Convolution, Smoothing, and Image Derivatives

Carlo Tomasi

Computer vision operates on images that usually come in the form of arrays of pixel values. These values are invariably affected by noise, so it is useful to clean the images somewhat by an operation, called *smoothing*, that replaces each pixel by a linear combination of some of its neighbors. Smoothing reduces the effects of noise, but blurs the image. When noise suppression is the goal, blurring is an undesired side-effect. In other applications, when it is desired to emphasize slow spatial variations over abrupt changes, blurring is beneficial. In yet another set of circumstances, these abrupt changes are themselves of interest, and then one would like to apply an operator that is in some sense complementary to smoothing (in signal processing, this operator would be called a high-pass filter).

All these operations take the form of what is called a *convolution*. The closely related but more immediately intuitive notion of *correlation* is introduced first in this note.

While an image is an array of pixel values, it is often useful to regard it as a sampling of an underlying continuous function of spatial coordinates. This function is the brightness of light impinging onto the camera sensor, before this brightness is measured and sampled by the individual sensor elements. *Partial derivatives* of this continuous function can be used to measure the extent and direction of edges, that is, abrupt changes of image brightness that occur along curves in the image plane. The estimation of these derivatives can again be cast as a convolution. The next section uses a naive version of differentiation to motivate convolution. The last section of this note shows how derivatives are estimated more accurately.

What follows refers to a coordinate frame with its origin at the upper-left image pixel, which has coordinates $(1, 1)$ (to conform with matrix notation). The vertical axis is called x and points downward, and the horizontal axis is called y and points to the right. This convention makes (x, y) pairs consistent with the usual indexing of rows and columns as (i, j), as done in linear algebra, so increasing x corresponds to increasing i, and increasing y corresponds to increasing j.

1 Correlation

Suppose that we want to determine where in the image there are vertical edges. Since an edge is an abrupt change of image intensity, we might start by computing the derivatives of an image in the horizontal direction. Pixels where the derivatives have a large magnitude, either positive or negative, are elements of vertical edges. The partial derivative of a continuous function $f(x, y)$ with respect to the "horizontal" variable y is defined as the local slope of the plot of the function along the y direction or, formally, by the following limit:

$$\frac{\partial f(x,y)}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y} \ .$$

An image from a digitizer is a function of a discrete variable, so we cannot make Δy arbitrarily small: the smallest we can go is one pixel. If our unit of measure is the pixel, we have

$$\Delta y = 1$$

and a rather crude approximation to the derivative at an integer position $x = i$, $y = j$ is therefore

$$\left.\frac{\partial f(x,y)}{\partial y}\right|_{x=i,y=j} \approx f(i,j+1) - f(i,j) \; .$$

We will see much better ways to estimate image derivatives, but this example is good enough for introducing correlation and convolution.

Here is a piece of pseudo-code that computes this approximation along row i in the image:

for $j = j_{\text{start}}, \ldots, j_{\text{finish}}$ **do**
  $d(i,j) \leftarrow f(i,j+1) - f(i,j)$
end for

Notice, in passing, that the last value of j for which this computation is defined is the next-to-last pixel in the row, so j_{finish} must be defined appropriately.

The computation above amounts to taking a little two-cell *mask* (or *template* or *kernel*) k with the values $k(0) = -1$ and $k(1) = 1$ in its two entries, placing the mask in turn at every position j along row i, multiplying what is under the mask by the mask entries, and adding the result. In pseudo-code,

for $j = j_{\text{start}}, \ldots, j_{\text{finish}}$ **do**
  $d(i,j) \leftarrow k(0)f(i,j) + k(1)f(i,j+1,)$
end for

This adds a little generality, because we can change the values of k without changing the code. Since we are generalizing, we might as well allow for several entries in k. For instance, we might in the future switch to a central approximation to the derivative,

$$\left.\frac{\partial f(x,y)}{\partial y}\right|_{x=i,y=j} \approx \frac{f(i,j+1) - f(i,j-1)}{2} \; .$$

So now we can define for instance $k[-1] = -1/2$, $k[0] = 0$, and $k[1] = 1/2$ and write a general-purpose loop in view of possible future changes in our choice of k:

for $j = j_{\text{start}}, \ldots, j_{\text{finish}}$ **do**
  $d(i,j) \leftarrow 0$
  for $v = v_{\text{start}}, \ldots, v_{\text{finish}}$ **do**
    $d(i,j) \leftarrow d(i,j) + k(v)f(i,j+v)$
  end for
end for

