Correlation, Convolution, and Filtering

Carlo Tomasi

January 16, 2023

This note discusses the closely-related image-processing operations of *correlation* and *convolution*, which are pervasive in image processing and computer vision. A simple pattern matching problem described in Section 1 motivates correlation. Convolution is only slightly different from correlation and is introduced in Section 2. Section 3 discusses a first application of convolution, that is, image filtering for smoothing and noise reduction.

1 Image Correlation

The image in figure 1(a) shows a detail of the ventral epidermis of a fruit fly embryo viewed through a microscope. Biologists are interested in studying the shapes and arrangement of the dark, sail-like shapes that are called *denticles*.

A simple idea for writing an algorithm to find the denticles automatically is to create a *template* T, that is, an image of a typical denticle. Figure 1(b) shows a possible template, which was obtained by blurring (more on blurring later) a detail out of another denticle image. One can then place the template at all possible positions (r, c) of the input image I, where r is the row coordinate and c denotes the column, and somehow measure the similarity between the template T and a *window* $W(r, c)$ out of I, of the same shape and size as T. Places where the similarity is high are declared to be denticles, or at least image regions worthy of further analysis.

In practice, different denticles, although more or less similar in shape, may differ even dramatically in size and orientation, so the scheme above needs to be refined, perhaps by running the denticle detector with templates (or images) of varying scale and rotation. For now, let us focus on the simpler situation in which the template and the image are fixed.

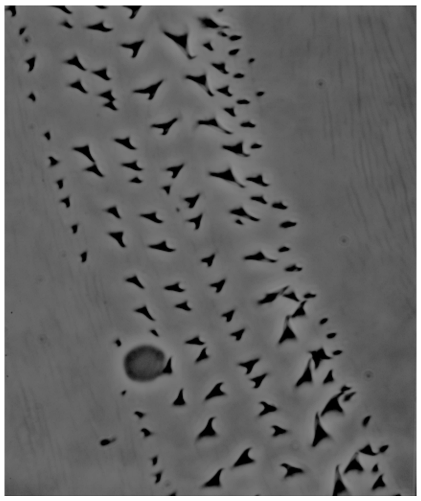
If the pixels values in template T and window $W(r, c)$ are strung into vectors $\mathbf{t}$ and $\mathbf{w}(r, c)$, one way to measure their similarity is to take the inner product

$$\rho(r,c) = \boldsymbol{\tau}^T \boldsymbol{\omega}(r,c) \tag{1}$$

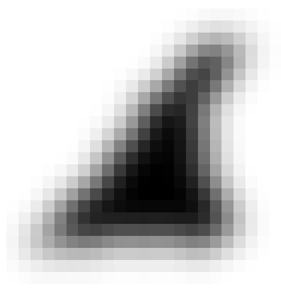
of *nomalized* versions

$$\boldsymbol{\tau} = \frac{\mathbf{t} - m_{\mathbf{t}}}{\|\mathbf{t} - m_{\mathbf{t}}\|} \quad \text{and} \quad \boldsymbol{\omega}(r,c) = \frac{\mathbf{w}(r,c) - m_{\mathbf{w}(r,c)}}{\|\mathbf{w}(r,c) - m_{\mathbf{w}(r,c)}\|} \tag{2}$$

of $\mathbf{t}$ and $\mathbf{w}(r, c)$, where $m_{\mathbf{t}}$ and $m_{\mathbf{w}(r,c)}$ are the mean values of $\mathbf{t}$ and $\mathbf{w}(r, c)$ respectively. Subtracting the means makes the resulting vectors insensitive to possible changes in image brightness, and dividing by the vector norms makes them insensitive to possible changes in image contrast. The



(a)



(b)

Figure 1: (a) Denticles on the ventral epidermis of a *Drosophila* embrio. Image courtesy of Daniel Kiehart, Duke University. (b) A denticle template.

dot product of two unit vectors is equal to the cosine of the angle between them, and therefore the correlation coefficient is a number between -1 and 1:

$$-1 \leq \rho(r,c) \leq 1 \ .$$

It is easily verified that $\rho(r,c)$ achieves the value 1 when $W(r,c) = \alpha T + \beta$ for some positive number α and arbitrary number β, and it achieves the value -1 when $W(r,c) = \alpha T + \beta$ for some negative number α and arbitrary number β. In words, $\rho(r,c) = 1$ when the window $W(r,c)$ is identical to template T except for brightness β or contrast α, and $\rho(r,c) = -1$ when $W(r,c)$ is a contrast-reversed version of T with possibly scaled contrast α and brightness possibly shifted by β.

