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. In the case of noise suppression, blurring is an undesired 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). Fortunately, all these operations take the form of what is called a *convolution*. This note introduces the concept of convolution in a simplistic but useful way. Smoothing is subsequently treated as an important special case.

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. Derivatives, or rather their estimates, can again be cast as convolution operators. 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.

1 Convolution

To introduce the concept of convolution, 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. Derivatives with 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 x is defined as the local slope of the plot of the function along the x direction or, formally, by the following limit:

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

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

$$\Delta x = 1$$

and a rather crude approximation to the derivative at an integer position $j = x$, $i = y$ is therefore[1]

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

Here we assume for simplicity that the origins and axis orientations of the x, y reference system and the i, j system coincide. When we do edge detection, we will see that we can do much better than this as an approximation to the derivative, but this example is good enough for introducing convolution.

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

```
for (j = jstart; j <= jend; j++) h[i][j]= f[i][j+1] - f[i][j];
```

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 `jend` must be defined appropriately. This operation amounts to taking a little two-cell mask g with the values $g[0] = 1$ and $g[1] = -1$ 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 C, we have

```
for (j = jstart; j <= jend; j++) h[i][j]= g[0]*f[i][j+1] + g[1]*f[i][j];
```

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

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

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

```
for (j = jstart; j <= jend; j++)
{
  h[i][j] = 0;
  for (b = bstart; b <= bend; b++)
    h[i][j] += g[b]*f[i][j-b];
}
```

