2. Image Processing

[Last Modified on August 21, 2012]
2 Image Processing

2.1 Introduction

Computer vision operates on images that come in the form of arrays of pixel values. When these images are transformed to obtain another image (as opposed to some other data structure), we say that image processing (as opposed to computer vision) is performed. This Section discusses some essential image processing operations.

The pixel values in the input image are invariably affected by noise, and it is often useful to clean the images somewhat with what is called a filter before they are further processed. Depending on the type of noise, a linear or nonlinear filter may be more appropriate. These filters may merely suppress fine detail so as to remove as much noise and as little useful signal as possible. Alternatively, image enhancement filters may attempt to undo part of an undesired image transformation, such as the blur resulting from poor lens focusing or from relative motion between the camera and the scene.

Applications of filtering to computer vision are not limited to image cleanup or enhancement. Other uses include taking derivatives of image intensity (a concept that strictly speaking is defined only for the continuous distribution of intensities on the imaging sensor) and detecting image structure (edges, periodic patterns, regions).

This Section discusses these concepts at a level that is sufficient for a good amount of work in computer vision. A more formal and thorough treatment can be found in classic books on image processing [2]. In addition, many topics of image processing are not treated at all in this Section. These include point processing (for instance, gamma correction, histogram equalization, or contrast enhancement); color processing (which requires substantial amount of material from psychophysics for a proper treatment); image morphology, compression, and compositing.

2.2 Images

Mathematically, a digital image is a mapping from a rectangle of integers $D = \{ r : r_s \leq r \leq r_e \} \times \{ c : c_s \leq c \leq c_e \} \subset \mathbb{Z}^2$ into a range $R \subset \mathbb{N}^k$ for some value of $k$.

More intuitively, a digital image is often an array of integers ($k = 1$), or of triples of integers ($k = 3$). For instance, full-color images (Figure 2.1(a)) are arrays of $(R, G, B)$ triples of integers, typically (but not always) between 0 and 255, that specify the intensity of the red, green, and blue components of the image. An uncompressed, full-color image that has $m$ rows and $n$ columns of pixels takes $3mn$ numbers (bytes) to specify.

In some applications it is useful to compress such an image based on the assumption that only some of the $256^3$ possible colors actually appear in the image. One then creates a color map, that is, a list of the unique colors triples $(R_i, G_i, B_i)$ that occur in the image, and then stores the color map together with one index $i$ into it for each pixel in the image. So instead of $3mn$ numbers, this scheme requires storing only $mn + 3c$ numbers, where $c$ is the number of triples in the color map.\footnote{In any case, no more than $mn$, but hopefully many fewer.} \footnote{However, unless the image has at most 256 rows and columns, the indices take more than one byte each.}
2.2

(a) Full Color

\[ k = 3 ; \quad \mathcal{R} = \{(R, G, B) : 0 \leq R, G, B \leq 255\} \]

(b) Color Mapped (100 colors)

\[ k = 3 ; \quad \mathcal{R} = \{(R_1, G_1, B_1), \ldots, (R_{100}, G_{100}, B_{100}) : 0 \leq R_i, G_i, B_i \leq 255\} \]

(c) Color Mapped (10 colors)

\[ k = 3 ; \quad \mathcal{R} = \{(R_1, G_1, B_1), \ldots, (R_{10}, G_{10}, B_{10}) : 0 \leq R_i, G_i, B_i \leq 255\} \]

(d) Gray

\[ k = 1 ; \quad \mathcal{R} = \{L : 0 \leq L \leq 255\} \]

(e) Half-Toned Binary

\[ k = 1 ; \quad \mathcal{R} = \{0, 1\} \]

(e) Thresholded Binary

\[ k = 1 ; \quad \mathcal{R} = \{0, 1\} \]

Figure 2.1: Images with different ranges.

The original image is from http://buytaert.net/album/miscellaneous-2006/. Reproduced under the Creative Commons Attribution - Non Commercial - Share Alike 2.5 License, http://creativecommons.org/licenses/by-nc-sa/2.5/.
Additional compression can be obtained if groups of similar colors are approximated by single colors. For instance, the image in Figure 2.1(a) has 28952 colors, but a reasonable approximation of this image can be obtained with only 100 distinct colors, as shown in the color-mapped image in Figure 2.1(b). The 100 colors were obtained by a clustering algorithm, which finds the 100 groups of colors in the original image that can best be approximated by one color per group. Different definitions of “best approximation” yield different clustering algorithms. Figure 2.1(c) shows a more aggressive color map compression with only ten colors.

A color image, color mapped or not, can be converted into a gray image by recording only the luminance

\[
L = 0.30R + 0.59G + 0.11B
\]

of each pixel. The three coefficients in this formula approximate the average response of the human eye to the three colors. Figure 2.1(d) shows the gray version of the color image in Figure 2.1(a).

Extreme compression can be achieved by storing a single bit (0 for black and 1 for white) for each pixel, as done in Figure 2.1(e). Half-toned images of this type are obtained by a class of techniques called halftoning or dithering, which produce the impression of gray by properly placed clusters of black dots in white areas. This compression method used to be useful for two-tone printing devices or displays, and can still be found in some newspapers that are printed with devices that can only either paint or not paint a black dot at any position on a white page.

More generally, a binary image is any image with only two levels of brightness. For instance, the image in Figure 2.1(f) was obtained by thresholding the gray image in Figure 2.1(d) with the value 100: pixels that are at least as bright as this value are transformed to white, and the others to black.

Regardless of range, we denote the pixel value (an integer or a triple) at row \( r \) and column \( c \) by \( I(r,c) \) (or other symbols instead of \( I \)) when we want to emphasize the array nature of an image. When graphics-style conventions are more useful, we use \( I(x_i,y_i) \) and express image position in the image reference system introduced in the Section on image formation. In discussions of reconstruction methods, writing coordinates in the camera reference system is more natural, so we use \( I(x,y) \) instead. For video, we add a third coordinate, typically \( t \) for “time” or \( f \) for “frame.”