This is now much more general: it lets us choose which horizontal neighbors to combine and with what weights. But clearly we will soon want to also combine pixels above i,j, not only on its sides, and for the whole picture, not just one row. This is easily done:

for $i = i_{\text{start}}, \ldots, i_{\text{finish}}$ **do**
  for $j = j_{\text{start}}, \ldots, j_{\text{finish}}$ **do**
    $d(i, j) \leftarrow 0$
    for $u = u_{\text{start}}, \ldots, u_{\text{finish}}$ **do**
      for $v = v_{\text{start}}, \ldots, v_{\text{finish}}$ **do**
        $d(i, j) \leftarrow d(i, j) + k(u, v) f(i + u, j + v)$
      end for
    end for
  end for
end for

where now $k(u, v)$ is a two-dimensional array. The two innermost `for` loops just keep adding values to $d(i, j)$,so we can express that piece of code by the following mathematical expression:

$$d(i,j) = \sum_{u=u_{\text{start}}}^{u_{\text{finish}}} \sum_{v=v_{\text{start}}}^{v_{\text{finish}}} k(u,v) f(i+u, j+v) \ . \tag{1}$$

This is called a *correlation*. A very closely related operation is *convolution*:

$$h(i,j) = \sum_{a=a_{\text{start}}}^{a_{\text{finish}}} \sum_{b=b_{\text{start}}}^{b_{\text{finish}}} g(a,b) f(i-a, j-b) \tag{2}$$

where the only difference is in the two minus signs. From a programming point of view, there is little difference between convolution and correlation:

$$d(i,j) = \sum_{u=u_{\text{start}}}^{u_{\text{finish}}} \sum_{v=v_{\text{start}}}^{v_{\text{finish}}} k(u,v) f(i+u, j+v) = \sum_{a=a_{\text{start}}}^{a_{\text{finish}}} \sum_{b=b_{\text{start}}}^{b_{\text{finish}}} g(a,b) f(i-a, j-b)$$

where

$$u = -a \ , \quad v = -b \ , \quad a_{\text{start}} = -u_{\text{finish}} \ , \quad a_{\text{finish}} = -u_{\text{start}} \ , \quad b_{\text{start}} = -v_{\text{finish}} \ , \quad b_{\text{finish}} = -v_{\text{start}}$$

and

$$g(a,b) = k(-a, -b) \ .$$

The last equality makes convolution less natural to think about than correlation, because of the need to "mirror flip" the straightforward kernel k into $g(a, b)$ when replacing correlation with convolution: to correlate, first mirror-flip the kernel and then convolve. For instance, to take the derivative in the horizontal direction we used kernel

$$k(0) = -1 \quad \text{and} \quad k(1) = 1 \ .$$

The coefficient $k(1)$ multiplies the pixel value to the *right* of the current position (i, j), so the index 1 is a natural choice. If we express finite-difference differentiation with a convolution, we need to use a kernel g with

$$g(0) = 1 \quad \text{and} \quad g(1) = -1$$

and it is somewhat awkward that $g(1)$ multiplies the pixel value to the *left* of the current position (i, j).

Mathematically, however, convolution enjoys important properties that would be expressed in a more complicated way with correlation. This is because convolution looks like the familiar multiplication rule for polynomials in terms of their coefficients. To see this, consider two polynomials

$$\begin{aligned} f(z) &= f_0 + f_1 z + \ldots + f_m z^m \\ g(z) &= g_0 + g_1 z + \ldots + g_n z^n \; . \end{aligned}$$

Then, the sequence of coefficients of the product

$$h(z) = h_0 + h_1 z + \ldots + h_{m+n} z^{m+n}$$

of these polynomials is the (one-variable) convolution of the sequences of their coefficients:

$$h_i = \sum_{a=a_{\text{start}}}^{a_{\text{finish}}} g_a f_{i-a} \tag{3}$$

for appropriate values of a_{start} and a_{finish}. This is because g_a multiplies z^a and f_{i-a} multiplies z^{i-a}, so the power corresponding to $g_a f_{i-a}$ is z^i for all values of a, and h_i as defined by equation (3) is the sum of all the products with a term z^i, as required by the definition of product between two polynomials.[1] This connection with polynomial multiplication makes convolution an even deeper are more pervasive concept in mathematics than image processing would justify.