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 = istart; i <= iend; i++)
for (j = jstart; j <= jend; j++)
{
  h[i][j] = 0;
  for (a = astart; a <= aend; a++)
  for (b = bstart; b <= bend; b++)
    h[i][j] += g[a][b]*f[i-a][j-b];
}
```

[1]Notice that to conform with usual notation the order of variables i, j in the discrete array is switched with respect to that of the corresponding variables x, y in the continuous function: x and j are right, and y and i are down, respectively. Other conventions are possible, of course. For instance, Forsyth and Ponce have the y axis pointing up.

where now `g[a][b]` is a two-dimensional array. The part within the braces is a very important operation in signal processing. The two innermost `for` loops just keep adding values to `h[i][j]`,so we can express that piece of code by the following mathematical expression:

$$h(i,j) = \sum_{a=a_{start}}^{a_{end}} \sum_{b=b_{start}}^{b_{end}} g(a,b) f(i-a, j-b) \ . \tag{1}$$

This is called a *convolution*. 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. In fact, if we let

$$f(i,j) = \delta(i,j) = \begin{cases} 1 & \text{if } i = j = 0 \\ 0 & \text{otherwise} \end{cases} \ , \tag{2}$$

then the image f is a single point (the 1) in a sea of zeros. When the convolution (1) 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 choice of subscripts for the entries of g, in both the code and the mathematical expression, seems arbitrary at first. In fact, instead of defining $g[-1] = 1$, $g[0] = 0$, $g[1] = -1$, we could have written, perhaps more naturally, $g[-1] = -1$, $g[0] = 0$, $g[1] = 1$, and in the expressions `f[i-a][j-b]` and $f(i-a, j-b)$ the minus signs would be replaced by plus signs. In terms of programming, there is no difference between these two options (and others as well). Mathematically, on the other hand, the minus sign is much preferable. The first reason is that $g(i,j)$ can be interpreted, as done above, as a point spread function. With the other choice of signs the convolution of $f = \delta$ with g would yield a doubly-mirrored image $g(-i,-j)$ of the mask g.

Another reason for this choice of signs is that the convolution now looks like the familiar multiplication for polynomials. In fact, 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:

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

In fact, notice that 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. Verify this with an example. Thus, putting a minus sign in the definition (1) of the convolution makes the latter coincide with the product of two polynomials, thereby making the convolution an even deeper are more pervasive concept in mathematics.

The interpretation of the convolution mask $g(i,j)$ as a point-spread function suggests another useful way to look at the operation of filtering. The δ function defined in (2) is a single spike of unit height at the

origin. A generic image $f(i,j)$, on the other hand, can be seen as a whole collection of spikes, one per pixel, whose height equals the image value. In formulas,

$$f(i,j) = \sum_a \sum_b f(a,b)\delta(i-a, j-b) \ ,$$

where the summations range over the entire image. This expression is the convolution of f and δ. Notice that this is the same as

$$f(i,j) = \sum_a \sum_b f(i-a, j-b)\delta(a,b)$$

after the change of variables $i \to i-a$, $j \to j-b$ at least if the summation ranges are assumed to be $(-\infty, +\infty)$[2]. But if the output to $\delta(i,j)$ is the point-spread function $g(i,j)$, then the output to $\sum_a \sum_b f(a,b)\delta(i-a, j-b)$ is a linear combination of point-spread functions, amplified each by one of the pixels in the image. This describes, for instance, what happens in a pinhole camera with a pinhole of nonzero radius. In fact, one point in the world spreads into a small disk on the image plane (the point-spread function, literally). Each point in the world draws a little disk onto the image, and the brightness of each disk is proportional to the brightness of the point in the world. This results in a blurred image. In conclusion, the image formed by a pinhole camera is the convolution of the ideal (sharp) image with a pillow-case function.