### 2.3 Linear Processing in the Space Domain

An image transformation \( \mathcal{L} \) is linear if it satisfies superposition:

\[
\mathcal{L}(a_1 I_1 + a_2 I_2) = a_1 \mathcal{L}(I_1) + a_2 \mathcal{L}(I_2)
\]

Then and only then there exists a four-dimensional array \( L(i, j, r, c) \) such that for every input image \( I \) the output image \( J \) is expressed by the following summation:

\[
\mathcal{L}(I)(r, c) = \sum_{(i,j) \in D} L(i, j, r, c) I(i, j)
\]

where \( D \) is the domain of \( I \). This is just like matrix multiplication, but with twice the indices: every pixel \( (r, c) \) of the output is a linear combination of all of the pixels \( (i, j) \) of the input, and the combination for each output pixel can in principle have a different set of coefficients.
2.4

Even more explicitly, equation (1) can be written in matrix form if the input and output images $I$ and $J$ are transformed into vectors $i$ and $j$ that list all the pixels in a fixed order. For instance, if the image $I$ has $m$ rows and $n$ columns, then the transformation

$$i(entry(i, j)) = I(i, j) \quad \text{where} \quad entry(i, j) = m(j - 1) + i \quad \text{for} \quad 1 \leq i \leq m \quad \text{and} \quad 1 \leq j \leq n$$

lists the pixels in $I$ in column-major order. Equivalently,

$$i(k) = I(row(k), col(k)) \quad \text{where} \quad row(k) = ((k - 1) \mod m) + 1 \quad \text{and} \quad col(k) = \lceil k/m \rceil$$

where $\lceil \cdot \rceil$ is the “ceil” operator.

The block $L$ of coefficients in equation (1) can then be transformed accordingly into a matrix $L$ where

$$L(entry(i, j), entry(r, c)) = L(i, j, r, c).$$

The matrix $L$ is square, with $mn$ rows and $mn$ columns. Its huge size reflects the great generality of the definition of a linear transformation.

**Practical Aspects: Indices in Matlab and C.** In Matlab, transformation into a vector in column-major order is conveniently achieved by the instruction

$$i = I(:);$$

The reverse operation is

$$I = \text{reshape}(i, m, n);$$

In a language like C, arrays and vectors are typically indexed starting at 0 rather than 1, and row-major order is preferred:

$$i(entry_C(i, j)) = I(i, j) \quad \text{where} \quad entry_C(i, j) = ni + j \quad \text{for} \quad 0 \leq i < m \quad \text{and} \quad 0 \leq j < n.$$  

In this case,  

$$col_C(k) = k \mod n \quad \text{and} \quad row_C(k) = \lceil (k + 1)/n \rceil.$$  

### 2.3.1 Linearity, Saturation, and Quantization

With this definition, only trivial image transformations are linear: since the range of pixel values is finite, adding two images, or multiplying an image by a constant, often produces values out of range. This saturation problem must be taken into account for nearly every image processing algorithm. However, the advantages of linear transformations are so great that we temporarily redefine the range of images to be subsets of the real numbers. Conceptually, we proceed as illustrated in Figure 2.2: we first transform each image $I$ into its “real-range” version $\mathcal{R}(I)$: $I$ and $\mathcal{R}(I)$ are the same image, but the values of the latter are thought of as real numbers, rather than integers in a finite set. We then do image processing (the transformation $\mathcal{P}$ in the Figure), and
finally revert to the original range through a *quantization* operator $Q$. Quantization restores the range to a finite set, and is more substantial than $R$:

$$Q(I(r,c)) = \begin{cases} 
0 & \text{for } I(r,c) \leq 1/2 \\
 i & \text{for } i - 1/2 < I(r,c) \leq i + 1/2 \text{ and } i = 1, \ldots, 254 \\
255 & \text{for } I(r,c) > 254.5 
\end{cases}$$

This is a *point operation*, in that the result at pixel $(r, c)$ depends only on the input at that pixel. Quantization is of course a nonlinear operation, since superposition does not hold.

This is a point operation, in that the result at pixel $(r, c)$ depends only on the input at that pixel. Quantization is of course a nonlinear operation, since superposition does not hold.

![Figure 2.2: For linear processing, images are transformed through a “conceptual” point operation $R$ that regards image values as real numbers. After processing $P$ occurs, the values of the resulting image are quantized through point operator $Q$.](image)

**Practical Aspects: Casts.** When programming, the operator $R$ is often implemented as a type cast operation that converts integers to floating-point numbers. For instance, in Matlab we would write something like the following:

```matlab
img = double(img);
```

Conveniently, the Matlab `uint8` operator (for “unsigned integer with 8 bits”) performs quantization $Q$:

```matlab
img = uint8(img);
```

### 2.3.2 Convolution

The size of the matrix $L$ for a general, linear image transformation makes it essentially impossible to use in practice. Fortunately, many transformations of great practical interest correspond to a matrix $L$ with a great deal of structure.

The most important case is *convolution*, which adds structure to a general linear transformation in two ways. First, the value of a pixel at position $(r, c)$ in the output image $J$ is a linear combination of the pixel values in a relatively small neighborhood of position $(r, c)$ in the input image $I$, rather than of all the pixel values in $I$. Second, the coefficients of the linear combination are the same for all output pixel positions $(r, c)$.

The first property of convolution implies that $L(i, j, r, c)$ and its flattened form $L(\text{entry}(i, j), \text{entry}(r, c))$ have only a small number of nonzero entries. The second property implies that there
is a small matrix $H$ such that $L(i, j, r, c) = H(r - i, c - j)$. Correspondingly, the nonzero entries in $L$ take on the same, small set of values in every row of $L$, albeit in different positions.  

An example of convolution is lens blur, discussed in the Section on image formation. 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 2.3 (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 \textit{neighborhood} of the pixel at its center.

---

\textsuperscript{3}It is a useful exercise to work out the structure of $L$ for a small case. The resulting matrix structure is called “block-Toeplitz with Toeplitz blocks.”
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 2.3(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 $5 \times 5$ 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 the rows and columns of $H$ with integers between $-2$ and 2:

$$H = \frac{1}{21} \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

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:

```plaintext
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 2.3(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:

\[
\begin{align*}
\text{for } r = 1:m \\
\text{for } c = 1:n \\
\quad J(r, c) &= 0 \\
\quad \text{for } u = -2:2 \\
\quad &\quad \text{for } v = -2:2 \\
\quad &\quad \quad J(r, c) = J(r, c) + H(u, v) \times I(r-u, c-v) \\
\quad &\quad \text{end} \\
\quad \text{end} \\
\text{end} \\
\end{align*}
\]

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:

\[
J(r, c) = \sum_{u=-2}^{2} \sum_{v=-2}^{2} H(u, v) I(r-u, c-v). 
\] (2)

In contrast, a direct mathematical translation from the earlier piece of code is not obvious. The operation described by equation (2) is called \textit{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 2.3(a)) that overlaps the corresponding pixel. Simple but lengthy mathematics, or numerical approximation, yields the following values for a \(5 \times 5\) approximation \(H\) to the pillbox function:

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

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.

Note the minus signs in equation (2). 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 2.4.

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:

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

The changes of variables $u \leftarrow r - u$ and $v \leftarrow c - v$ then entail the equivalence of equation (3) with the following expression:

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

with “image” and “point-spread function” playing interchangeable roles.

In summary, and introducing the symbol ‘∗’ for convolution:

---

4It would not be wise to do so also in the equivalent program!
The convolution of an image $I$ with the point-spread function $H$ is defined as follows:

$$J(r, c) = [I * H](r, c) = [H * I](r, c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(u, v)I(r - u, c - v)$$

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

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 (5) shows immediately that the convolution of a delta image with a point-spread function $H$ returns $H$ itself, centered at $(a, b)$:

$$[\delta_{a,b} * H](r, c) = H(r - a, c - b) .$$ (6)

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 (5) 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 (6) 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. As seen in the Section on image formation, a shallow depth of field leads to different amounts of blur in different parts of the image. In this case, one needs to consider a point-spread function of the form $H(u, v, r, c)$. This is still simpler than the general linear transformation in equation (1), because the number of nonzero entries for each value of $r$ and $c$ in $H$ is equal to the neighborhood size, rather than the image size.
The convolution of an image \( I \) with the \textit{space-varying} point-spread function \( H \) is defined as follows:

\[
J(r, c) = [I * H](r, c) = [H * I](r, c) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(u, v, r, c)I(r - u, c - v)
\]

\[
= \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} H(r - u, c - v, r, c)I(u, v). \quad (7)
\]

**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 sufficiently wide rim of zero-valued pixels. This solution is simple to implement (although not as simple as the previous one), and preserves image size. 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 \( 4 \times 5 \) image would look as follows:

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

This is a relatively simple solution, avoids spurious discontinuities, and preserves image size.
The Matlab convolution function \texttt{conv2} provides options ‘full’, ‘valid’, and ‘same’ to implement the first three alternatives above, in that order.

Figure 2.5 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 2.5(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 2.5(c).

Each of the three color bands in the image in Figure 2.5(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

\[
u = \arg \max_i \{ r_i : r_i \leq r \} \quad \text{and} \quad v = \arg \max_j \{ c_j : c_j \leq c \}
\]

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

\[
d(r,c) = \text{round} \left( d(r_u,c_v) \frac{r_{u+1} - r}{r_{u+1} - r_u} \frac{c_{v+1} - c}{c_{v+1} - c_v} \right) + d(r_{u+1},c_v) \frac{r}{r_{u+1} - r_u} \frac{c_{v+1} - c}{c_{v+1} - c_v} + d(r_u,c_{v+1}) \frac{r}{r_{u+1} - r_u} \frac{c_{v+1} - c}{c_{v+1} - c_v} + d(r_{u+1},c_{v+1}) \frac{r}{r_{u+1} - r_u} \frac{c}{c_{v+1} - c_v} \right)
\]

The space-varying convolution is performed by literal implementation of equation (7), and is therefore very slow.

\section*{2.4 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. We discuss linear noise filters in this Section, and differentiation in the next.

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
Figure 2.5: (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 $2592 \times 3872$ 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 2.6: 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 ($r = 0$).

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 averaging 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 along edges, where the image intensity function is nearly discontinuous, as shown in Figure 2.8. 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$^5$:

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

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 2.6).

The plot in figure 2.6 was obtained from the (unnormalized) Gaussian function

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

$^5$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.
with \( \sigma = 6 \) pixels (one pixel corresponds to one cell of the mesh in figure 2.6), so that

\[
G(v, u) = e^{-\frac{1}{2} \frac{u^2 + v^2}{\sigma^2}}.
\] (8)

The greater \( \sigma \) 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 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 2.7) 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 2.8 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.6

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.

\textbf{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

---

6This 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 2.8: 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.
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

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

which leads to the equality

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

where

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

is the one-dimensional (unnormalized) Gaussian.

Thus, the Gaussian of equation (8) 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 $[-n, n]^2$. With a separable kernel, the convolution (5) can then itself be separated into two one-dimensional convolutions:

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

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

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

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 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:

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

where

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

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 $g$ in the horizontal direction, then convolve the resulting image with $g$ 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 $(r, u) = (d-n, -n), (d-n+1, -n+1), \ldots, (d+n, n)$ when equation (10) 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 (8), and then normalize it by sum of the samples:

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

$$c = \sum_{i=-n}^{n} \sum_{j=-n}^{n} G(j, i)$$

$$\tilde{G}(v, u) = \frac{1}{c} G(v, u).$$

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

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

$$= I_0 \sum_{u=-n}^{n} \sum_{v=-n}^{n} \tilde{G}(v, u)$$

$$= I_0$$

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:

$$g(u) = e^{-\frac{1}{2} (\frac{u}{\sigma})^2}$$

$$k_g = \frac{1}{\sum_{v=-n}^{n} g(v)}$$

$$\tilde{g}(u) = k_g g(u).$$

2.4.1 Image Differentiation

Since images are defined on discrete domains, “image differentiation” is undefined. To give this notion some meaning, we think of images as sampled versions of continuous distributions of

---

7“Continuous” here refers to the domain: we are talking about functions of real valued variables.
brightness. This indeed they are: the distribution of intensities on the sensor is continuous, and we saw in the Section on image formation that the sensor integrates this distribution over the active area of each pixel and then samples the result at the pixel locations.

“Differentiating an image” means to compute the samples of the derivative of the continuous distribution of brightness values on the sensor surface.