Convolving a signal with a given mask g is also called *filtering* that signal with that mask. When referred to image filtering, the mask is also called the *point-spread function* of the filter. If we let

$$f(i,j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j = 0 \\ 0 & \text{otherwise} \end{cases} \quad , \tag{4}$$

then the image f is a single point (the 1) in a sea of zeros. When the convolution (2) is computed, we obtain

$$h(i,j) = g(i,j) \; .$$

In words, the single point at the origin is spread into a blob equal to the mask (interpreted as an image).

The concept of convolution can be extended to continuous functions as well. In analogy with equation (2), we define the convolution between two continuous functions $f(x,y)$ and $g(x,y)$ as the following double integral:

$$h(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} g(a,b) f(x-a, y-b)\, da\, db \; .$$

2 Smoothing

The effects of noise on images can be reduced by smoothing, that is, by replacing every pixel by a weighted average of its neighbors. This operation can be expressed by the following convolution:

$$h(i,j) = \sum_{a=a_{\text{start}}}^{a_{\text{finish}}} \sum_{b=b_{\text{start}}}^{b_{\text{finish}}} g(a,b) f(i-a, j-b) \tag{5}$$

[1] Verify this fact with an example.

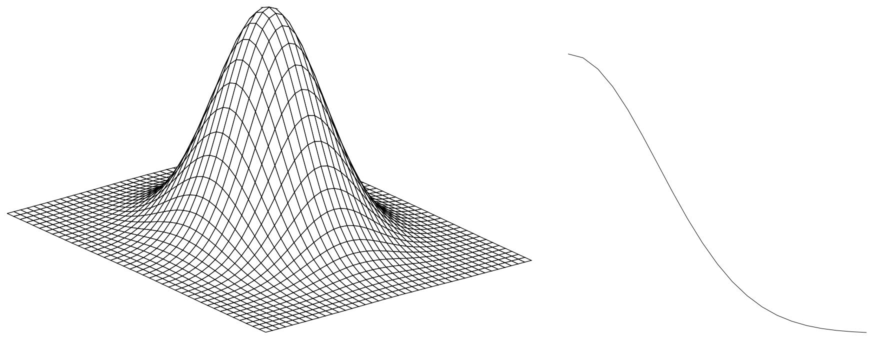

Figure 1: The two dimensional kernel on the left can be obtained by rotating the function $\gamma(r)$ on the right around a vertical axis through the maximum of the curve ($r = 0$).

where g is the convolution mask that lists the weights, f is the image, and $a_{\text{start}}, a_{\text{finish}}, b_{\text{start}}, b_{\text{finish}}$ delimit the domain of definition of the kernel, that is, the size of the neighborhood involved in smoothing. The kernel is usually rotationally symmetric, as there is no reason to privilege, say, the pixels on the left of position i, j over those on the right[2]:

$$\begin{aligned} -a_{\text{start}} = a_{\text{finish}} &= -b_{\text{start}} = b_{\text{finish}} = n \\ g(a,b) &= \gamma(r) \end{aligned} \tag{6}$$

where

$$r = \sqrt{a^2 + b^2}$$

is the distance from the center of the kernel to its element a, b. Thus, a rotationally symmetric kernel can be obtained by rotating a one-dimensional function $\gamma(r)$ defined on the nonnegative reals around the origin of the plane (figure 1).

2.1 The Gaussian Function

The plot in figure 1 was obtained from the Gaussian function

$$\gamma(r) = \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2}\left(\frac{r}{\sigma}\right)^2}$$

with $\sigma = 6$ pixels (one pixel corresponds to one cell of the mesh in figure 1), so that

$$g(a,b) = \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2}\frac{a^2+b^2}{\sigma^2}} \ . \tag{7}$$

The normalizing factor $1/(2\pi\sigma^2)$ makes the integral of the two-dimensional Gaussian equal to one. This normalization, however, assumes that a, b in $g(a, b)$ are real variables, and that the Gaussian is defined over the entire plane.

In the following, we first justify the choice of the Gaussian, by far the most popular smoothing function in computer vision, and then give a better normalization factor for a discrete and truncated version of it.

[2]This only holds for smoothing. Nonsymmetric filters *tuned* to particular orientations are very important in vision. Even for smoothing, some authors have proposed to bias filtering along an edge away from the edge itself. An idea worth pursuing.

