CHAPTER 5

Image Processing

Overview

At this point we have all of the basics at our disposal. We understand the structure of the library as well as the basic data structures it uses to represent images. We understand the HighGUI interface and can actually run a program and display our results on the screen. Now that we understand these primitive methods required to manipulate image structures, we are ready to learn some more sophisticated operations.

We will now move on to higher-level methods that treat the images as images, and not just as arrays of colored (or grayscale) values. When we say "image processing", we mean just that: using higher-level operators that are defined on image structures in order to accomplish tasks whose meaning is naturally defined in the context of graphical, visual images.

Smoothing

Smoothing, also called *blurring*, is a simple and frequently used image processing operation. There are many reasons for smoothing, but it is usually done to reduce noise or camera artifacts. Smoothing is also important when we wish to reduce the resolution of an image in a principled way (we will discuss this in more detail in the "Image Pyramids" section of this chapter).

OpenCV offers five different smoothing operations at this time. All of them are supported through one function, `cvSmooth()`,* which takes our desired form of smoothing as an argument.

```
void cvSmooth(
  const CvArr*   src,
  CvArr*         dst,
  int            smoothtype  = CV_GAUSSIAN,
  int            param1      = 3,
```

* Note that—unlike in, say, Matlab—the filtering operations in OpenCV (e.g., `cvSmooth()`, `cvErode()`, `cvDilate()`) produce output images of the same size as the input. To achieve that result, OpenCV creates "virtual" pixels outside of the image at the borders. By default, this is done by replication at the border, i.e., `input(-dx,y)=input(0,y)`, `input(w+dx,y)=input(w-1,y)`, and so forth.

```
    int             param2      = 0,
    double          param3      = 0,
    double          param4      = 0
);
```

The `src` and `dst` arguments are the usual source and destination for the smooth operation. The `cv_Smooth()` function has four parameters with the particularly uninformative names of param1, param2, param3, and param4. The meaning of these parameters depends on the value of `smoothtype`, which may take any of the five values listed in Table 5-1.* (Please notice that for some values of ST, "in place operation", in which `src` and `dst` indicate the same image, is not allowed.)

Table 5-1. Types of smoothing operations

Smooth type	Name	In place?	Nc	Depth of src	Depth of dst	Brief description
CV_BLUR	Simple blur	Yes	1,3	8u, 32f	8u, 32f	Sum over a param1×param2 neighborhood with subsequent scaling by 1/(param1×param2).
CV_BLUR_NO_SCALE	Simple blur with no scaling	No	1	8u	16s (for 8u source) or 32f (for 32f source)	Sum over a param1×param2 neighborhood.
CV_MEDIAN	Median blur	No	1,3	8u	8u	Find median over a param1×param1 square neighborhood.
CV_GAUSSIAN	Gaussian blur	Yes	1,3	8u, 32f	8u (for 8u source) or 32f (for 32f source)	Sum over a param1×param2 neighborhood.
CV_BILATERAL	Bilateral filter	No	1,3	8u	8u	Apply bilateral 3-by-3 filtering with color sigma=param1 and a space sigma=param2.

The *simple blur* operation, as exemplified by `CV_BLUR` in Figure 5-1, is the simplest case. Each pixel in the output is the simple mean of all of the pixels in a window around the corresponding pixel in the input. Simple blur supports 1–4 image channels and works on 8-bit images or 32-bit floating-point images.

Not all of the smoothing operators act on the same sorts of images. `CV_BLUR_NO_SCALE` (*simple blur without scaling*) is essentially the same as simple blur except that there is no division performed to create an average. Hence the source and destination images must have different numerical precision so that the blurring operation will not result in an overflow. Simple blur without scaling may be performed on 8-bit images, in which case the destination image should have `IPL_DEPTH_16S` (`CV_16S`) or `IPL_DEPTH_32S` (`CV_32S`)

* Here and elsewhere we sometimes use 8u as shorthand for 8-bit unsigned image depth (`IPL_DEPTH_8U`). See Table 3-2 for other shorthand notation.

Figure 5-1. Image smoothing by block averaging: on the left are the input images; on the right, the output images

data types. The same operation may also be performed on 32-bit floating-point images, in which case the destination image may also be a 32-bit floating-point image. Simple blur without scaling cannot be done in place: the source and destination images must be different. (This requirement is obvious in the case of 8 bits to 16 bits, but it applies even when you are using a 32-bit image). Simple blur without scaling is sometimes chosen because it is a little faster than blurring with scaling.