The operation (1) of computing the inner product of a template with the contents of an image window—when the window is slid over all possible image positions (r,c)—is called *cross-correlation*, or *correlation* for short. When the normalizations (2) are applied first, the operation is called *normalized cross-correlation*. Since each image position (r,c) yields a value ρ, the result is another image, in the sense that it is an array of values. However, the pixel values in the output image can be positive or negative.

For simplicity, we now explore correlation by itself, rather than its normalized version. So let us think about the correlation of an image I and a template T without normalization. The inner product between the vector version $\mathbf{t}$ of T and the vector version $\mathbf{w}(r,c)$ of window $W(r,c)$ at position (r,c) in the image I can be spelled out as follows:

$$J(r,c) = \sum_{u=-h}^{h} \sum_{v=-h}^{h} I(r+u, c+v) T(u,v) \qquad (3)$$

where J is the resulting output image. For simplicity, the template T and the window $W(r,c)$ are assumed to be squares with $2h+1$ pixels on their side—so that h is a bit less than half the width of the window (or template). This expression can be interpreted as follows:

Place the template T with its center at pixel (r, c) in image I. Multiply the template values with the pixel values under them, add up the resulting products, and put the resulting sum into pixel $J(r, c)$ of the output image. Repeat for all positions (r, c) in I.

Note that we do not alter I, and the output image J is a fresh image. In code, if the output image has `m` rows and `n` columns:

```
for r = 1:m
   for c = 1:n
      J(r, c) = 0
      for u = -h:h
         for v = -h:h
            J(r, c) = J(r, c) + T(u, v) * I(r+u, c+v)
         end
      end
   end
end
```

In practice, we need to make sure that the window $W(r, c)$ does not fall off the image boundaries. We will consider this aspect later.

If you are curious, Figure 2(a) shows the normalized cross-correlation for the image and template in Figure 1. The code also considers multiple scales and rotations, and returns the best matches after additional image cleanup operations (Figure 2(b)). Pause to look for false positive and false negative detections. Again, just to satisfy your curiosity, the full code is listed in the Appendix. Look for the call to `cv.matchTemplate`, the Python OpenCV implementation of 2-dimensional normalized cross correlation. This code contains too many "magic numbers" to be useful in general, and is used here for pedagogical reasons only.

2 Image Convolution

Two-dimensional convolution is the same as two-dimensional correlation but for two minus signs:

$$J(r, c) = \sum_{u=-h}^{h} \sum_{v=-h}^{h} I(r-u, c-v) H(u, v) \ .$$

The "template" H is now called the *kernel* of the convolution. By applying the changes of variables $u \leftarrow -u$, $v \leftarrow -v$, we can write

$$J(r, c) = \sum_{u=-h}^{h} \sum_{v=-h}^{h} I(r+u, c+v) H(-u, -v) \ .$$

Thus, convolution with the kernel $H(u, v)$ is the same as correlation with the template $T(u, v) = H(-u, -v)$. In other words, to compute a convolution, one flips the kernel upside-down and left-to-right[1] and performs a correlation with the resulting template.[2] For instance:

[1]Take a minute to verify that it does not matter which flip you apply first.

[2]Of course, for symmetric kernels this flip makes no difference.