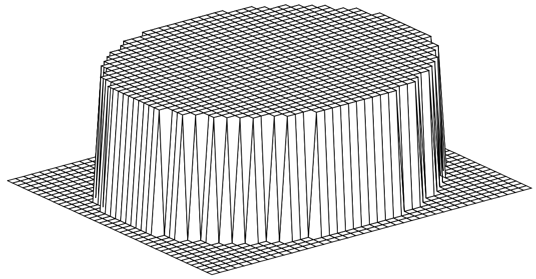

Figure 2: The pillbox function.

The Gaussian function satisfies an amazing number of mathematical properties, and describes a vast variety of physical and probabilistic phenomena. Here we only look at properties that are immediately relevant to computer vision.

The first set of properties is qualitative. The Gaussian is, as noted above, symmetric. It also emphasizes nearby pixels over more distant ones, a property shared by any nonincreasing function $\gamma(r)$. This property reduces smearing (blurring) while still maintaining noise averaging properties. In fact, compare a Gaussian with a given support to a pillbox function over the same support (figure 2) and having the same volume under its graph. Both kernels reach equally far around a given pixel when they retrieve values to average together. However, the pillbox uses all values with equal emphasis. Figure 3 shows the effects of convolving a step function with either a Gaussian or a pillbox function. The Gaussian produces a curved ramp at the step location, while the pillbox produces a flat ramp. However, the pillbox ramp is wider than the Gaussian ramp, thereby producing a sharper image.

Another useful property of the Gaussian function is its smoothness. If $g(a, b)$ is considered as a function of real variables a, b, it is differentiable infinitely many times. Although this property by itself is not too useful with discrete images, it implies that in the frequency domain the Gaussian drops as fast as possible among all functions of a given space-domain support. Thus, it is as low-pass a filter as one can get for a given spatial support. This holds approximately also for the discrete and truncated version of the Gaussian. In addition, the Fourier transform of a Gaussian is again a Gaussian, a mathematically convenient fact. Specifically,

$$\mathcal{F}\left[e^{-\pi(x^2+y^2)}\right] = e^{-\pi(u^2+v^2)} \; .$$

In words, the Gaussian function $e^{-\pi(x^2+y^2)}$ is an eigenfunction of the Fourier transformation.[3] The Fourier transform of the normalized and scaled Gaussian $g(a, b)$ defined in equation (7) is

$$G(u, v) = e^{-\frac{1}{2}(2\pi\sigma)^2(u^2+v^2)} \; .$$

Another important property of $g(a, b)$ is that it never crosses zero, since it is always positive. This is essential for instance for certain types of edge detectors, for which smoothing cannot be allowed to introduce its own zero crossings in the image.

The Gaussian function is also a separable function. A function $g(a, b)$ is said to be *separable* if there are two functions g_1 and g_2 of one variable such that

$$g(a, b) = g_1(a) g_2(b) \; .$$

[3] A function f is an eigenfunction for a transformation T if $Tf = \lambda f$ for some scalar λ.