Thus, differentiation involves, at least conceptually, undoing sampling (that is, computing a continuous image from a discrete one), differentiating, and sampling again. The process of undoing sampling is called interpolation. Figure 2.9 shows a conceptual bloc diagram for the computation of the image derivative in the $x$ (or $c$) direction.

Figure 2.9: Conceptual bloc diagram for the computation of the derivative of image $I(r, c)$ in the horizontal ($c$) direction. The first block interpolates from a discrete to a continuous domain. The second computes the partial derivative in the horizontal ($x$) direction. The third block samples from a continuous domain back to a discrete one.

Interpolation can be expressed as a hybrid-domain convolution, that is a convolution of a discrete image with a continuous kernel. This is formally analogous to a discrete convolution, but has a very different meaning:  

$$C(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) P(x - j, y - i)$$

where $x, y$ are real variables and $P(x, y)$ is called the interpolation kernel.

This hybrid convolution seems hard to implement: how can we even represent the output, a function of two real variables, on a computer? However, the chain of the three operations depicted in Figure 2.9 goes from discrete-domain to discrete-domain. As we now show, this chain can be implemented easily and without reference to continuous-domain variables.

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.

---

8For simplicity, we assume that the $x$ and $y$ axes have the same origin and direction as the $j$ and $i$ axes: to the right and down, respectively.
2.20

Formally, 

\[
D(x, y) = \frac{\partial C}{\partial x}(x, y) = \frac{\partial}{\partial x} \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) P(x - j, y - i)
\]

\[
= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) P_x(x - j, y - i)
\]

where

\[P_x(x, y) = \frac{\partial P}{\partial x}(x, y)\]

is the partial derivative of \(P(x, y)\) with respect to \(x\).

Finally, we need to sample the result \(D(x, y)\) at the grid points \((r, c)\) to obtain a discrete image \(I_c(r, c)\). This yields the final, discrete convolution that computes the derivative of the underlying continuous image with respect to the horizontal variable:

\[
J(r, c) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) P_x(c - j, r - i) .
\]

Note that all continuous variables have disappeared from this expression: this is a standard, discrete-domain convolution, so we can implement this on a computer without difficulty.

The correct choice for the interpolation function \(P(x, y)\) is outside the scope of this course, but it turns out that the truncated Gaussian function is adequate, as it smooths the data while differentiating. We therefore let \(P\) be the (unnormalized) Gaussian function of two continuous variables \(x\) and \(y\):

\[P(x, y) = G(x, y)\]

and \(P_x, P_y\) its partial derivatives with respect to \(x\) and \(y\) (Figure 2.10). We then sample \(P_x\) and \(P_y\) over a suitable interval \([-n, n]\) of 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

\[u(i, j) = j\]

and we want to find a constant \(u_0\) such that

\[u_0 \sum_{i=-n}^{n} \sum_{j=-n}^{n} u(i, j) P_x(c - j, r - i) = 1 .\]

for all \(r, c\) so that

\[\tilde{P}_x(x, y) = u_0 P_x(x, y) \quad \text{and} \quad \tilde{P}_y(x, y) = u_0 P_y(x, y) .\]

\(^9\)Again, \(c\) and \(r\) are assumed to have the same origin and orientations as \(x\) and \(y\).
In particular for \( r = c = 0 \) we obtain

\[
I_0 = -\frac{1}{\sum_{i=-n}^{n} \sum_{j=-n}^{n} jG_x(j, i)}.
\]  

(15)

Since the partial derivative \( G_x(x, y) \) of the Gaussian function with respect to \( x \) is negative for positive \( x \), this constant \( u_0 \) is positive. By symmetry, the same constant normalizes \( G_y \).

Figure 2.10: 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 horizontally to the right, and the \( y \) variable points vertically down.

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

\[
I_c(r, c) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} I(i, j)G_x(c-j, r-i) = \sum_{j=-n}^{n} d(c-j) \sum_{i=-n}^{n} I(i, j)g(r-i)
\]

where

\[
d(x) = \frac{dg}{dx} = -\frac{x}{\sigma^2} g(x)
\]

is the ordinary derivative of the one-dimensional Gaussian function \( g(x) \) defined in equation (9). A similar expression holds for \( I_r(r, c) \) (see below).

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

\[
I_c(r, c) = \sum_{i=-n}^{n} g(r - i) \sum_{j=-n}^{n} I(i, j)d(c - j) = \sum_{j=-n}^{n} d(c - j) \sum_{i=-n}^{n} I(i, j)g(r - i)
\]

\[
I_r(r, c) = \sum_{i=-n}^{n} d(r - i) \sum_{j=-n}^{n} I(i, j)g(c - j) = \sum_{j=-n}^{n} g(c - j) \sum_{i=-n}^{n} I(i, j)d(r - i) .
\]

Normalization can also be done separately: the one-dimensional Gaussian \( g \) is normalized according to equation (14), and the one-dimensional Gaussian derivative \( d \) is normalized by the one-dimensional equivalent of equation (15):

\[
d(u) = -ue^{-\frac{1}{2}(\frac{u}{\sigma})^2}
\]

\[
k_d = \frac{1}{\sum_{v=-n}^{n} vd(v)}
\]

\[
\tilde{d}(u) = k_d d(u) .
\]

We can summarize this discussion as follows.

The “derivatives” \( I_c(r, c) \) and \( I_r(r, c) \) of an image \( I(r, c) \) in the horizontal (to the right) and vertical (down) direction, respectively, are approximately the samples of the derivative of the continuous distribution of brightness values on the sensor surface. The images \( I_c \) and \( I_r \) can be computed by the following convolutions:

\[
I_c(r, c) = \sum_{i=-n}^{n} \tilde{g}(r - i) \sum_{j=-n}^{n} I(i, j)\tilde{d}(c - j) = \sum_{j=-n}^{n} \tilde{d}(c - j) \sum_{i=-n}^{n} I(i, j)\tilde{g}(r - i)
\]

\[
I_r(r, c) = \sum_{i=-n}^{n} \tilde{d}(r - i) \sum_{j=-n}^{n} I(i, j)\tilde{g}(c - j) = \sum_{j=-n}^{n} \tilde{g}(c - j) \sum_{i=-n}^{n} I(i, j)\tilde{d}(r - i) .
\]

In these expressions,

\[
\tilde{g}(u) = k_g g(u) \quad \text{where} \quad g(u) = e^{-\frac{1}{2}(\frac{u}{\sigma})^2} \quad \text{and} \quad k_g = \frac{1}{\sum_{v=-n}^{n} g(v)}
\]

and

\[
\tilde{d}(u) = k_d d(u) \quad \text{where} \quad d(u) = -ue^{-\frac{1}{2}(\frac{u}{\sigma})^2} \quad \text{and} \quad k_d = \frac{1}{\sum_{v=-n}^{n} vd(v)} .
\]

The constant \( \sigma \) determines the amount of smoothing performed during differentiation: the greater \( \sigma \), the more smoothing occurs. The integer \( n \) is proportional to \( \sigma \), so that the effect of truncating the Gaussian function is independent of \( \sigma \).
2.5 **Nonlinear Filters**

Equation (1) shows that linear filters combine input pixels in a way that depends only on where a pixel is in the image, not on its value. In many situations in image processing, on the other hand, it is useful to take input pixel values in account as well before deciding how to use them in the output.

For instance, transmission of a digital image over a noisy channel may corrupt the values of some pixels while leaving others intact. The new value is often very different from the original value, because noise affects individual bits during transmission. If a high-order bit is changed, the corresponding pixel value can change dramatically. The resulting noise is called salt-and-pepper noise, because it results into a sprinkle of isolated pixels that are much lighter (salt) or much darker (pepper) than their neighbors. These anomalous values can be detected, and replaced by some combination of their neighbors. Such a filter would only process anomalous pixels, and is therefore nonlinear, because whether a pixel is anomalous or not depends on its value, ont its position in the image. The median filter does this, and is discussed in Section 2.5.1 below.

Another situation in which nonlinear filtering is useful is when noise (with Gaussian statistics or otherwise) is to be cleaned up without blurring image edges. The standard smoothing filter discussed in Section 2.4 cannot do this: since the coefficients of linear filters are blind to input values, smoothing smooths everything equally, including edges. It turns out that the median filter does well even in this situation. The bilateral filter discussed in Section 2.5.2 can do even better.

**Edge detection**, described in Section 2.6, is an example of even more strongly nonlinear image processing. The decision as to whether a pixel belongs to an edge or not is a binary function of the image and of pixel position, and is therefore nonlinear.

### 2.5.1 The Median Filter

The principle of operation of the median filter is straightforward: replace the value of each pixel in the input image with the median of the pixel values in a fixed-size window around the pixel. In contrast, a smoothing filter replaces the pixel with the mean (weighted by the filter point-spread function) of the values in the window.

