Image Correlation, Convolution and Filtering

Carlo Tomasi

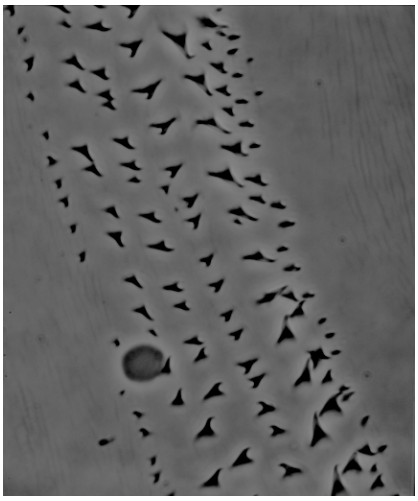
January 13, 2019

This note discusses the basic image operations of *correlation* and *convolution*, and some aspects of one of the applications of convolution, image *filtering*. Image correlation and convolution differ from each other by two mere minus signs, but are used for different purposes. Correlation is more immediate to understand, and the discussion of convolution in section 2 clarifies the source of the minus signs.

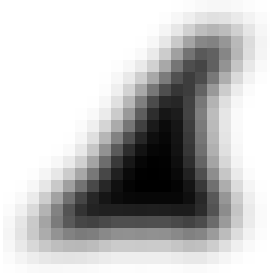
1 Image Correlation

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 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.



(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.

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 kept constant.

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 make the resulting vectors insensitive to image brightness, and dividing by the vector norms makes them insensitive to image contrast. The 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, although the pixel values now can be positive or negative.

For simplicity, 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:

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

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:

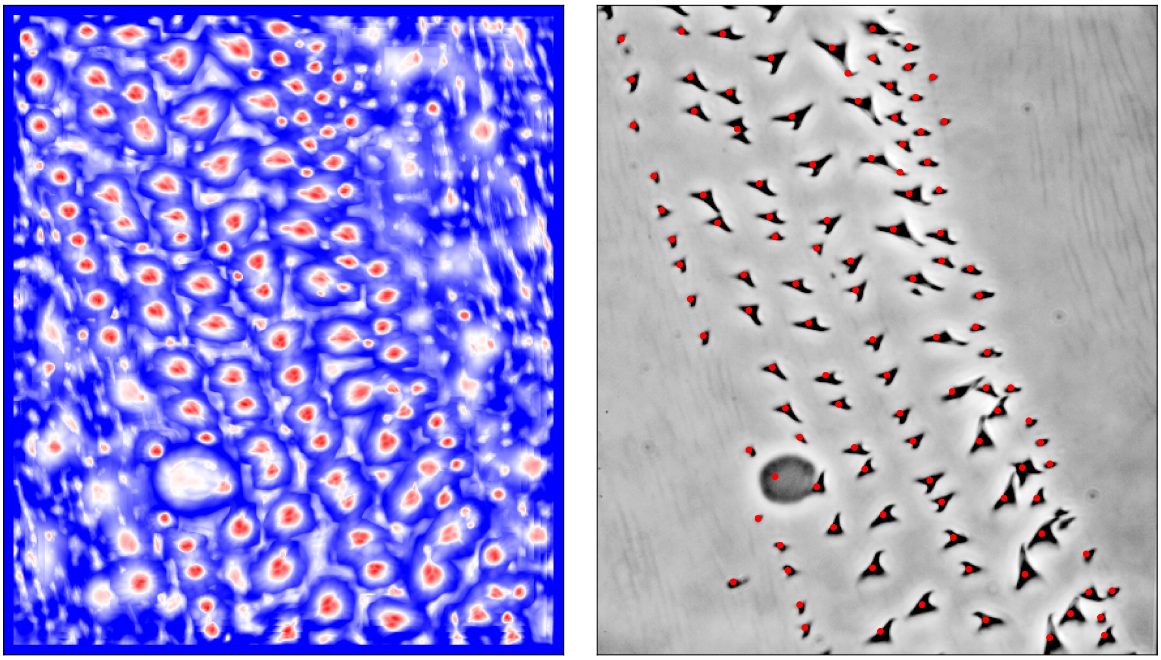
> Place the template T with its center at pixel (r, c) in image I. Multiply the template values with the pixel values under them, add up the resulting products, and put the result in pixel $J(r, c)$ of the output image. Repeat for all positions (r, c) in I.