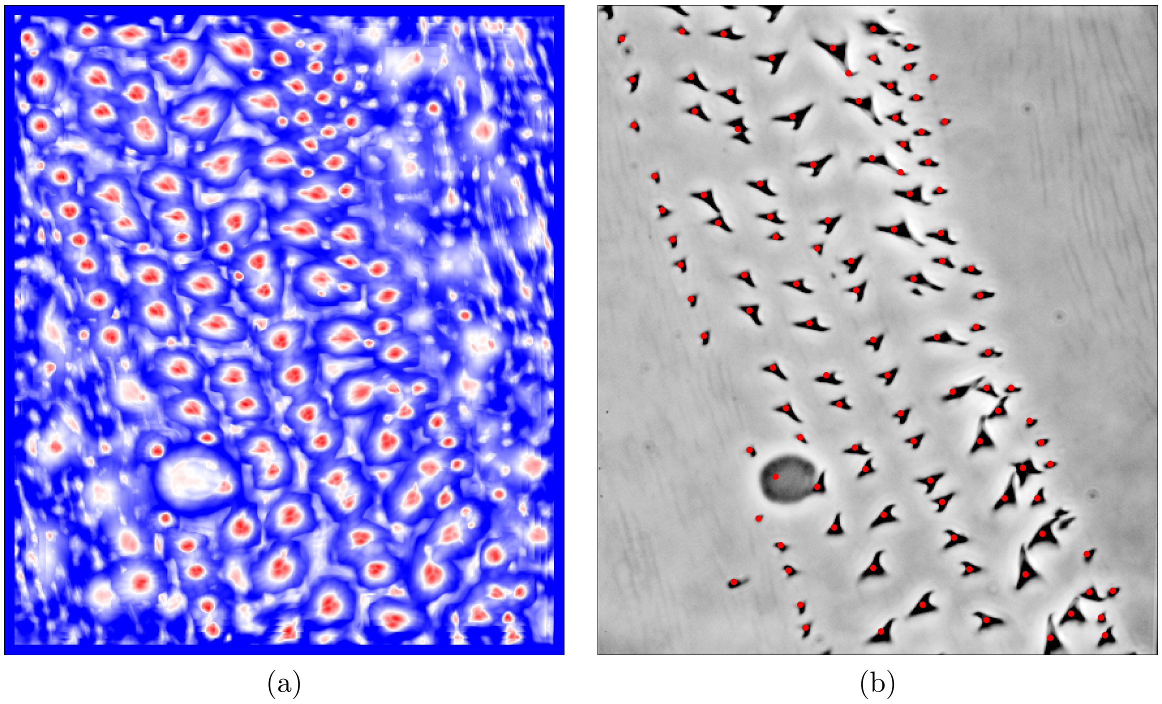


Figure 2: (a) Rotation- and scale-sensitive correlation image $\rho(r, c)$ for the image in Figure 1 (a). Positive peaks (red areas) correlate with denticle locations. Blue represents negative values, and zeros are rendered in white. (b) Red dots are cleaned-up maxima in the correlation image, superimposed on the input image.

$$H = \begin{array}{ccc} 11 & 12 & 13 \\ 21 & 22 & 23 \end{array} \quad \rightarrow \quad T = \begin{array}{ccc} 23 & 22 & 21 \\ 13 & 12 & 11 \end{array} \; .$$

For either operation (correlation or convolution), mathematical manipulation becomes easier if the domains of both kernel (or template) and image are extended to the entire integer plane $\mathbb{Z}^2$ by the convention that unspecified values are set to zero. In this way, the summations in the definition of convolution can be extended to the entire plane as well:[3]

$$J(r, c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(u, v) I(r - u, c - v) \tag{4}$$

for convolution and

$$J(r, c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} T(u, v) I(r + u, c + v) \tag{5}$$

for correlation. This simpler notation lets us examine more easily one of the reasons why mathematicians prefer convolution over correlation: The changes of variables $u \leftarrow r - u$ and $v \leftarrow c - v$

[3] It would not be wise to do so also in the equivalent program!

entail the equivalence of equation (4) with the following expression:

$$J(r,c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(r-u, c-v) I(u,v) \tag{6}$$

which shows that "image" and "kernel" are interchangeable. Thus, *convolution commutes.* If we introduce the symbol '$*$' for convolution we can conclude as follows.

The *convolution* of an image I with the kernel H is defined as follows:

$$\begin{aligned} J(r,c) &= [I * H](r,c) = [H * I](r,c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(u,v) I(r-u, c-v) \\ &= \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(r-u, c-v) I(u,v) \quad \text{for} \quad (r,c) \in \mathbb{Z}^2 \ . \end{aligned} \tag{7}$$

In contrast, correlation does *not* commute: The changes of variables $u \leftarrow r+u$ and $v \leftarrow c+v$ yield the following expression equivalent to (5):

$$J(r,c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} T(u-r, v-c) I(u,v) \ ,$$

so correlating T with I (as opposed to correlating I with T) yields an image $J(-r,-c)$ that is the upside-down and left-to-right flipped version of $J(r,c)$: By replacing r with $-r$ and c with $-c$, we obtain

$$J(-r,-c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} T(u+r, v+c) I(u,v) \ ,$$

which is the correlation of T with I.