Let \((r, c)\) be the image position for which a filter is computing the current value, and let \((i, j)\) be the position of a pixel within the given window around \((r, c)\). Then the contribution that pixel \((i, j)\) gives to the output value at \((r, c)\) for a smoothing filter is proportional to \(I(i, j)\), and equals \(H(r – i, c – j)I(i, j)\), where \(H\) is the point-spread function. Thus, the smoothing filter is linear.

In contrast, the contribution that pixel \((i, j)\) gives to the output value at \((r, c)\) for a median filter depends in a nonlinear way on the values of all the pixels in the window. In principle, the output value at \((r, c)\) can be computed by sorting all the values in the window, and picking the value in the middle of the sorted sequence as the output. If the number of elements in the window is even, the tie between the two values in the middle is usually broken by taking the mean of the two elements. In image processing, the resulting value is rounded, because the mean is not necessarily an integer while image pixel values usually are.
Filters as Statistical Estimators  The relative merits of mean and median can be understood by viewing a filter as a statistical estimator of the value of a pixel. Specifically, the pixel values in an image window (the support of the point-spread function) are a “population sample” that is used to estimate the value in the middle of the window under the assumption that the image intensity is an approximately linear function within the window.

The (arithmetic, i.e., un-weighted) mean is the most efficient estimator of a population, in the sense that it is the unbiased estimator that has smallest variance. This makes the mean a good smoother: if the estimate is repeated multiple times, the error committed is on the average as small as possible with any unbiased estimator. The weighting introduced by a non-constant point-spread function typically emphasizes pixels that are closer to the window center. This reduces the effects of the unwarranted assumption of linear image brightness within the window.

The median is less efficient, as its variance is $\pi/2 \approx 1.57$ greater than that of the mean for a large population sample (window).\(^\text{10}\) However, the median has an important advantage in terms of its breakdown point.

The breakdown point of a statistical estimator is the smallest fraction of population sample points that must be altered to make the estimate arbitrarily poor. For instance, by altering even just one value, the mean can be changed by an arbitrary amount: with $N$ members in the sample, a change of $x$ in the value of the mean can be achieved by changing the value of a single member by $Nx$. So the mean has a breakdown point of 0.

No statistical estimator can have a breakdown point greater than 0.5: if the sample values are drawn from a population $A$, but more than half of them are then replaced by drawing them from a different population $B$, no estimator can distinguish between a highly corrupted sample from population $A$ and a moderately corrupted sample from population $B$.

Interestingly, the median achieves the maximum attainable breakdown point of 0.5: Although changing even a single value does in general change the value of the median, it cannot do so by an arbitrary value. Instead, since the median is the middle value of the sorted list of values from the sample (window), to change the list so that its middle value, say, increases arbitrarily requires changing at least half of the values to be no less than the new, desired middle value. A similar reasoning holds if we want to decrease the median instead.

Removal of “Salt and Pepper” Noise  The high breakdown point of the median quantifies its relative insensitivity to outliers in a sample. For salt-and-pepper noise, for instance, consider the case in which a “grain of salt” (i.e., an anomalously bright pixel) is at position $(r, c)$ in the middle of the current median-filter window. Unless at least half of the pixels in the window are “salt” as well, that value will be very different from the median of the values within the window, and it will not influence the mean at all. The new, filtered pixel value at $(r, c)$ will therefore replace the bright pixel value with one that is more representative of the majority of the pixel values within the window. The “grain of salt” has been removed.

For the same reason the median filter is also a good edge preserving filter. To illustrate in an idealized situation, suppose that pixel $(r, c)$ is on the dark side of an edge between a bright and a

\(^{10}\)More precisely, for a window with $2n + 1$ pixels, the variance of the median is $\pi(2n + 1)/4n$ times greater than that of the mean.
dark region in an image, as illustrated in Figure 2.11. Since the center pixel \((r, c)\) is on the dark side of the edge, most of the pixels in the window are on the dark side as well. Because of this, the median of the values in the window is one of the dark pixel values. In particular, if the pixel at \((r, c)\) were a “grain” of salt-and-pepper noise, then its value would be replaced by another value from the dark region. Pixels on the light side of the edge cannot influence the median value by their values, but only by their number (which is in the minority). Because of this, the median filter preserves edges while it removes salt-and-pepper noise (and other types of noise as well). This is in contrast with the smoothing filter, which smooths noise and edges in equal measure.

Figure 2.11: Pixel \((r, c)\) is on the dark side of a dark-light edge in the image. The median-filter window is 5 \(\times\) 5 pixels.

Of course, the edge-preserving property of the median filter relies on the majority of the pixels in the window being on the same side of the edge as the central pixel. This holds for straight edges, but not necessarily for curved edges and corners.

**Exercise.** Convince yourself with a small, 5 \(\times\) 5 example similar to that in Figure 2.11 that a median filter eats away at corners.

Figure 2.12 shows the result of applying a median filter to an image corrupted by salt-and-pepper noise.

**Algorithms**  Median filtering with small windows (even 3 \(\times\) 3 or so) often yields good results. With increasing sensor resolutions, however, the amount of light that hits a single pixel decreases. While the size of the filter window necessary to achieve a certain degree of noise cleanup is likely to remain more or less constant per unit of solid angle, it is likely to increase, as a function of image resolution, when it is measured in pixels, and therefore in terms of computational complexity. Because of this, the asymptotic complexity of median filtering, which is irrelevant for small windows, is of some interest.
Figure 2.12: (a) An image corrupted by salt-and-pepper noise. (b) Detail in the red box in (a). (c) The image in (a) filtered with a median filter with a $5 \times 5$ pixel window. (d) Detail in the red box in (c). Note the more rounded corners than in (b).
Straightforward computer implementation of the definition of the median filter with an \( n \times n \) window costs \( O(n^2 \log n) \) for sorting. This cost, which is then multiplied by the number of pixels in the image, grows quite rapidly. However, the median (or any order statistics) of \( N \) numbers can be computed in \( O(N) \) by a selection algorithm [1], thereby bringing the complexity of the median down to \( O(n^2) \) per image pixel.

A different route for increased efficiency capitalizes on the fixed number of bits per pixel. A histogram of pixel values has a fixed number of bins, and can be computed in each window in \( O(n^2) \) time. The median can then be computed from the histogram in constant time, so median filtering takes \( O(n^2) \) again. In addition, Huang [6] noted that windows for neighboring pixels overlap over \( n^2 - 2n \) pixels, and calculation of the histogram can take advantage of this overlap: When sliding the filter window one pixel to the right, decrease the proper bin counts for the leftmost, exiting column of the window, and increase them for the rightmost, entering column. This reduces the complexity to \( O(n) \) per image pixel.

Weiss [12] devised an algorithm that is faster both asymptotically and practically by observing that (i) the window overlap exploited by Huang also occurs vertically and, perhaps more importantly, that (ii) histograms are additive over disjoint regions: if regions \( A \) and \( B \) do not share pixels, then \( h_{A \cup B} = h_A + h_B \), where \( h_X \) is the histogram of values in region \( X \). This allows building a hierarchical structure for the maintenance (in \( O(\log n) \) time and \( O(n) \) space) of multiple histograms over image columns, and to compute the required medians in constant time. This results in an \( O(\log n) \) (per image pixel) median filtering algorithm.