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
```

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



(a) (b)

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. (b) Red dots are cleaned-up maxima in the correlation image, superimposed on the input image.

2 Image Convolution

The short story here is that convolution is the same as correlation but for two minus signs:

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

Equivalently, by applying the changes of variables $u \leftarrow -u$, $v \leftarrow -v$,

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

So before placing the template T onto the image, one flips it upside-down and left-to-right.[1]

If this is good enough for you, you can skip to the paragraph just before equation (5) on page 7. If the two minus signs irk you (they should), read on.

The operation of convolution can be understood by referring to an example in optics. If a camera lens is out of focus, the image appears to be blurred: Rays from any one point in the world are spread out into a small patch as they reach the image. Let us look at this example in two different ways. The first one follows light rays in their natural direction, the second goes upstream. Both ways approximate physics at pixel resolution.

In the first view, the unfocused lens spreads the brightness of a point in the world onto a small circle in the image. We can abstract this operation by referring to an ideal image I that would be obtained in the absence of blur, and to a blurred version J of this image, obtained through some transformation of I.

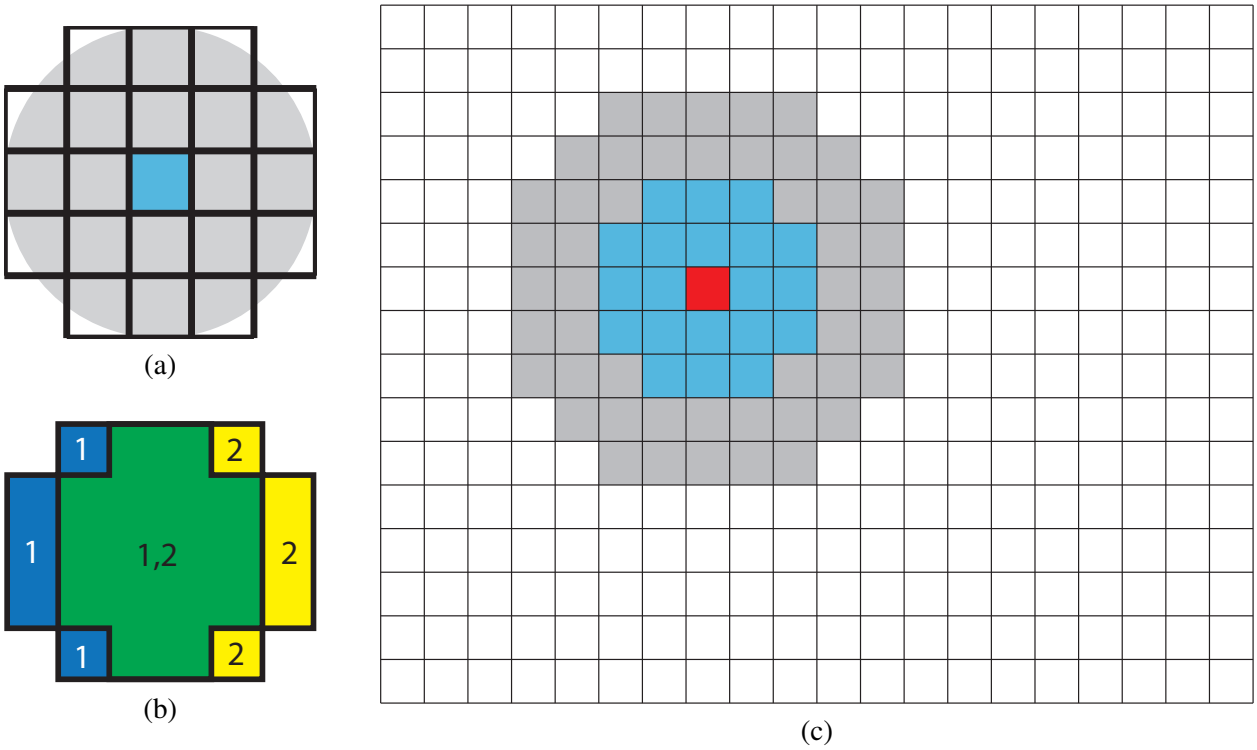
Suppose that the diameter of the blur circle is five pixels. As a first approximation, represent the circle with the cross shape in Figure 3 (a): This is a cross on the pixel grid that includes a pixel if most of it is inside the blur circle. Let us call this cross the *neighborhood* of the pixel at its center.