A Mathematical Aside A second reason why mathematicians prefer convolution over correlation is that convolution in one dimension has a deep connection with polynomial multiplication. Consider for instance the two sequences of real numbers

$$a = (a_0, a_1, a_2) \quad \text{and} \quad b = (b_0, b_1, b_2, b_3) \ .$$

If we think of these two sequences as extending their domains indefinitely with zeros,

$$a = (\ldots, 0, a_0, a_1, a_2, 0, \ldots) \quad \text{and} \quad b = (\ldots, 0, b_0, b_1, b_2, b_3, 0, \ldots) \ ,$$

then the nonzero terms of their (one-dimensional) convolution form the sequence

$$c = a * b = (a_0 b_0,\ a_0 b_1 + a_1 b_0,\ a_0 b_2 + a_1 b_1 + a_2 b_0,\ a_0 b_3 + a_1 b_2 + a_2 b_1,\ a_1 b_3 + a_2 b_2,\ a_0 b_3) \ .$$

These terms are the coefficients $c_0, \ldots, c_5$ of the product

$$c_0 + c_1 x + c_2 x^2 + c_3 x_3 + c_4 x^4 + c_5 x^5$$

of the two polynomials

$$a_0 + a_1 x + a_2 x^2 \quad \text{and} \quad b_0 + b_1 x + b_2 x^2 + b_3 x_3 \ .$$

Thus, multiplying two polynomials corresponds to convolving their coefficients. This connection has important ramifications, since Fourier transformations are polynomials (in the variable $\frac{1}{n}e^{-2\pi i/n}$). However, we will not need these considerations in this course.

Practical Aspects: Image Boundaries. The convolution neighborhood becomes undefined at pixel positions that are very close to the image boundaries, where the kernel window is not entirely contained in the image. Typical solutions to this issue include the following:

- Consider images and kernels to extend with zeros where they are not defined, and then only output the nonzero part of the result. See Figure 3, top. This *full convolution* yields an output image J that is larger than the input image I. For instance, convolution of an $m \times n$ image with a $k \times l$ kernel yields an image of size $(m + k - 1) \times (n + l - 1)$. Full convolution corresponds to polynomial multiplication, as discussed above.
- Define the output image J to be smaller than the input I, so that the values of pixels in J that are too close to the image boundaries are not computed. See Figure 3, middle. For instance, convolution of an $m \times n$ image with a $k \times l$ kernel yields an image of size $(m - k + 1) \times (n - l + 1)$. This *valid convolution* is the least committal of all solutions, in that it does not make up spurious pixel values outside the input image. However, this solution shares with the previous one the disadvantage that image sizes vary with the number and neighborhood sizes of the convolutions performed on them.
- Pad the input image with a rim of zero-valued pixels that is wide enough that the convolution kernel H fits inside the padded image whenever its center is placed anywhere on the unpadded image. See Figure 3, bottom. This *"same" convolution* is simple to implement (although not as simple as the previous one), and preserves image size: J is as big as I. However, now the input image has an unnatural frame of zeros (black pixels) all around it, just as it does for the full convolution. This causes problems for certain types of kernels. For instance, an averaging kernel is used to smooth (or blur) images. Such a kernel contains nonnegative values that add up to one.[4] Then, the output image merely darkens at the edges, because of all the zeros that enter the calculation. If the kernel is designed so that convolution with it computes image derivatives (a situation described later on), the value jumps around the rim of the image yield very large values in the output image.
- Pad the input image with replicas of the boundary pixels (with any of the three padding styles described above) rather than with zeros. For instance, padding with a 2-pixel rim around a 4×5 image would look as follows:

$$\begin{bmatrix} i_{11} & i_{12} & i_{13} & i_{14} & i_{15} \\ i_{21} & i_{22} & i_{23} & i_{24} & i_{25} \\ i_{31} & i_{32} & i_{33} & i_{34} & i_{35} \\ i_{41} & i_{42} & i_{43} & i_{44} & i_{45} \end{bmatrix} \rightarrow \begin{bmatrix} i_{11} & i_{11} & i_{11} & i_{12} & i_{13} & i_{14} & i_{15} & i_{15} & i_{15} \\ i_{11} & i_{11} & i_{11} & i_{12} & i_{13} & i_{14} & i_{15} & i_{15} & i_{15} \\ i_{11} & i_{11} & \mathbf{i}_{11} & \mathbf{i}_{12} & \mathbf{i}_{13} & \mathbf{i}_{14} & \mathbf{i}_{15} & i_{15} & i_{15} \\ i_{21} & i_{21} & \mathbf{i}_{21} & \mathbf{i}_{22} & \mathbf{i}_{23} & \mathbf{i}_{24} & \mathbf{i}_{25} & i_{25} & i_{25} \\ i_{31} & i_{31} & \mathbf{i}_{31} & \mathbf{i}_{32} & \mathbf{i}_{33} & \mathbf{i}_{34} & \mathbf{i}_{35} & i_{35} & i_{35} \\ i_{41} & i_{41} & \mathbf{i}_{41} & \mathbf{i}_{42} & \mathbf{i}_{43} & \mathbf{i}_{44} & \mathbf{i}_{45} & i_{45} & i_{45} \\ i_{41} & i_{41} & i_{41} & i_{42} & i_{43} & i_{44} & i_{45} & i_{45} & i_{45} \\ i_{41} & i_{41} & i_{41} & i_{42} & i_{43} & i_{44} & i_{45} & i_{45} & i_{45} \end{bmatrix} .$$