### 2.5.2 The Bilateral Filter

The bilateral filter [10, 11] offers a more flexible and explicit way to account for the values of the pixels in the filter window than the median filter does. In the bilateral filter, a closeness function \( H \) accounts for the spatial distance between the window’s central pixel \( (r, c) \) and a pixel \( (i, j) \) within the window. This is exactly what the point-spread function \( H \) does for a smoothing filter. In addition, the bilateral filter has a similarity function \( s \) that measures how similar the values of two pixels are. While the domain of \( H \) is the set of image position differences \( (r - i, c - j) \), the domain of \( s \) is the set of pixel value differences \( I(r, c) - I(i, j) \).

The bilateral filter combines closeness and similarity in a multiplicative fashion to determine the contribution of the pixel \( (i, j) \) to the value resulting at \( (r, c) \):

\[
J(r, c) = \frac{\sum_{(i,j) \in W(r,c)} I(i, j) H(r - i, c - j) s(I(r,c) - I(i,j))}{\sum_{(i,j) \in W(r,c)} H(r - i, c - j) s(I(r,c) - I(i,j))} \tag{16}
\]

where \( W(r, c) \) is the customary fixed-size window centered at \( (r, c) \). As usual, the denominator ensures proper normalization of the output: for a constant image, \( I(i, j) = I_0 \) for all \( (i, j) \), the output is unaltered: \( J(r, c) = I_0 \) for all \( (r, c) \). In a typical implementation, both closeness \( H \) and similarity \( s \) are truncated Gaussian functions with different spread parameters \( \sigma_H \) and \( \sigma_s \).

The main property of the bilateral filter is that it preserves edges while smoothing away image noise and smaller variations of image intensity. Edges are preserved because pixels \( (i, j) \) that are across an edge from the output pixel position \( (r, c) \) have a dissimilar intensity, and the similarity
term \( s(I(r, c) - I(i, j)) \) is correspondingly small. In contrast with the median filter, this effect depends on the difference in the values of the two pixels, rather than on the number of pixels on either side of the edge. As a consequence, the bilateral filter does not eat away at corners.

In addition, it can be shown formally [11] that the similarity term of the filter has the effect of narrowing the modes of the histogram of image intensities, that is, to reduce the number of distinct pixel values in an image. Informally, this effect is a consequence of the smoothing property of the bilateral filter for groups of mutually close pixels that have similar values.

However, the bilateral filter does not remove salt and pepper noise: when a “grain” of noise is at the output position \((r, c)\), the similarity term \( s(I(r, c) - I(i, j)) \) is very small for all other pixels, and the value \( I(r, c) \) is copied essentially unaltered in the output \( J(r, c) \).

Figure 2.13 illustrates the effects of the bilateral filter on an image with rich texture.

![Figure 2.13](image_url)

**Figure 2.13:** (a) Image of a kitten. (b) The image in (a) filtered with a bilateral filter with closeness spread \( \sigma_H = 3 \) pixels and similarity spread \( \sigma_s = 30 \) gray levels. Images are from [11].

All edges around the head of the kitten are faithfully preserved, because intensity differences across these edges are strong. This also holds for the whiskers, eyes, nose, and mouth, and for the glint in the kitten’s eyes. On the other hand, much of the texture of the kitten’s fur has been wiped out by the filter, because intensity differences between different points on the fur are smaller.

The similarity function \( s \) of the bilateral filter can be defined in different ways depending on the application. This allows applying the filter to images in which each pixel is a vector of values, rather than a single value. The main example for this is the filtering of color images. Inherently scalar filters such as the median filter (which requires a total ordering of the pixel values) are
sometimes applied to vector images by applying the filter to each vector component separately (for instance, to the red, green, blue color bands). This has often undesired effects along edges, because smoothing alters the balance of the color components, and therefore changes the hue of the colors. In contrast, the bilateral filter can use any desired color similarity $s$, including functions derived from a psychophysical study of how humans perceive color differences, thereby making sure that colors are handled correctly everywhere. [11]

The nonlinear nature of the bilateral filter disrupts separability (discussed earlier in Section 2.4). Direct implementation of the filter after its definition is therefore rather slow. However, the methods cited earlier to speed up the median filter are applicable to the bilateral filter as well, and result in fast implementations. [12]

### 2.6 Edge Detection

The goal of edge detection is to compute (possibly open) curves in the image that are characterized by a comparatively strong change of image intensity across them. Change of image intensity is usually measured by the magnitude of the image gradient, defined in Section 2.6.1. Different notions of “strength” lead to different edge detectors. The most widely used was developed by Canny in his master thesis[3] together with a general, formal framework for edge detection. Canny’s edge detector is described in Section 2.6.2. Section 2.6.3 discusses the notion of the scale at which an edge occurs, that is, the width of a band around the edge in which intensity changes from darker to brighter.

#### 2.6.1 Image Gradient

We saw in Section 2.4.1 how to compute the partial derivatives of image intensity $I$ anywhere in the image. In that Section, we had identified the $x$ axis with the $c$ (column) axis, and the $y$ axis with the $r$ (row) axis. To avoid confusion, now that we have settled the image processing aspects, we can revert to a more standard notation. Specifically, the $x$ and $y$ axes follow the “computer graphics” convention: $x$ is horizontal and to the right, while $y$ is vertical and up, with the origin of the reference frame at the center of the pixel at the bottom left corner of the image, which is now the point $(0, 0)$. If $R$ is the number of rows in the image, the new image function $I$ in terms of the old one, $I^{(old)}$, is given by the following change of variables:

$$I(x, y) = I^{(old)}(R - r, c - 1).$$

Note that the first argument $x$ of $I$ is in the horizontal direction, while the first argument of $I^{(old)}$ used to be in the vertical direction. Similarly, for the partial derivatives we have

$$I_x(x, y) = \frac{\partial I}{\partial x}(x, y) = I_x^{(old)}(R - r, c) \quad \text{and} \quad I_y(x, y) = \frac{\partial I}{\partial y}(x, y) = -I_y^{(old)}(R - r, c).$$

The minus sign before $I_y^{(old)}(R - r, c)$ is a consequence of the reversal in the direction of the vertical axis.

A value of $I_x$ and one of $I_y$ can be computed at every image position\(^\text{11}\) $(x, y)$, and these values

---

\(^{11}\) We assume to pad images by replication beyond their boundaries.
can be collected into two images. Two sample images are shown in Figure 2.14 (c) and (d).

A different view of a detail of these two images is shown in Figure 2.15 in the form of a quiver diagram. For each pixel \((x, y)\), this diagram shows a vector with components \(I_x(x, y)\) and \(I_y(x, y)\). The diagram is shown only for a detail of the eye in Figure 2.14 to make the vectors visible.

The two components \(I_x(x, y)\) and \(I_y(x, y)\) considered together as a vector at each pixel form the gradient of the image,

\[
g(x, y) = (I_x(x, y), I_y(x, y))^T .
\]

The gradient vector at pixel \((x, y)\) points in the direction of greatest change in \(I\), from darker to lighter. The magnitude

\[
g(x, y) = \|g(x, y)\| = \sqrt{I_x^2(x, y) + I_y^2(x, y)}
\]

of the gradient is the local amount of change in the direction of the gradient, measured in gray levels per pixel. The (total) derivative of image intensity \(I\) along a given direction with unit vector \(u\) is the rate of change of \(I\) in the direction of \(u = (u, v)\). This can be computed from the gradient by noticing that the coordinates \(p = (x, y)^T\) of a point on the line through \(p_0 = (x_0, y_0)^T\) and along \(u\) are

\[
p = p_0 + tu \quad \text{that is,} \quad x = x_0 + tu, \quad y = y_0 + tv ,
\]

so that the chain rule for differentiation yields

\[
\frac{dI}{dt} = \frac{\partial I}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial I}{\partial y} \frac{\partial y}{\partial t} = I_x u + I_y v .
\]

In other words, the derivative of \(I\) in the direction of the unit vector \(u\) is the projection of the gradient onto \(u\):

\[
\frac{dI}{dt} = g^T u . \quad (17)
\]