If the input (focused) image I has a certain value $I(i,j)$ at the pixel in row i, column j, then that value is divided by 21, to reflect the fact that the energy of one pixel is spread over 21 pixels, and then written into the pixels in the neighborhood of pixel (i,j) in the output (blurred) image J. Consider now the pixel just to the right of (i,j) in the input image, at position $(i,j+1)$. That will have a value $I(i,j+1)$, which in turn needs to be written into the pixels of the neighborhood of $(i,j+1)$ in the output image J. However, the two neighborhoods just defined overlap. What value is written into pixels in the overlap region?

The physics is additive: each pixel in the overlap region is lit by both blurs, and is therefore painted with the *sum* of the two values. The region marked $1,2$ in Figure 3(b) is the overlap of two adjacent neighborhoods. Pixels in the areas marked with only 1 or only 2 only get one value.

This process can be repeated for each pixel in the input image I. In programming terms, one can start with all pixels in the output image J set to zero, loop through each pixel (i,j) in the input image, and for each pixel add $I(i,j)$ to the pixels that are in the neighborhood of pixel (i,j) in J. In order to do this, it is convenient to define a *point-spread function*, that is, a 5×5 matrix H that contains a value of 1/21 everywhere except at its four corners, where it contains zeros. For symmetry, it is also convenient to number

[1] Of course, for symmetric templates this flip makes no difference.



(a) (b) (c)

Figure 3: (a) A neighborhood of pixels on the pixel grid that approximates a blur circle with a diameter of 5 pixels. (b) Two overlapping neighborhoods. (c) The union of the 21 neighborhoods with their centers on the blue area (including the red pixel in the middle) forms the gray area (including the blue and red areas in the middle). The intersection of these neighborhoods is the red pixel in the middle.

the rows and columns of H with integers between -2 and 2:

$$H = \frac{1}{21} \begin{array}{c} \begin{array}{ccccc} -2 & -1 & 0 & 1 & 2 \end{array} \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{array}\right] \end{array} \begin{array}{c} -2 \\ -1 \\ 0 \\ 1 \\ 2 \end{array}$$

Use of the matrix H allows writing the summation at each pixel position (i, j) with the two `for` loops in `u` and `v` below:

```
for i = 1:m
  for j = 1:n
    J(i, j) = 0
  end
end
for i = 1:m
  for j = 1:n
    for u = -2:2
      for v = -2:2
        J(i+u, j+v) = J(i+u, j+v) + H(u, v) * I(i, j)
      end
    end
  end
end
```

This first view of the blur operation paid attention to an individual pixel in the input image I, and followed its values as they were dispersed into the output image J.

The second view lends itself better to mathematical notation, and does the reverse of the first view by asking the following question: what is the final value at any given pixel (r, c) in the output image? This pixel receives a contribution from each of the neighborhoods that overlap it, as shown in Figure 3(c). There are 21 such neighborhoods, because each neighborhood has 21 pixels. Specifically, whenever a pixel (i, j) in the input image is within 2 pixels from pixel (r, c) horizontally and vertically, pixel (i, j) is in the square bounding box of the neighborhood. The positions corresponding to the four corner pixels, for which there is no overlap, can be handled, as before, through the point-spread function H to obtain a similar piece of code as before:

```
for r = 1:m
  for c = 1:n
    J(r, c) = 0
    for u = -2:2
      for v = -2:2
        J(r, c) = J(r, c) + H(u, v) * I(r-u, c-v)
      end
    end
  end
end
```

Note that the two outermost loops now scan the output image, rather than the input image.

This program can be translated into mathematical notation as follows:

$$J(r,c) = \sum_{u=-2}^{2} \sum_{v=-2}^{2} H(u,v) I(r-u, c-v) \ . \tag{4}$$

While the two code snippets above yield the same image J, a direct mathematical translation from the earlier piece of code is not obvious. The operation described by equation (4) (or by either code snippet above) is called *convolution*.

Of course, the point-spread function can contain arbitrary values, not just 0 and 1/21. For instance, a better approximation for the blur point-spread function can be computed by assigning to each entry of H a value proportional to the area of the blur circle (the gray circle in Figure 3(a)) that overlaps the corresponding pixel. Simple but lengthy mathematics, or numerical approximation, yields the following values for a 5×5 approximation H to the *pillbox function*, a function that is 1 within the blur circle and 0 elsewhere:

$$H = \begin{bmatrix} 0.0068 & 0.0391 & 0.0500 & 0.0391 & 0.0068 \\ 0.0391 & 0.0511 & 0.0511 & 0.0511 & 0.0391 \\ 0.0500 & 0.0511 & 0.0511 & 0.0511 & 0.0500 \\ 0.0390 & 0.0511 & 0.0511 & 0.0511 & 0.0390 \\ 0.0068 & 0.0390 & 0.0500 & 0.0391 & 0.0068 \end{bmatrix}$$

The values in H add up to one to reflect the fact that the brightness of the input pixel is spread over the support of the pillbox function.

So the minus signs in equation (4) comes from looking at the same process in two different ways: Following light rays forward leads to no minus sign, as the code above showed. Viewing the same process from the point of view of each output pixel leads to the negative signs.

For the specific choice of point-spread function H, these signs do not matter, since $H(u,v) = H(-u,-v)$, so replacing minus signs with plus signs would have no effect. However, if H is not symmetric, the signs make good sense: with u, v positive, say, pixel $(r-u, c-v)$ is to the left of and above pixel (r,c), so the contribution of the input pixel value $I(r-u, c-v)$ to output pixel value $J(r,c)$ is mediated through a low-right value of the point-spread function H. This is illustrated in Figure 4 and, in one dimension, in Figure 5.