  This is a relatively simple solution, avoids spurious discontinuities, and preserves image size.

The Python convolution function `scipy.signal.convolve2d` provides `mode` options `'full'`, `'valid'`, and `'same'` to implement the first three alternatives above, in that order. The function `scipy.signal.correlate2d` performs correlation. The Python Open CV function `cv2.filter2D()` performs correlation for image filtering (see next Section) and also provides various types of padding.

[4]More on smoothing kernels in Section 3.

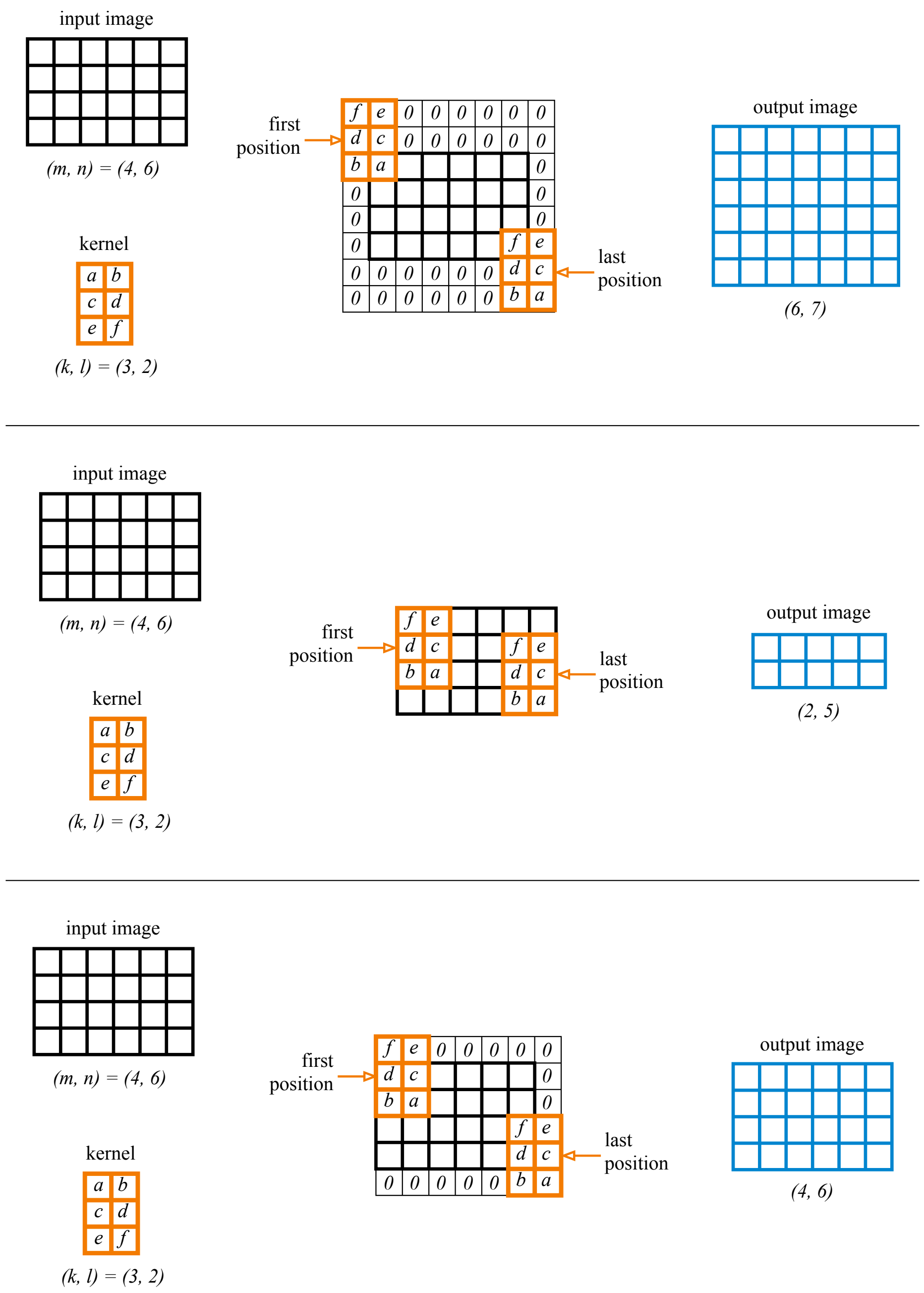


Figure 3: The full (top), valid (middle) and “same” (bottom) styles of convolution.

3 Filters

Convolutions are pervasive in image processing, where they are used for various purposes, including the reduction of the effects of image noise and image differentiation. We consider the first of these uses here.

Images are taken with cameras, and the electronics of a camera introduce random noise as a result of the thermal agitation of electrons in the circuits. In addition, a camera's sensing elements (pixels) effectively count the number of photons that hit them during the picture's exposure period. This number can vary at random, resulting in further noise. Finally, even the output of a perfect, ideal camera are eventually stored as integer pixel values in a computer array (typically eight-bit integers). Clipping real values to integers results in another discrepancy between true and recorded image called *pixel quantization*. While quantization is deterministic, the value being quantized is not, so even pixel quantization can be described as random noise.

The effects of random noise on images can be reduced by smoothing, that is, by replacing every pixel value by a weighted average of the values of its neighbors. The reason why this helps can be understood by thinking of an image patch that is small enough that the image intensity function I is well approximated by its first-order Taylor expansion around the center (r, c) of the patch (that is, a locally affine[5] intensity function). Then, by (odd) symmetry, the average value of the patch is $I(r, c)$, so that averaging does not alter image value. On the