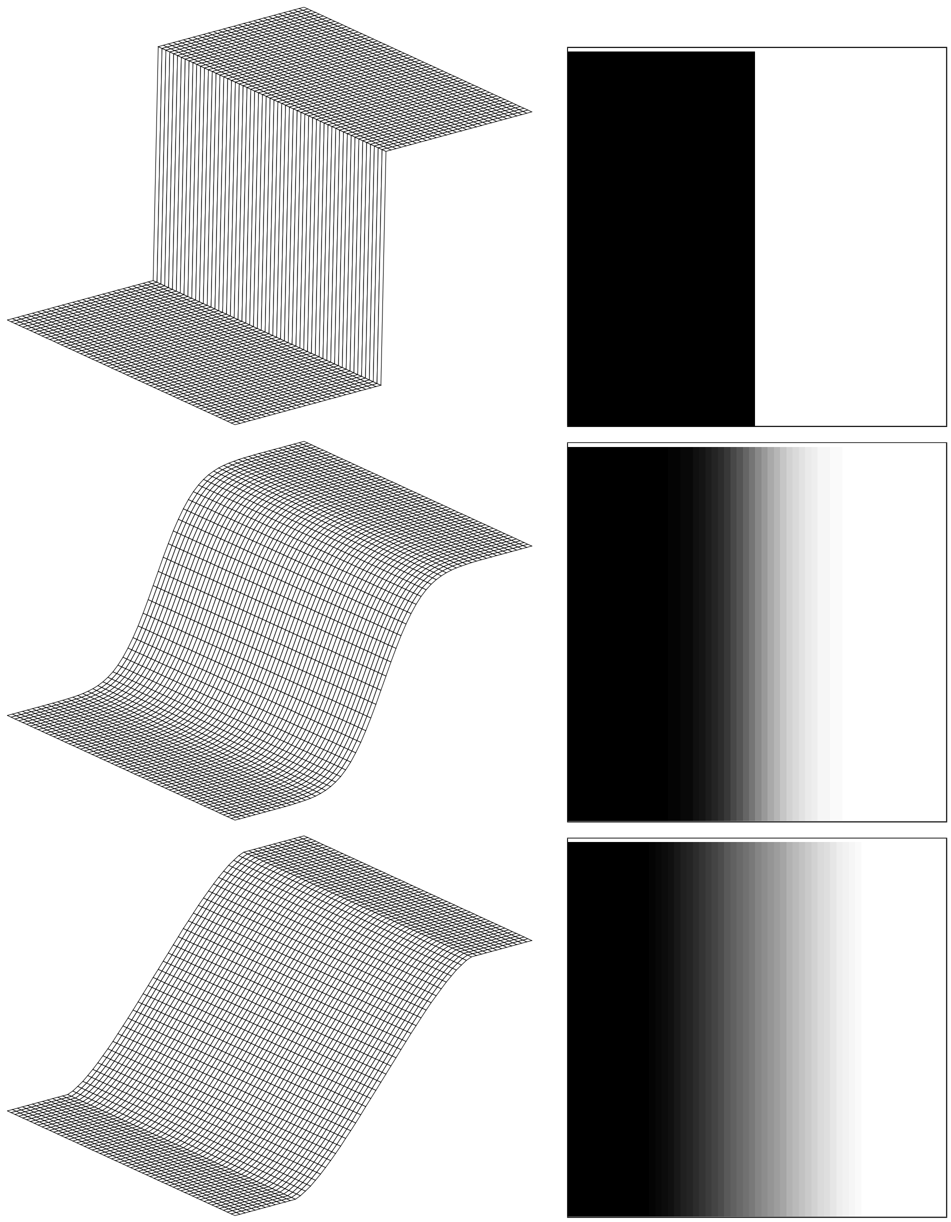

Figure 3: Intensity graphs (left) and images (right) of a vertical step function (top), and of the same step function smoothed with a Gaussian (middle), and with a pillbox function (bottom). Gaussian and pillbox have the same support and the same integral.

For the Gaussian, this is a consequence of the fact that

$$e^{x+y} = e^x e^y$$

which leads to the equality

$$g(a,b) = g_1(a)g_1(b)$$

where

$$g_1(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \tag{8}$$

is the one-dimensional Gaussian, whose integral is also 1.

Thus, the Gaussian of equation (7) separates into two equal factors. This is computationally important, because the convolution (5) can then itself be separated into two one-dimensional convolutions:

$$h(i,j) = \sum_{a=-n}^{n} g_1(a) \sum_{b=-n}^{n} g_1(b) f(i-a, j-b) \tag{9}$$

(we also used equation (6) for simplicity), with substantial savings in the computation. In fact, the double summation

$$h(i,j) = \sum_{a=-n}^{n} \sum_{b=-n}^{n} g(a,b) f(i-a, j-b)$$

requires m^2 multiplications and $m^2 - 1$ additions, where $m = 2n + 1$ is the number of pixels in one row or column of the convolution mask $g(a,b)$. The sums in (9), on the other hand, can be rewritten so as to be computed by $2m$ multiplications and $2(m-1)$ additions as follows:

$$h(i,j) = \sum_{a=-n}^{n} g_1(a)\, \phi(i-a, j) \tag{10}$$

where

$$\phi(i,j) = \sum_{b=-n}^{n} g_1(b) f(i, j-b)\ . \tag{11}$$

Both these expressions are convolutions, with an $m \times 1$ and a $1 \times m$ kernel, respectively, so they each require m multiplications and $m - 1$ additions.

Of course, to actually achieve this gain, convolution must now be performed in the two steps (11) and (10): first convolve the entire image with g_1 in the horizontal direction, then convolve the resulting image with g_1 in the vertical direction (or in the opposite order, since convolution commutes). If we were to perform (9) literally, there would be no gain, as for each value of $i-a$, the internal summation is recomputed m times, since any fixed value $d = i - a$ occurs for pairs $(i, a) = (d - n, -n), (d - n + 1, -n + 1), \ldots, (d + n, n)$ when equation (9) is computed for every pixel (i, j).