### 2.6.2 Canny’s Edge Detector

In a nutshell, Canny’s edge detector computes the magnitude of the gradient everywhere in the image, finds ridges of this function, and preserves ridges that have a high enough value. The magnitude of the gradient measures local change of image brightness in the direction of maximum change. Ridges of the magnitude \(g\) are points where \(g\) achieves a local maximum in the direction of \(g\), and are therefore inflection points of the image intensity function \(I\). High-valued ridges are likely to correspond to inflection points of the signal. In contrast, low-valued ridges have some likelihood of being caused by noise, and are therefore suppressed. This suppresses noise, but also edges with low contrast.

Ridge detection and thresholding are discussed in more detail next.
Figure 2.14: (a) Image of an eye. See Figure 2.15 for a different view of the detail in the box. (b) The gradient magnitude of the image in (a). Black is zero, white is large. (c), (d) The partial derivatives in the horizontal (c) and vertical (d) direction. Gray is zero, black is large negative, white is large positive. Recall that a positive value of $y$ is upwards.
Figure 2.15: Quiver diagram of the gradient in the detail from the box of Figure 2.14 (a). Note the long arrows pointing towards the bright glint in the eye, and those pointing from the pupil and from the eyelid towards the eye white.
Ridge Detection. The gradient vector \( g(x, y) \) encodes both the direction of greatest change and the rate of change in that direction. Consider a small image patch around the lower tip of the letter 'C' in Figure 2.16(a).

The image intensity function from this detail is shown in Figure 2.16(b). The blue curve in this Figure is the curve of inflection of the intensity function, that is, the curve where the function changes from convex to concave. When walking along the gradient \( g \), the derivative along \( g \) at a point \((x, y)\) on the curve of inflection reaches a maximum, and then starts decreasing. Thus, the gradient magnitude reaches a maximum at points on the inflection curve when walking in the direction of the gradient.

Figure 2.16(c) shows a plot of the magnitude \( g(x, y) \) of the gradient of the image function \( I(x, y) \) in Figure 2.16(b). The blue curve in (c) is the ridge of the magnitude of the gradient, and corresponds to the curve of inflection (blue curve) in (b).

Algorithmically, ridge points of the gradient magnitude function \( g(x, y) \) can be found as follows. For each pixel position \( q = (x, y) \) determine the values of \( g \) at the three points

\[
p = q - u, \quad q, \quad r = q + u
\]

where

\[
u = \frac{g(x, y)}{g(x, y)}
\]

is a unit vector along the gradient. Thus, \( p, q, r \) are three points just before, at, and just after the pixel position \( q \) when walking through it in the direction of the gradient. The point \( q \) is a ridge point if the gradient magnitude reaches a local maximum at \( q \) in that direction, that is, if the following conditions are satisfied:

\[
g(p) < g(q) \quad \text{and} \quad g(q) > g(r) .
\]

(18)

If \( s(x, y) \) is the skeleton image, that is, the image of the local maxima thus found, and \( g(x, y) \) is the gradient magnitude image, then edge detection forms the ridge image

\[
r(x, y) = s(x, y) g(x, y)
\]
which records the value of $g$ at all ridge points, and is zero everywhere else.

**Practical Aspects: Issues Related to Image Discretization and Quantization.** The direction $\mathbf{u}$ of the gradient magnitude is generally not aligned with the pixel grid. Because of this, the coordinates of the points $p$ and $r$ are usually not integer. One then determines the values of $g(p)$ and $g(r)$ by bilinear interpolation. For instance, let $p = (x_p, y_p)$, and

$$
\xi = \lfloor x_p \rfloor, \quad \eta = \lfloor y_p \rfloor
$$

$$
\Delta x = x_p - \xi, \quad \Delta y = y_p - \eta.
$$

Then,

$$
g(p) = g(\xi, \eta) (1 - \Delta x) (1 - \Delta y) + g(\xi + 1, \eta) \Delta x (1 - \Delta y) + g(\xi, \eta + 1) (1 - \Delta x) \Delta y + g(\xi + 1, \eta + 1) \Delta x \Delta y.
$$

In addition, because of possible quantization in the function values, there may be a plateau rather than a single ridge point:

$$
g(p) < g(q_1) = \ldots = g(q_k) > g(r).
$$

In this case the check (18) for a maximum would fail. To address this problem, one can use a weaker check:

$$
g(p) \leq g(q) \quad \text{and} \quad g(q) \geq g(r).
$$

This check would reveal several ridge point candidates where only one is desired. Rather than a curve, one then generally finds a ridge region. This region must then be thinned into a curve.

Thinning is a tricky operation if the topology of the original region is to be preserved. A survey of thinning methods is given in [7]. One of these methods is implemented by the Matlab function `bwmorph` called with the ‘thin’ option.

A simpler thinning method based on the distance transform is presented next. This method does not necessarily preserve topology, and is described mainly to give the flavor of some of these methods. The region to be thinned is assumed to be given in the form of a binary image where a value of 1 denotes pixels in the region, and 0 denotes pixels outside (“background”). The *distance transform* of the image is a new image that for each pixel records the Manhattan distance\(^{12}\) to the nearest background point. The transform can be computed in two passes in place over the image [9]. Let $I$ be an image with $m$ rows and $n$ columns. The image contains binary values initially, as specified above, but is of an unsigned integer type, so that it can hold the distance transform. Define $I(x, y)$ to be infinity outside the image (or modify the min operations in the code below so they do not check for values outside the image). A value that is a sufficiently large proxy for infinity is $m + n$. The two-pass algorithm is as follows:

\[^{12}\text{The Manhattan or } L_1 \text{ distance between points } (x, y) \text{ and } (x', y') \text{ is } |x - x'| + |y - y'|. \text{ Think of streets (x) and avenues (y) in Manhattan.}\]
end
end
end
for y = m - 1 down to 0
    for x = n - 1 down to 0
        \[ I(x, y) = \min(I(x, y), I(x + 1, y) + 1, I(x, y + 1) + 1) \]
    end
end

Given the result \( I(x, y) \) of this algorithm, the skeleton or medial axis of the region defined by the original binary image is the set of local (four-connected) maxima, that is, the set of points that satisfy the following predicate:

\[ I(x, y) \geq \max(I(x - 1, y), I(x, y - 1), I(x + 1, y), I(x, y + 1)) . \]

This region is either one or two pixels thick everywhere.

Figure 2.17 shows the results of this procedure. For edge detection, thinning would have to preserve the endpoints of the “thick edges” as well as the topology.

![Figure 2.17](image_url)

**Figure 2.17:** (a) A binary image with a “thick curve.” (b) The distance transform of the image in (a). (c) The skeleton of the region in (a).

**Hysteresis Thresholding**  Ridges of the gradient magnitude that have a low value are often due to whatever amount of image noise is left after the smoothing that is implicit in the differentiation process used to compute the image gradient. Because of this, it is desirable to suppress small ridges after thinning.

However, use of a single threshold on the ridge values would also shorten strong ridges, since ridge values tend to taper off at the endpoints of edges. To avoid this, two thresholds can be used instead, \( g_L \) and \( g_H \), with \( g_L < g_H \). First, ridge points whose value exceeds the high threshold \( g_H \) are declared to be strong edge points:

\[ h(x, y) = r(x, y) \geq g_H . \]
Second, the binary, weak-edge image

\[ l(x, y) = r(x, y) \geq g_L \]

is formed. The final edge map is the image \( e \) of all the pixels in \( l \) that can be reached from pixels in \( h \) without traversing zero-valued pixels in \( l \).

One way to compute \( e \) is to consider the “on” pixels (pixels with value 1) in \( l \) as nodes of a graph, whose edges are the adjacency relations between pixels (a pixel is adjacent to its north, east, south, and west neighbors). By construction, the “on” pixels in \( h \) are a subset of those of \( l \). One can then build a spanning forest for the components of \( l \) that contain some pixels of \( h \). The pixels in this forest form the edge image \( e \). Figure 2.18 shows a simple image, a plot of its intensity, gradient magnitude, and edge map.

**Practical Aspects: Computing the Spanning Forest.** As a sample implementation outline, the following algorithm scans \( h \) in raster order. Once it encounters an “on” pixel at position \((x, y)\), it uses a stack to traverse pixels in \( l \) starting at \((x, y)\) in depth-first order. As it goes, it erases the pixels it visits in both \( l \) (to avoid loops) and \( h \) (to avoid revisiting a component), and marks them in the initially empty image \( e \).