other hand, noise added to the image can usually be assumed to be zero mean, so that averaging reduces the noise component. Since both filtering and summation are linear, they can be switched: the result of filtering image plus noise is equal to the result of filtering the image (which does not alter values) plus the result of filtering noise (which reduces noise). The net outcome is an increase of the signal-to-noise ratio.

For independent noise values, noise reduction is proportional to the square root of the number of pixels in the smoothing window, so a large window is preferable. However, the assumption that the image intensity is approximately affine within the smoothing window fails more and more as the window size increases, and is violated particularly badly along edges, that is, curves in the image where the image intensity function is nearly discontinuous. As a consequence, smoothing blurs edges. Thus, when smoothing, a compromise must be reached between noise reduction and image blurring.

Averaging in a $k \times k$ window amounts to multiplying all values by $1/k^2$ and adding up the results. In other words, convolving with a kernel with values $1/k^2$. For instance, when $k = 3$, the averaging kernel is

$$H = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} .$$

However, more general kernels can be used. Any smoothing kernel is usually rotationally symmetric, as there is no reason to privilege, say, the pixels on the left of a given pixel over those on its right[6]:

$$H(v, u) = \gamma(\rho)$$

[5] "Affine" means linear plus constant: $y = ax$ is linear and $y = ax + b$ is affine if $b \neq 0$.

[6] 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.

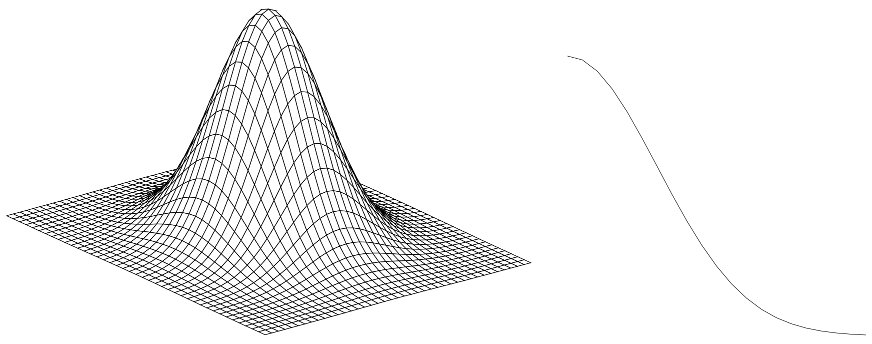


Figure 4: The two dimensional kernel on the left can be obtained by rotating the function $\gamma(r)$ on the right around a vertical axis through the maximum of the curve ($r = 0$).

where

$$\rho = \sqrt{u^2 + v^2}$$

is the distance from the center of the kernel to its pixel (u, v). Thus, a rotationally symmetric kernel can be obtained by rotating a one-dimensional function $\gamma(\rho)$ defined on the nonnegative reals around the origin of the plane (figure 4).