Thus, separability decreases the operation to $2m$ multiplications and $2(m - 1)$ additions, with an approximate gain

$$\frac{2m^2 - 1}{4m - 2} \approx \frac{2m^2}{4m} = \frac{m}{2}\ .$$

If for instance $m = 21$, we need only 42 multiplications instead of 441, with an approximately tenfold increase in speed.

Exercise. Notice the similarity between $\gamma(r)$ and $g_1(a)$. Is this a coincidence?

2.2 Normalization and Truncation

All Gaussian functions in this section were given with normalization factors that make the integral of the kernel equal to one, either on the plane or on the line. This normalization factor must be taken into account when actual values output by filters are important. For instance, if we want to smooth an image, initially stored in a file of bytes, one byte per pixel, and write the result to another file with the same format, the values in the smoothed image should be in the same range as those of the unsmoothed image. Also, when we compute image derivatives, it is sometimes important to know the actual value of the derivatives, not just a scaled version of them.

However, using the normalization values as given above would not lead to the correct results, and this is for two reasons. First, we do not want the *integral* of $g(a, b)$ to be normalized, but rather its sum over an integer grid. Second, our grids are invariably finite, so we want to add up only the values we actually use, as opposed to every value for a, b between $-\infty$ and $+\infty$.

The solution to this problem is simple. For a smoothing filter we first compute the unscaled version of, say, the Gaussian in equation (7), and then normalize it by sum of the samples:

$$\begin{aligned} g_0(a,b) &= e^{-\frac{1}{2}\frac{a^2+b^2}{\sigma^2}} \\ c &= \sum_{a=-n}^{n}\sum_{b=-n}^{n} g_0(a,b) \\ g(a,b) &= \frac{1}{c} g_0(a,b) \ . \end{aligned} \tag{12}$$

To verify that this yields the desired normalization, consider an image with constant intensity f_0. Then its convolution with the new $g(a, b)$ should yield f_0 everywhere as a result. In fact, we have

$$\begin{aligned} h(i,j) &= \sum_{a=-n}^{n}\sum_{b=-n}^{n} g(a,b) f(i-a, j-b) \\ &= f_0 \sum_{a=-n}^{n}\sum_{b=-n}^{n} g(a,b) \\ &= f_0 \end{aligned}$$

as desired. Of course, normalization can be performed on one-dimensional Gaussian functions separably, if the two-dimensional Gaussian function is written as the product of two one-dimensional Gaussian functions. The concept is the same:

$$\begin{aligned} g_{10}(b) &= e^{-\frac{1}{2}\left(\frac{b}{\sigma}\right)^2} \\ c &= \sum_{b=-n}^{n} g_1(b) \\ g_1(b) &= \frac{1}{c} g_{10}(b) \ . \end{aligned} \tag{13}$$

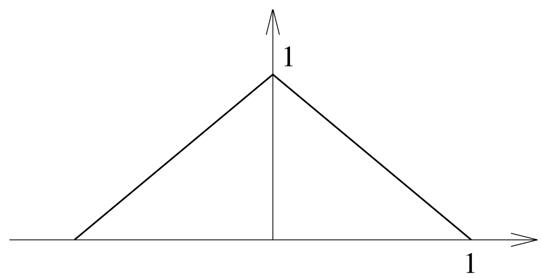

Figure 4: The triangle function interpolates linearly.

3 Image Differentiation

In order to compute derivatives of discrete images, one needs a model for how the underlying continuous[4] image behaves between pixel values. For instance, approximating the derivative with a first-order difference

$$f(i,j+1)-f(i,j)$$

implies that the underlying image is piecewise linear, because then the first-order difference is exactly the derivative of a linear function that goes through $f(i,j+1)$ and $f(i,j)$.

More generally, if the discrete image is formed by samples of the continuous image, then the latter interpolates the former. Interpolation can be expressed as a hybrid-domain convolution:

$$h(x,y)=\sum_{a=-n}^{n}\sum_{b=-n}^{n} f(a,b)p(x-a,y-b)$$

where x, y are real variables and $p(x,y)$, the *interpolation function*, must satisfy the constraint

$$p(a,b)=\begin{cases} 1 & \text{if } a=b=0 \\ 0 & \text{for all other integers } a,b \end{cases} .$$

This constraint ensures that

$$h(i,j)=f(i,j)$$

on all integer grid points, that is, that p actually interpolates the image points $f(i,j)$. For instance, for linear interpolation in one dimension, p is the triangle function of figure 4.