```plaintext
for y = 0 to m - 1
    for x = 0 to n - 1
        e(x, y) = 0
    end
end

for y = 0 to m - 1
    for x = 0 to n - 1
        if h(x, y)
            push((x, y))
        while the stack is not empty
            \((x', y') = pop\)
            e(x', y') = 1
            h(x', y') = l(x', y') = 0
            for all neighbors \((x'', y'')\) of \((x', y')\)
                if l(x'', y'')
                    push((x'', y''))
            end
        end
    end
end
```

### 2.6.3 Edge Scale

The gradient measurement step in edge detection computes derivatives by convolving the image with derivatives of a Gaussian function, as explained in Section 2.4.1. As the \( \sigma \) parameter of this
Figure 2.18: (a) Input image. (b) Plot of the intensity function in (a). (c) The magnitude of the gradient. (d) Edges found by Canny’s edge detector.

Figure 2.19: (a) An image from [4] that exhibits edges at different scales. (b) Edges detected with $\sigma = 1$ pixel. (c) Edges detected with $\sigma = 20$ pixels.
Gaussian is increased, the differentiation operator blurs the image more and more. In addition to smoothing away noise, the useful part of the image is smoothed as well. This has the effect of making the edge detector sensitive to edges that occur over different spatial extents, or *scales*. The image in Figure 2.19(a) illustrates a situation in which consideration of scale is important.

If $\sigma$ is small, the sharp edges around the mannequin are detected, but the fuzzier edges of the shadow are either missed altogether (if the gradient magnitude is too small) or result in multiple edges, as shown in Figure 2.19(b). If $\sigma$ is large, most of the shadow edges are found, but all edges are rounded, and multiple sharp edges can be mixed together under the wide differentiation kernel, with unpredictable results. This is shown in Figure 2.19(c).

These considerations suggest detecting edges with differentiation operators at multiple scales (values of $\sigma$), and to define the “correct” edge at position $(x,y)$ to be the edge detected with the value $\sigma(x,y)$ of the scale parameter that yields the strongest edge. The definition of edge strength relies naturally on the value $g(x,y)$ of the magnitude of the gradient. However, since $g$ has the dimensions of gray levels per pixel, it has been proposed [8, 5] to measure strength with the quantity

$$S(x,y) = \sigma(x,y)g(x,y)$$

which is dimensionless, and therefore more appropriate to compare strengths measured at different scales. Specific definitions, methods, and algorithms that embody this notion of *scale space analysis* can be found in the references above.

**References**