The plot in figure 4 was obtained from the (unnormalized) Gaussian function

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

with $\sigma = 6$ pixels (one pixel corresponds to one cell of the mesh in figure 4), so that

$$H(v, u) = G(v, u) \overset{\text{def}}{=} e^{-\frac{1}{2}\frac{u^2+v^2}{\sigma^2}} \ . \tag{8}$$

The greater σ is, the more smoothing occurs.

This Gaussian kernel is by far the most popular smoothing function in computer vision. 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, rotationally symmetric. It also emphasizes nearby pixels over more distant ones, and therefore reduces smearing (blurring). And yet smoothing still occurs, because convolution with a properly normalized Gaussian kernel still performs averaging. We will look at normalization soon.

A more quantitative, useful property of the Gaussian function is its smoothness. If $G(v, u)$ is considered as a function of real variables u, v, it is differentiable infinitely many times. Although this property by itself is not too useful with discrete images, it implies that the function is composed by as compact a set of frequencies as possible.[7]

Another important property of the Gaussian function for computer vision 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.

[7]This last sentence will only make sense to you if you have had some exposure to the Fourier transform. If not, it is OK to ignore this statement.

Practical Aspects: Separability. An important property of the Gaussian function from a programming standpoint is its *separability*. A function $G(x, y)$ is said to be separable if there are two functions g and g' of one variable such that

$$G(x, y) = g(x)g'(y) \; .$$

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

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

which leads to the equality

$$G(x, y) = g(x)g(y)$$

where

$$g(x) = e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \tag{9}$$

is the one-dimensional (unnormalized) Gaussian.

Thus, the Gaussian of equation (8) separates into two factors, one in x and one in y, which happen to be equal to each other. This separation has useful computational consequences. Suppose that for the sake of concrete computation we revert to a finite domain for the kernel function. Because of symmetry, the kernel is defined on a square, say $[-h, h]^2 \subset \mathbb{Z}^2$. With a separable kernel, the convolution (7) can then itself be separated into two one-dimensional convolutions:

$$J(r, c) = \sum_{u=-h}^{h} g(u) \sum_{v=-h}^{h} g(v) I(r - u, c - v) \; , \tag{10}$$

with substantial savings in the computation. In fact, the double summation

$$J(r, c) = \sum_{u=-h}^{h} \sum_{v=-h}^{h} G(v, u) I(r - u, c - v)$$

requires m^2 multiplications and $m^2 - 1$ additions, where $m = 2h + 1$ is the number of pixels in one row or column of the convolution kernel $G(v, u)$. The sums in (10), on the other hand, can be rewritten so as to be computed by $2m$ multiplications and $2(m - 1)$ additions as follows:

$$J(r, c) = \sum_{u=-h}^{h} g(u)\, \phi(r - u, c) \tag{11}$$

where

$$\phi(r, c) = \sum_{v=-h}^{h} g(v) I(r, c - v) \; . \tag{12}$$

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 (12) and (11): first convolve the entire image with a horizontal version of g in the horizontal direction, then convolve the resulting image with a vertical version of g in the vertical direction (or in the opposite order, since convolution commutes). If we were to perform (10) literally, there would be no gain, as for each value of $r - u$, the internal summation is recomputed m times, since any fixed value $d = r - u$ occurs for pairs $(r, u) = (d - h, -h), (d - h + 1, -h + 1), \ldots, (d + h, h)$ when equation (10) is computed for every pixel (r, c).

Thus, separability can decrease the operation count to $2m$ multiplications and $2(m - 1)$ additions, with an approximate gain in efficiency by a factor

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

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 $\gamma(\rho)$ and $g(x)$. Is this a coincidence?

Practical Aspects: Truncation and Normalization. The Gaussian functions in this section were defined with no attention paid to normalization. However, proper normalization must be performed 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.