The difference between the convolution defined in (1) and what happens in the pinhole camera is that the points in the world are not neatly arranged onto a rectangular grid, as are pixels in an image, but form a continuous. Fortunately, all the concepts relative to convolution can be extended to continuous functions as well. In analogy with equation (1), we define the convolution between two continuous functions $f(x,y)$ and $g(x,y)$ as the following double integral:

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

The blurred image produced by the pinhole camera is then the convolution of the ideally sharp image $f(x,y)$ with the pillow-case function

$$g(x,y) = \begin{cases} 1 & \text{if } \sqrt{x^2+y^2} \le r \\ 0 & \text{otherwise} \end{cases} \ ,$$

where r is the radius of the pinhole.

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:

$$h(i,j) = \sum_{a=a_{start}}^{a_{end}} \sum_{b=b_{start}}^{b_{end}} g(a,b) f(i-a, j-b) \tag{4}$$

where g is the convolution mask (or kernel or point-spread function) that lists the weights, f is the image, and a_{start}, a_{end}, b_{start}, b_{end} delimit the domain of definition of the kernel, that is, the size of the neighborhood

[2] Otherwise, they should be modified according to the change of variables.

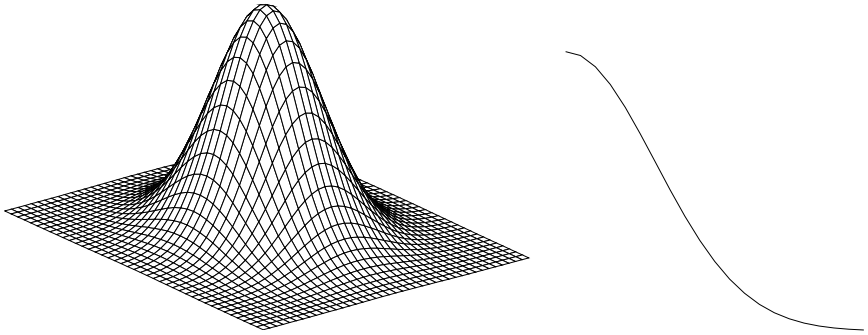

Figure 1: The two dimensional kernel on the left can be obtained by rotating the function $\gamma(r)$ on the right around a vertical axis through the maximum of the curve ($r = 0$).

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[3]:

$$\begin{aligned} -a_{start} = a_{end} &= -b_{start} = b_{end} = n \\ g(a,b) &= \gamma(r) \end{aligned} \tag{5}$$

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

$$g(a,b) = \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2}\frac{a^2+b^2}{\sigma^2}} \ . \tag{6}$$

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.

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.

[3]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.

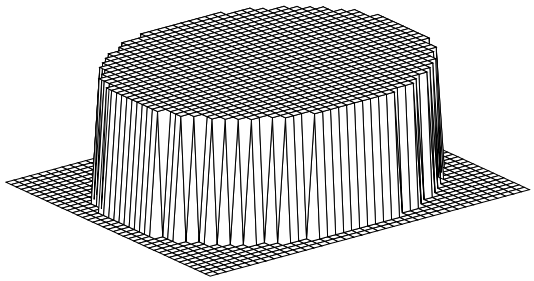

Figure 2: The pillbox function.

The 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.

A more quantitative 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.[4] The Fourier transform of the normalized and scaled Gaussian $g(a, b)$ defined in equation (6) 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

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

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

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

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

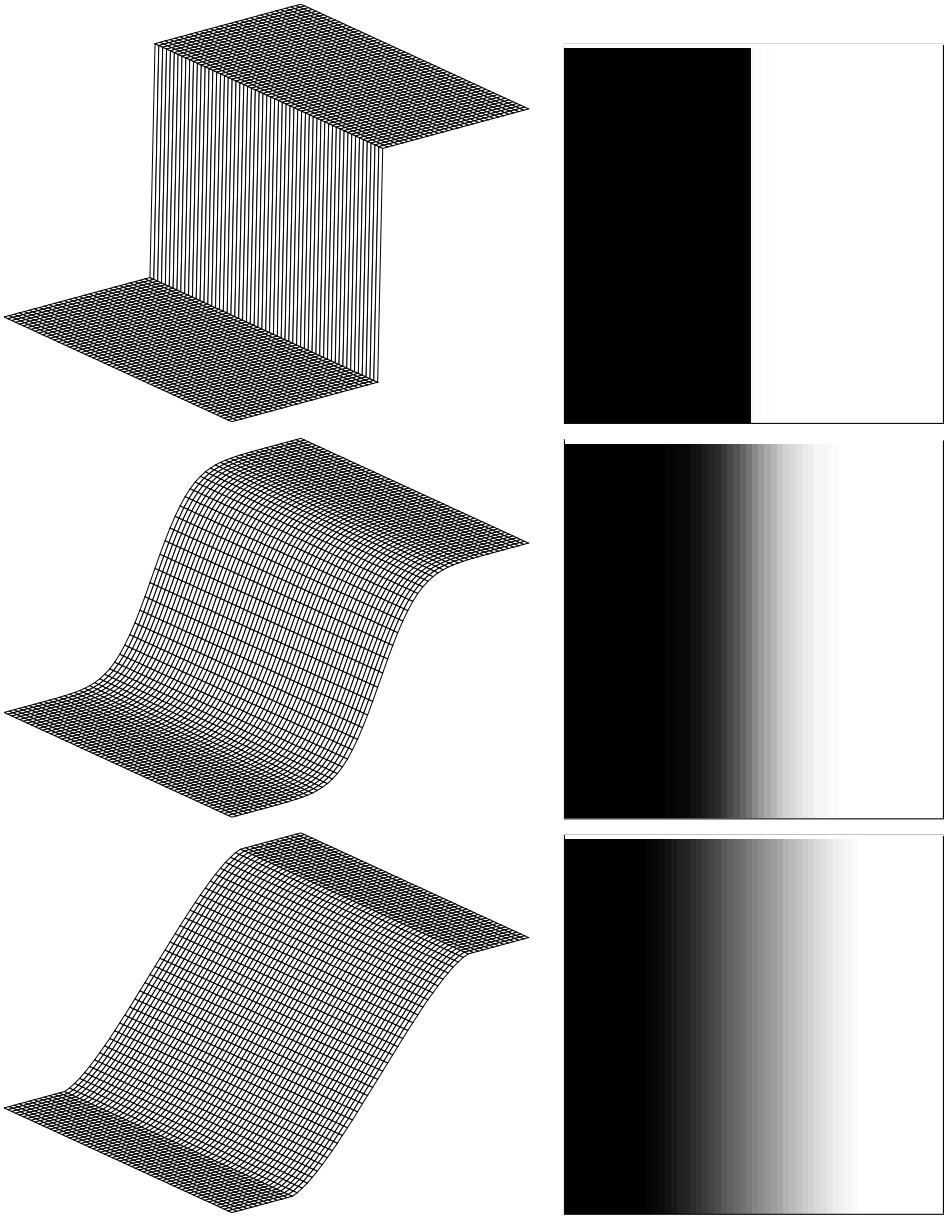

Figure 3: Intensity graphs (left) and images (right) of a vertical step function (top), and of the same step function smoothed with a Gaussian (middle), and with a pillbox function (bottom). Gaussian and pillbox have the same support and the same integral.

which leads to the equality

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

where

$$g_1(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \tag{7}$$

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

Thus, the Gaussian of equation (6) separates into two equal factors. This is computationally very important. In fact, the convolution (4) can then itself be separated into two one-dimensional convolutions:

$$h(i,j) = \sum_{a=-n}^{n} g_1(a) \sum_{b=-n}^{n} g_1(b) f(i-a, j-b) \tag{8}$$