Since both interpolation and differentiation are linear, instead of interpolating the image and then differentiating we can interpolate the image with the derivative of the interpolation function. Formally,

$$\begin{aligned} h_x(x,y) &= \frac{\partial h}{\partial x}(x,y)=\frac{\partial}{\partial x}\sum_{a=-n}^{n}\sum_{b=-n}^{n} f(a,b)p(x-a,y-b) \\ &= \sum_{a=-n}^{n}\sum_{b=-n}^{n} f(a,b)p_x(x-a,y-b) . \end{aligned}$$

We then need to sample the result at the grid points i, j to obtain a discrete image. This yields the final, discrete convolution that computes the derivative of the underlying continuous image h with respect to the

[4]Continuity here refers to continuity of the domain: a and b are real numbers.

vertical variable:

$$h_x(i,j) = \sum_{a=-n}^{n} \sum_{b=-n}^{n} f(a,b) p_x(i-a, j-b) \ . \tag{14}$$

From Shannon's sampling theorem, we know that the mathematically correct interpolation function to use would be the sinc function:

$$p(x,y) = \text{sinc}(x,y) = \frac{\sin \pi x}{\pi x} \frac{\sin \pi y}{\pi y} \ .$$

However, the sinc decays proportionally to $1/x$ and $1/y$, which is a rather slow rate of decay. Consequently, only values that are far away from the origin can be ignored in the computation. In other words, the summation limit n in (14) must be large, which is a computationally undesirable state of affairs. In addition, if there is aliasing[5], the sinc function will amplify its effects, as it combines a large number of unrelated pixel values.

Although the optimal solution to this dilemma is outside the scope of this course, it is clear that a good interpolation function p must pass only frequencies below a certain value in order to smooth the image and reduce noise. At the same time, it should also have a small support in the spatial domain. We noted in the previous section that the Gaussian function fits this bill, since it is compact in both the space and the frequency domain. We therefore let p_0 be the (unnormalized) Gaussian function,

$$p_0(x,y) = g_0(x,y)$$

and p_{0x}, p_{0y} its partial derivatives with respect to x and y (figure 5). We then sample p_{0x} and p_{0y} over the integers and normalize them by requiring that their response to a ramp yield the slope of the ramp itself. A unit-slope, discrete ramp in the i direction is represented by

$$u(i,j) = i$$

and we want to find a constant c such that

$$c \sum_{a=-n}^{n} \sum_{b=-n}^{n} u(a,b) p_{0x}(i-a, j-b) = 1 \ .$$

for all i, j so that

$$p_x(x,y) = c\, p_{0x}(x,y) \quad \text{and} \quad p_y(x,y) = c\, p_{0y}(x,y) \ .$$

In particular for $i = j = 0$ we obtain

$$c = -\frac{1}{\sum_{a=-n}^{n} \sum_{b=-n}^{n} b g_{0x}(a,b)} \ . \tag{15}$$

Since the partial derivative $g_{0x}(a,b)$ of the Gaussian function with respect to a is negative for positive a, this constant c is positive. By symmetry, the same constant normalizes g_{0y}.

Of course, since the two-dimensional Gaussian function is separable, so are its two partial derivatives:

$$h_x(i,j) = \sum_{a=-n}^{n} \sum_{b=-n}^{n} f(a,b) g_x(i-a, j-b) = \sum_{a=-n}^{n} d_1(i-a) \sum_{b=-n}^{n} f(a,b) g_1(j-b)$$

[5] Aliasing means that the image is sampled too coarsely to properly represent all its spatial frequencies.

