.

Intermezzo

Dental radiograph registration

Dental radiographs are X-ray projection images of teeth and surrounding structures. With many dental procedures, it is useful to make follow-up radiographs of a patient. Since the imaging circumstances cannot be exactly recreated, the images need to be registered first before an accurate comparison of the radiographs can be done. The transformation that relates the radiographs to be registered can be assumed to be a central perspective one.

To establish the correct transformation, the parameters a,b,..., h need to be found. This can be done by identifying four corresponding points in the two images. Their coordinates are measured and substituted in the central perspective equations. This gives us eight equations in the eight unknowns. After solving these equations we can apply the found transformation to the first image to bring it into registration with the second.

In practice, more than four points are usually located to increase the accuracy of the solution found.

Curved transformations

The transformation function f is unconstrained in the case of curved transformations, except that f must be a continuous function.

$$(x', y') = f(x, y),$$

or

$$\begin{cases} x' &= f_1(x, y) \\ y' &= f_2(x, y) \end{cases}$$

Since the functions f_1 and f_2 are generally unknown, polynomial approximations (usually of an order no higher than five) are often used:

$$\begin{cases} x' &= a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2 + \dots \\ y' &= b_0 + b_1 x + b_2 y + b_3 x^2 + b_4 xy + b_5 y^2 + \dots \end{cases}.$$

Note that every affine –and hence, every rigid– transformation can be captured by this formulation by choosing appropriate values for a_i and b_i with $i \in \{0, 1, 2\}$.

Intermezzo

The "barrel" and "pincushion" distortions (shown in figure 5.24) that are common to many imaging systems can in most cases be captured and corrected well using third order polynomial transformation.

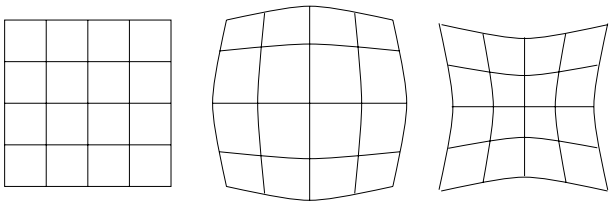


Figure 5.24 Example of barrel (middle) and pincushion (right) distortion of an image (left).

Another representation of curved transformation that is adequate for many applications is to represent the transformation as a sufficiently dense vector field, *i.e.*,

$$\begin{cases} x' &= x + t_x(x, y) \\ y' &= y + t_y(x, y) \end{cases}.$$

Where the displacement vectors $\begin{pmatrix} t_x(x, y) \\ t_y(x, y) \end{pmatrix}$ are known at a number of image locations (*e.g.*, all of the grid points) and the 'gaps' are filled in by interpolation. An example of a vector field is shown in figure 5.25.

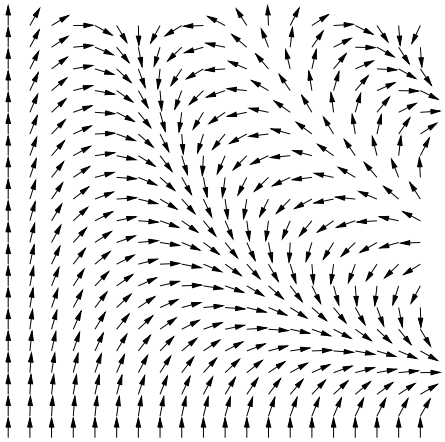


Figure 5.25 Example of a vector field representation of a curved transformation.

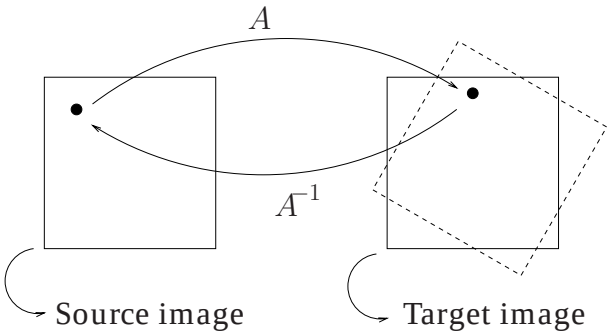


Figure 5.26 Example of backward mapping. The target image must be filled using the backward mapping A^{-1} instead of the forward mapping A.

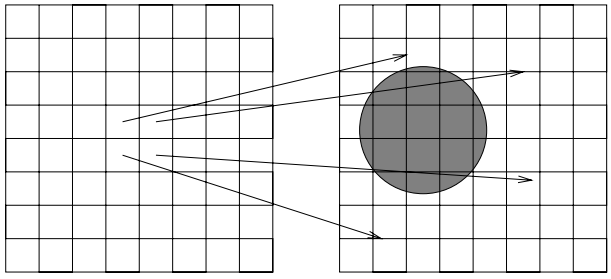


Figure 5.27 Why simple forward mapping can generally not be used for computation of a transformed image. See text for details.

5.5.1 Backward mapping

When computing the transformation of an image we often need the *inverse* (the backward mapping) of the intended transformation (the forward mapping). The reason for this is depicted in figure 5.26. Here, the forward mapping is depicted by the matrix A. By simply applying the forward mapping to all of the pixels in the source image we cannot adequately compute the grey values at all pixel locations of the target image. Figure 5.27 shows this. Here, we have drawn some example forward mappings of pixels. It will be clear that we never "reach" the pixels that are located in the circle area. These pixels should have approximately the same grey values as the ones we *do* reach in this example. By using backward mapping, this problem vanishes: by computing $A^{-1}\begin{pmatrix} x \\ y \end{pmatrix}$ for all pixel locations $\begin{pmatrix} x \\ y \end{pmatrix}$ in the *target* image we can find a corresponding location (and a grey value) in the source image. If we do not exactly end up in a pixel location of the source image, we can approximate an appropriate grey value by interpolation.

In the case of rigid transformations we can establish A^{-1} by matrix inversion. In the affine case too, but we must first check if the matrix is invertible, which is the case as long as $ad - bc \neq 0$. In the case of curved transformation finding an inverse (if it exists at all) often proves a very difficult task, and often the best way to compute a transformed image is to use forward mapping with computer graphics techniques.