Mathematical manipulation becomes easier if the domains of both point-spread function and images are extended to $\mathbb{Z}^2$ 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:[2]

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

The changes of variables $u \leftarrow r-u$ and $v \leftarrow c-v$ then entail the equivalence of equation (5) with the following expression:

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

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

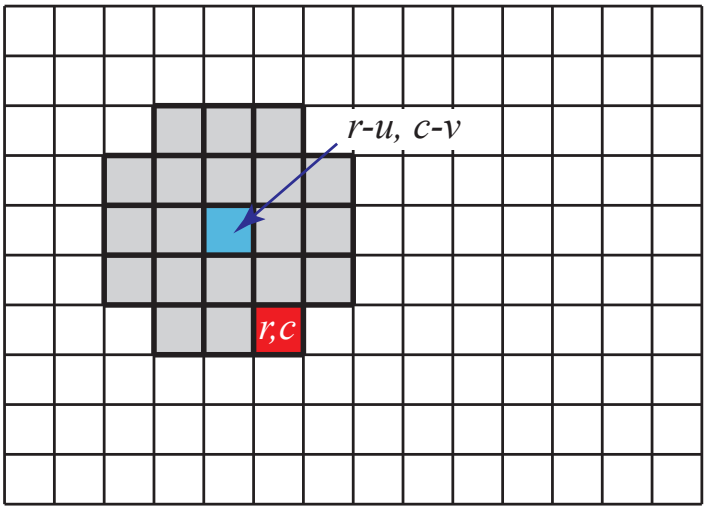


Figure 4: The contribution of input pixel $I(r-2, c-1)$ (blue) with to output pixel $J(r, c)$ (red) is determined by entry $H(2, 1)$ of the point-spread function (gray).

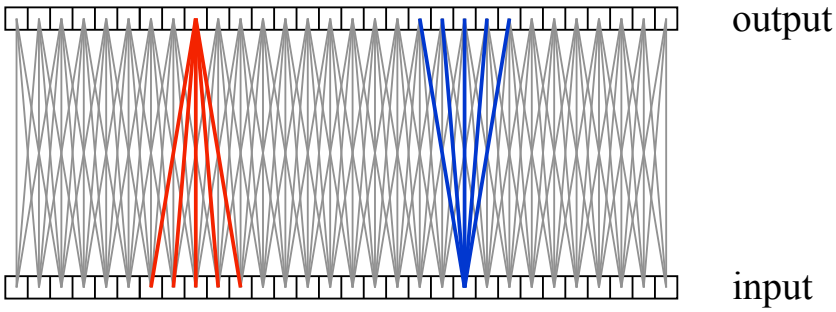


Figure 5: A one-dimensional view may more easily clarify the connection between the two different views of convolution introduced in the text. Convolution can be specified by a network of connections (gray) that connect input pixels to output pixels. Two pixels are connected by a gray edge if one contributes a value to the other. This is a symmetric view of convolution. An asymmetric view groups connections by specifying what output pixels are affected by each input pixel (blue). In another asymmetric view, connections may be grouped by specifying what input pixels contribute to each output pixel (red). It is the same operation viewed in three different ways. The blue view follows physics for image blur, and the view in red lends itself best to mathematical expression, because it reflects the definition of function. Similar considerations hold for correlation, which is however a different operation (equation (3) instead of equation (5)): Correlation with $T(u, v)$ is convolution with $H(u, v) = T(-u, -v)$.

with "image" and "point-spread function" playing interchangeable roles.[3]

In summary, and introducing the symbol '$*$' for convolution:

The *convolution* of an image I with the point-spread function H is defined as follows:

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

A *delta image* centered at a, b is an image that has a pixel value of 1 at (a, b) and zero everywhere else:

$$\delta_{a,b}(u,v) = \begin{cases} 1 & \text{for } u = a \text{ and } v = b \\ 0 & \text{elsewhere} \end{cases} \ .$$

Substitution of $\delta_{a,b}(u,v)$ for $I(u,v)$ in the second form of equation (7) shows immediately that the convolution of a delta image with a point-spread function H returns H itself, centered at (a, b):

$$[\delta_{a,b} * H](r,c) = H(r-a, c-b) \ . \tag{8}$$

This equation explains the meaning of the term "point-spread function:" the point in the delta image is spread into a blur equal to H. Equation (7) then shows that the output image J is obtained by the superposition (sum) of one such blur, centered at (u, v) and scaled by the value $I(u, v)$, for each of the pixels in the input image.

The result in equation (8) is particularly simple when image domains are thought of as infinite, and when $a = b = 0$. We then define

$$\delta(u,v) = \delta_{0,0}(u,v)$$

and obtain

$$\delta * H = H \ .$$

The point-spread function H in the definition of convolution (or, sometimes, the convolution operation itself) is said to be *shift-invariant* or *space-invariant* because the entries in H do not depend on the position (r, c) in the output image. In the case of image blur caused by poor lens focusing, invariance is only an approximation. For instance, a shallow depth of field leads to different amounts of blur in different parts of