Proper normalization for a smoothing filter is simple. We first compute the unscaled version of, say, the Gaussian in equation (8), and then normalize it by sum of the samples:

$$\begin{aligned}\tilde{G}(v,u) &= e^{-\frac{1}{2}\frac{u^2+v^2}{\sigma^2}} \\ c_{\tilde{G}} &= \sum_{i=-h}^{h}\sum_{j=-h}^{h}\tilde{G}(j,i) \\ G(v,u) &= \frac{1}{c_{\tilde{G}}}\tilde{G}(v,u)\ .\end{aligned} \tag{13}$$

To verify that this yields the desired normalization, consider an image with constant intensity I_0. Then its convolution with the new $G(v,u)$ should yield I_0 everywhere as a result. In fact, we have

$$\begin{aligned}J(r,c) &= \sum_{u=-h}^{h}\sum_{v=-h}^{h} G(v,u)I(r-u,c-v) \\ &= I_0\sum_{u=-h}^{h}\sum_{v=-h}^{h} G(v,u) \\ &= I_0\end{aligned}$$

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:

$$\begin{aligned}\tilde{g}(u) &= e^{-\frac{1}{2}\left(\frac{u}{\sigma}\right)^2} \\ c_{\tilde{g}} &= \sum_{v=-h}^{h}\tilde{g}(v) \\ g(u) &= \frac{1}{c_{\tilde{g}}}\tilde{g}(u)\ .\end{aligned} \tag{14}$$

Appendix

Python Code for Denticle Detection

[This code is available here for download.]

```
import numpy as np
import cv2 as cv
from math import ceil, floor, sqrt

def denticles(img, template):

    # Does a window around p contain a dark enough pixel?
    def isDark(img, p, thr):
        rows = range(p[0]-5, p[0]+6)
        cols = range(p[1]-5, p[1]+6)
        win = img[rows, cols]
        mn = np.amin(win)
        return mn <= thr

    m, n = img.shape
    corrThreshold, minArea = 0.5, m * n / 10000 # Smallest accepted correlation
    maxValue = np.percentile(img, 5) # Brightest accepted denticle pixel

    # Range of rotations and scales to consider
    rotations = np.linspace(0, 360, 17)
    rotations = rotations[:-1]
    scales = np.linspace(0.5, 2.0, 6)

    # Dimensions and center of template image
    srcShape = tuple(template.shape[1-k] for k in range(2))
    srcCenter = tuple(srcShape[k]/2 for k in range(2))

    # Loop over rotations and scales and store the best correlation at every pixel
    cmax = np.zeros(img.shape) # Max correlation value at every pixel
    for rotation in rotations:
        for scale in scales:
            # Dimensions and center of transformed template image
            dstShape = tuple(ceil(srcShape[k] * scale * sqrt(2)) for k in range(2))
            dstCenter = tuple(dstShape[k]/2 for k in range(2))

            # Transofrmed template image
            xform = cv.getRotationMatrix2D(srcCenter, rotation, scale)
            adjust = [dstCenter[k] - srcCenter[k] for k in range(2)]
            xform[:, 2] += adjust
            t = cv.warpAffine(template, xform, dstShape,
                              borderValue=255)

            # Compute the correlation image for this (rotation, scale) pair
            c = cv.matchTemplate(img, t, cv.TM_CCOEFF_NORMED)
            # Ensure that correlation image has the same size as img
            h = [floor((img.shape[k] - c.shape[k]) / 2) for k in range(2)]
            cpad = np.zeros(img.shape)
            cpad[h[0]:(h[0]+c.shape[0]), h[1]:(h[1]+c.shape[1])] = c

            # Accumulate the maximum correlation at every pixel
            cmax = np.maximum(cmax, cpad)

    # Find peak regions in cmax
    peak = (255 * (cmax > corrThreshold)).astype(np.uint8)
    image, contours, hierarchy = cv.findContours(peak, cv.RETR_CCOMP,
                                                 cv.CHAIN_APPROX_NONE)

    # Retain centroids of the peak regions that are large enough and
    # contain a dark enough pixel
    areas = [cv.contourArea(c) for c in contours]
    ok = []
    for area, contour in zip(areas, contours):
        if area >= minArea:
            m = cv.moments(contour)
            centroid = (int(m['m01']/m['m00']), int(m['m10']/m['m00']))
            if isDark(img, centroid, maxValue):
                ok.append((contour, area, centroid))

    # Repackage centroids into a numpy array of (row, column) coordinates
    centroids = np.array([[item[2][1], item[2][0]] for item in ok])

    return centroids, cmax
```