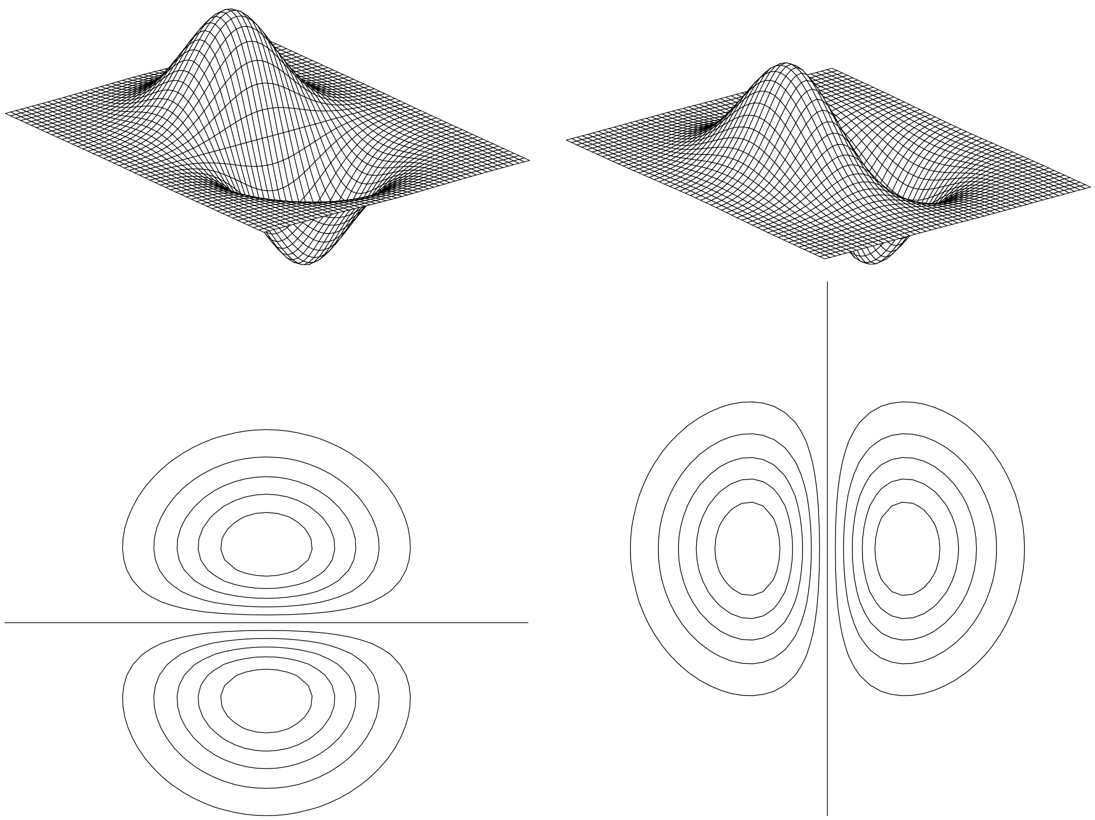

Figure 5: The partial derivatives of a Gaussian function with respect to x (left) and y (right) represented by plots (top) and isocontours (bottom). In the isocontour plots, the x variable points vertically down and the y variable points horizontally to the right.

where

$$d_1(x) = \frac{dg_1}{dx} = -\frac{x}{\sigma^2} g_1(x)$$

is the ordinary derivative of the one-dimensional Gaussian function $g_1(x)$ defined in (8). A similar expression holds for $h_y(i,j)$ (see below).

Thus, the partial derivative of an image in the x direction is computed by convolving with $d_1(x)$ and $g_1(y)$. The partial derivative in the y direction is obtained by convolving with $d_1(y)$ and $g_1(x)$. In both cases, the order in which the two one-dimensional convolutions are performed is immaterial:

$$\begin{aligned} h_x(i,j) &= \sum_{a=-n}^{n} d_1(i-a) \sum_{b=-n}^{n} f(a,b) g_1(j-b) = \sum_{b=-n}^{n} g_1(j-b) \sum_{a=-n}^{n} f(a,b) d_1(i-a) \\ h_y(i,j) &= \sum_{a=-n}^{n} g_1(i-a) \sum_{b=-n}^{n} f(a,b) d_1(j-b) = \sum_{b=-n}^{n} d_1(j-b) \sum_{a=-n}^{n} f(a,b) g_1(i-a) \; . \end{aligned}$$

Normalization can also be done separately: the one-dimensional Gaussian g_1 is normalized according to (13), and the one-dimensional Gaussian derivative $d_1(a)$ is normalized by the one-dimensional equivalent of (15):

$$\begin{aligned} d_0(x) &= -x e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \\ c &= \frac{1}{\sum_{b=-n}^{n} b d_0(b)} \\ d_1(x) &= \frac{1}{c} d_0(x) \; . \end{aligned}$$