[3]The situation is less symmetric for correlation: The changes of variables $u \leftarrow r + u$ and $v \leftarrow c + v$ yield the following expression equivalent to (3) (we again use infinite summation bounds for simplicity):

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

so correlating T with I yields an image $J(-r, -c)$ that is the upside-down and left-to-right flipped version of $J(r, c)$. So convolution commutes but correlation does not—another advantage of tolerating the two minus signs of convolution.

the image. In this case, one needs to consider a point-spread function of the form $H(u, v, r, c)$.

The *convolution* of an image I with the *space-varying* point-spread function H is defined as follows:

$$\begin{aligned} J(r,c) &= [I * H](r,c) = [H * I](r,c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(u,v,r,c) I(r-u, c-v) \\ &= \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(r-u, c-v, r, c) I(u,v) \ . \end{aligned} \tag{9}$$

Practical Aspects: Image Boundaries. The convolution neighborhood becomes undefined at pixel positions that are very close to the image boundaries. Typical solutions include the following:

- Consider images and point-spread functions to extend with zeros where they are not defined, and then only output the nonzero part of the result. This yields an output image J that is larger than the input image I. For instance, convolution of an $m \times n$ image with a $k \times l$ point-spread function yields an image of size $(m + k - 1) \times (n + l - 1)$.
- Define the output image J to be smaller than the input I, so that pixels that are too close to the image boundaries are not computed. For instance, convolution of an $m \times n$ image with a $k \times l$ point-spread function yields an image of size $(m - k + 1) \times (n - l + 1)$. This 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. This solution 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, sharp discontinuity all around it. This causes problems for certain types of point-spread functions. For the blur function, the output image merely darkens at the edges, because of all the zeros that enter the calculation. If the point-spread function is designed so that convolution with it computes image derivatives (a situation described later on), the discontinuities around the rim yield very large values in the output image.
- Pad the input image with replicas of the boundary pixels. 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, and also provides various types of padding.

Figure 6 shows an example of shift-variant convolution: for a coarse grid (r_i, c_j) of pixel positions, the diameter $d(r_i, c_j)$ of the pillbox function H that best approximates the blur between the image I in (a) and J in Figure 6(b) is found by trying all integer diameters and picking the one for which

$$\sum_{(r,c)\in W(r_i,c_j)} D^2(r,c) \quad \text{where} \quad D(r,c) = J(r,c) - [I * H](r,c)$$

is as small as possible for a fixed-size window $W(r_i, c_j)$ centered at (r_i, c_j). These diameter values are shown in Figure 6(c).

Each of the three color bands in the image in Figure 6(a) is then padded with replicas of its boundary values, as explained earlier, and filtered with a space-varying pillbox point-spread function, whose diameter $d(r, c)$ at pixel (r, c) is defined by *bilinear interpolation* of the surrounding coarse-grid values as follows. Let