(we also used equation (5) 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 separate sums in (8), on the other hand, can be computed by m multiplications and $m-1$ additions for the internal summation, followed by m more multiplications and $m-1$ more additions for the external summation. Thus, the operation count decreases to $2m$ multiplications and $2(m-1)$ additions. If for instance $m = 21$, we need only 42 multiplications instead of 441.

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, since we define $g(a,b)$ 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 (6), and then normalize it by sum of the samples:

$$\begin{aligned} g_0(a,b) &= e^{-\frac{1}{2}\frac{a^2+b^2}{\sigma^2}} \\ c &= \sum_{a=-n}^{n} \sum_{b=-n}^{n} g_0(a,b) \\ g(a,b) &= \frac{1}{c} g_0(a,b) \ . \end{aligned} \tag{9}$$

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:

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

3 Derivatives

In order to compute derivatives in discrete images, one needs a model for how the underlying continuous[5] 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. In fact, 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:[6]

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

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

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

In fact, with this constraint we have

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

on all integer grid points. In other words, this constraint guarantees 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.

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

[6] For simplicity, the x and y axes are assumed to point along columns and rows, respectively.

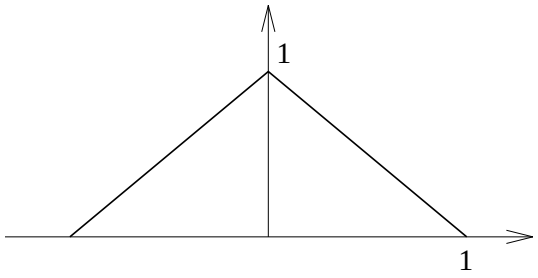

Figure 4: The triangle function interpolates linearly.

Exercise. Verify the last statement.

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,

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

Finally, we 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 horizontal variable:

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

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

$$p(x,y) = \mathrm{sinc}(x,y) = \frac{\sin \pi x}{\pi x}\,\frac{\sin \pi y}{\pi y}\ . \tag{11}$$

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 (11) must be large, which is a computationally undesirable state of affairs. In addition, if there is aliasing, the sinc function will amplify its effects, since 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. 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 j direction is represented by

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

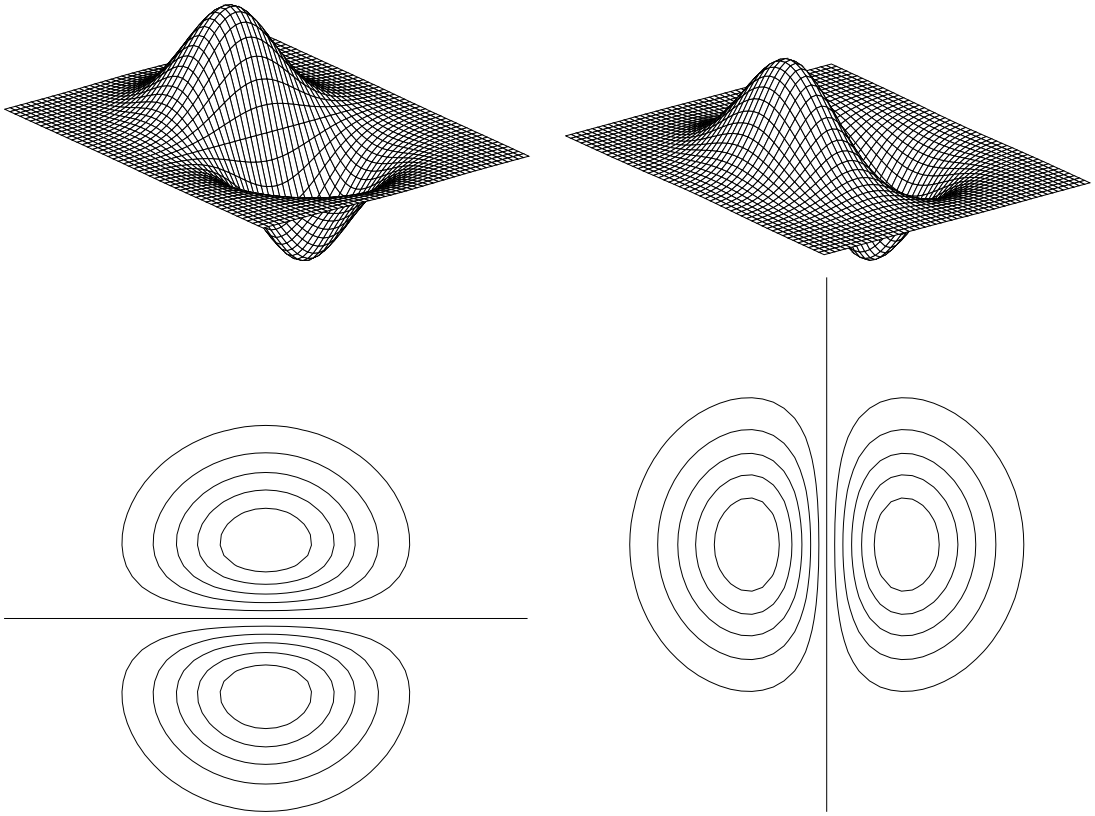

Figure 5: The partial derivatives of a Gaussian function with respect to x (left) and y (right) represented by plots (top) and isocontours (bottom). In the isocontour plots, the x variable points vertically down and the y variable points horizontally to the right.

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

$$c = -\frac{1}{\sum_{a=-n}^{n} \sum_{b=-n}^{n} b g_{0x}(a,b)} \ . \tag{12}$$

Since the partial derivative $g_{0x}(a,b)$ of the Gaussian function with respect to b is negative for positive b, 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:

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

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 (7). 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:

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

Normalization can also be done separately: the one-dimensional Gaussian g_1 is normalized according to (10), and the one-dimensional Gaussian derivative $d_1(a)$ is normalized by the one-dimensional equivalent of (12):

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