$$u = \arg\max_i \{r_i \ : \ r_i \le r\} \quad \text{and} \quad v = \arg\max_j \{c_j \ : \ c_j \le c\}$$

be the indices of the coarse-grid point just to the left of and above position (r, c). Then

$$\begin{aligned} d(r,c) \ = \ & \text{round}\left(d(r_u, c_v)\, \frac{r_{u+1} - r}{r_{u+1} - r_u}\, \frac{c_{v+1} - c}{c_{v+1} - c_v} \right. \\ & + \ d(r_{u+1}, c_v)\, \frac{r - r_u}{r_{u+1} - r_u}\, \frac{c_{v+1} - c}{c_{v+1} - c_v} \\ & + \ d(r_u, c_{v+1})\, \frac{r_{u+1} - r}{r_{u+1} - r_u}\, \frac{c - c_v}{c_{v+1} - c_v} \\ & \left. + \ d(r_{u+1}, c_{v+1})\, \frac{r - r_u}{r_{u+1} - r_u}\, \frac{c - c_v}{c_{v+1} - c_v} \right) . \end{aligned}$$

The space-varying convolution is performed by literal implementation of equation (9), and is therefore very slow.

3 Filters

The lens blur model is an example of shift-varying convolution. Shift-invariant convolutions are also pervasive in image processing, where they are used for different purposes, including the reduction of the effects of image noise and image differentiation.

The effects of noise on images can be reduced by smoothing, that is, by replacing every pixel by a weighted average of its neighbors. The reason for this can be understood by thinking of an image patch that is small enough that the image intensity function I is well approximated by its tangent plane at the center (r, c) of the patch. Then, 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.

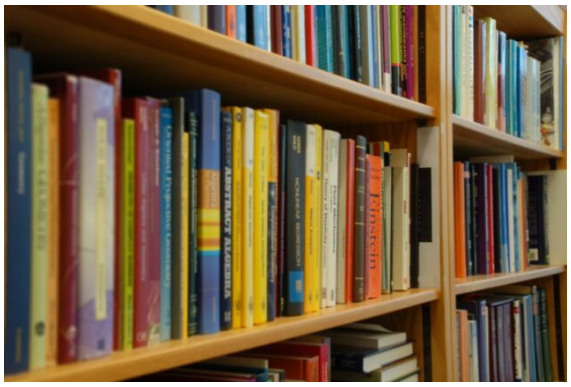
For independent noise values, noise reduction is proportional to the square root of the number of pixels in the smoothing window, so a large window is preferable. However, the assumption that the image intensity is approximately linear fails more and more as the window size increases, and is violated particularly badly

(a)

(b)

20	20	15	11	9	7	7	5	5	5	2	4
20	18	14	10	18	8	6	5	4	4	3	5
20	20	17	12	9	9	8	7	6	5	7	6
20	20	19	11	8	9	7	7	5	3	5	7
20	20	20	10	8	8	6	7	5	4	7	7
20	20	19	12	8	9	6	5	4	3	7	7
20	20	16	11	7	9	7	9	6	5	6	7
20	20	14	10	10	10	8	5	7	8	6	7

(c)



(d)

Figure 6: (a) An image taken with a narrow aperture, resulting in a great depth of field. (b) The same image taken with a wider aperture, resulting in a shallow depth of field and depth-dependent blur. (c) Values of the diameter of the pillbox point-spread function that best approximates the transformation from (a) to (b) in windows centered at a coarse grid on the image. The grid points are 300 rows and columns apart in a 2592×3872 image. (d) The image in (a) filtered with a space-varying pillbox function, whose diameters are computed by bilinear interpolation from the surrounding coarse-grid diameter values.

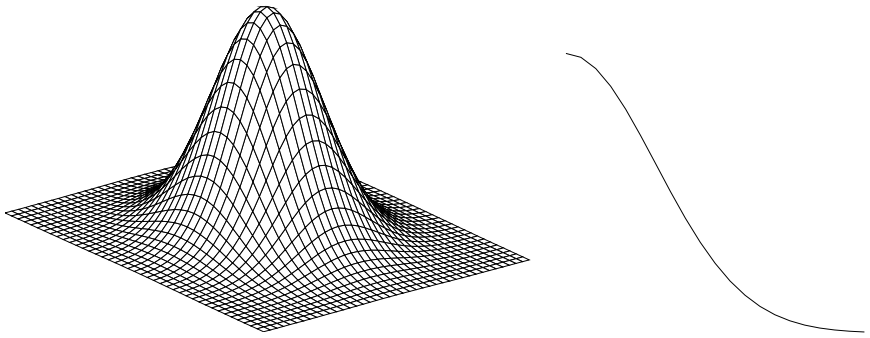


Figure 7: The two dimensional kernel on the left can be obtained by rotating the function $\gamma(r)$ on the right around a vertical axis through the maximum of the curve ($r = 0$).

along edges, where the image intensity function is nearly discontinuous, as shown in Figure 9. Thus, when smoothing, a compromise must be reached between noise reduction and image blurring.

The pillbox function is an example of a point-spread function that could be used for smoothing. The *kernel* (a shorter name for the point-spread function) 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[4]:

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

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 7).

The plot in figure 7 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 7), so that

$$G(v, u) = e^{-\frac{1}{2}\frac{u^2+v^2}{\sigma^2}} \ . \tag{10}$$

The greater σ is, the more smoothing occurs.

In the following, we first justify the choice of the Gaussian, by far the most popular smoothing function in computer vision, and then give an appropriate 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.

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

[4]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.

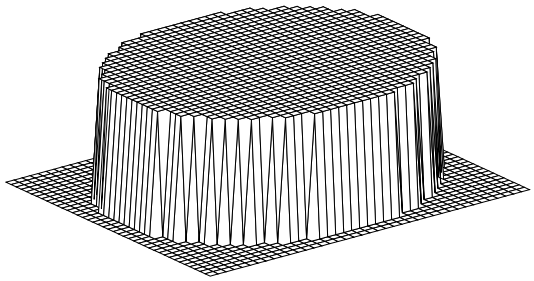


Figure 8: The pillbox function.

reduces smearing (blurring) while still maintaining noise averaging properties. To see this, compare a truncated Gaussian with a given support to a pillbox function over the same support (figure 8) 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 9 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 Gaussian ramp is narrower than the pillbox ramp, thereby producing a sharper image.

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.[5]

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.

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

$$g(x) = e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \tag{11}$$

is the one-dimensional (unnormalized) Gaussian.

Thus, the Gaussian of equation (10) separates into two equal factors. This 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 $[-h, h]^2$. With a separable kernel, the convolution (7) can then itself be separated into two one-dimensional convolutions:

$$J(r, c) = \sum_{u=-h}^{h} g(u) \sum_{v=-h}^{h} g(v)I(r - u, c - v) \ , \tag{12}$$

[5]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.

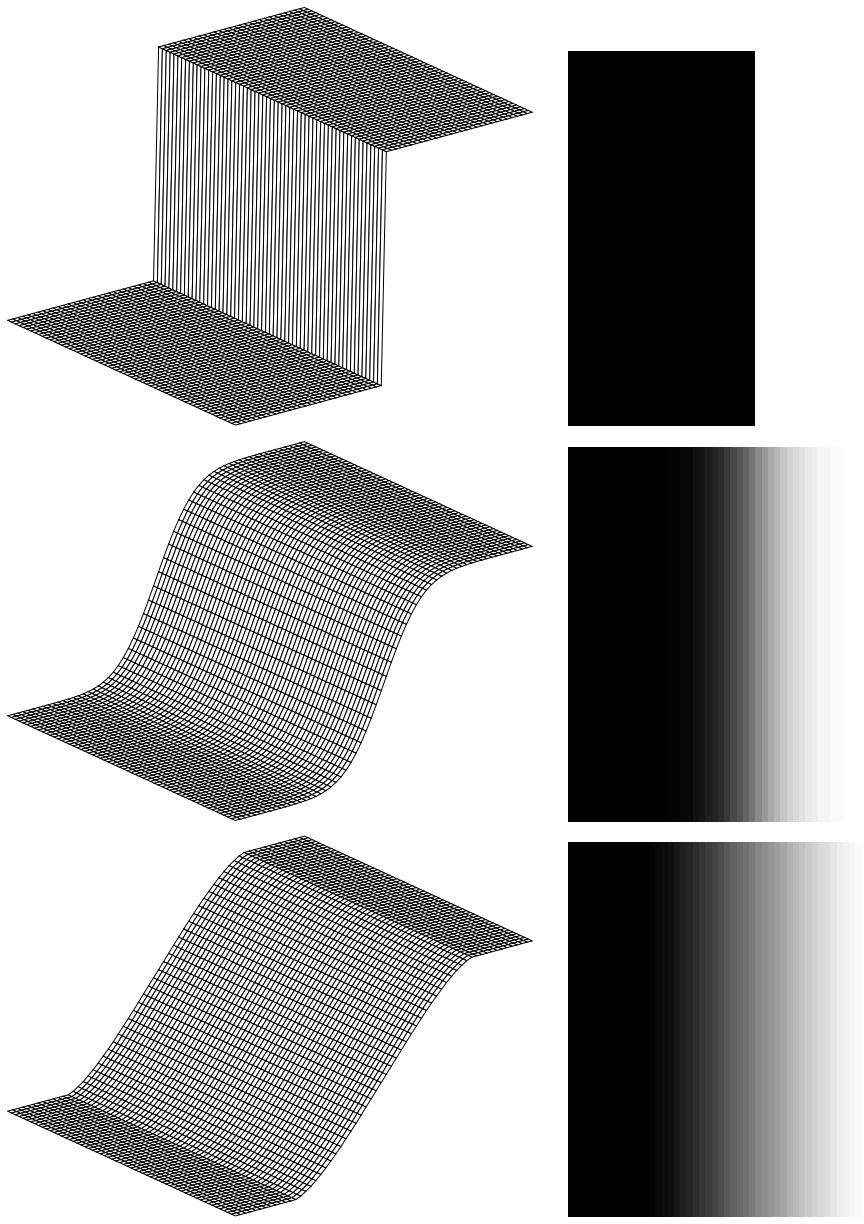


Figure 9: Intensity graphs (left) and images (right) of a vertical step function (top), and of the same step function smoothed with a Gaussian (middle), and with a pillbox function (bottom). Gaussian and pillbox have the same support and the same integral.

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

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

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 (12), on the other hand, can be rewritten so as to be computed by $2m$ multiplications and $2(m-1)$ additions as follows:

$$J(r,c) = \sum_{u=-h}^{h} g(u)\, \phi(r-u, c) \tag{13}$$

where

$$\phi(r,c) = \sum_{v=-h}^{h} g(v) I(r, c-v)\ . \tag{14}$$

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 (14) and (13): 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 (12) 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 (12) is computed for every pixel (r,c).

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

Practical Aspects: Truncation and Normalization. The Gaussian functions in this section were defined 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 $G(v,u)$ to be normalized, but rather its sum, since we define $G(v,u)$ 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 u, v 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 (10), and then normalize it by sum of the samples:

$$\begin{aligned} \tilde{G}(v,u) &= e^{-\frac{1}{2}\frac{u^2+v^2}{\sigma^2}} \\ c_{\tilde{G}} &= \sum_{i=-h}^{h} \sum_{j=-h}^{h} \tilde{G}(j,i) \\ G(v,u) &= \frac{1}{c_{\tilde{G}}} \tilde{G}(v,u)\ . \end{aligned} \tag{15}$$

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:

$$\begin{aligned} \tilde{g}(u) &= e^{-\frac{1}{2}\left(\frac{u}{\sigma}\right)^2} \\ c_{\tilde{g}} &= \frac{1}{\sum_{v=-h}^{h} \tilde{g}(v)} \\ g(u) &= c_{\tilde{g}} \tilde{g}(u) \ . \end{aligned} \tag{16}$$

Python Code for Denticle Detection

```
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
```