The *median filter* (`CV_MEDIAN`) [Bardyn84] replaces each pixel by the median or "middle" pixel (as opposed to the mean pixel) value in a square neighborhood around the center pixel. Median filter will work on single-channel or three-channel or four-channel 8-bit images, but it cannot be done in place. Results of median filtering are shown in Figure 5-2. Simple blurring by averaging can be sensitive to noisy images, especially images with large isolated outlier points (sometimes called "shot noise"). Large differences in even a small number of points can cause a noticeable movement in the average value. Median filtering is able to ignore the outliers by selecting the middle points.

The next smoothing filter, the *Gaussian filter* (`CV_GAUSSIAN`), is probably the most useful though not the fastest. Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel and then summing to produce the output array.

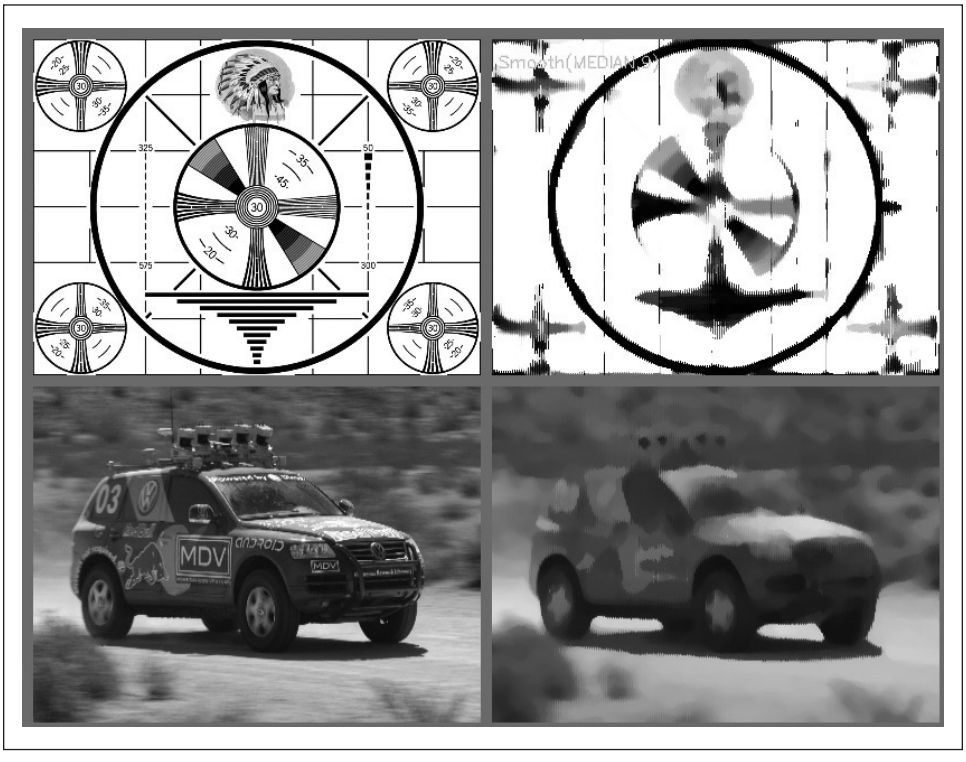

Figure 5-2. Image blurring by taking the median of surrounding pixels

For the Gaussian blur (Figure 5-3), the first two parameters give the width and height of the filter window; the (optional) third parameter indicates the sigma value (half width at half max) of the Gaussian kernel. If the third parameter is not specified, then the Gaussian will be automatically determined from the window size using the following formulae:

$$\sigma_x = \left(\frac{n_x}{2} - 1\right) \cdot 0.30 + 0.80, \quad n_x = \text{param1}$$

$$\sigma_y = \left(\frac{n_y}{2} - 1\right) \cdot 0.30 + 0.80, \quad n_y = \text{param2}$$

If you wish the kernel to be asymmetric, then you may also (optionally) supply a fourth parameter; in this case, the third and fourth parameters will be the values of sigma in the horizontal and vertical directions, respectively.

If the third and fourth parameters are given but the first two are set to 0, then the size of the window will be automatically determined from the value of sigma.

The OpenCV implementation of Gaussian smoothing also provides a higher performance optimization for several common kernels. 3-by-3, 5-by-5 and 7-by-7 with

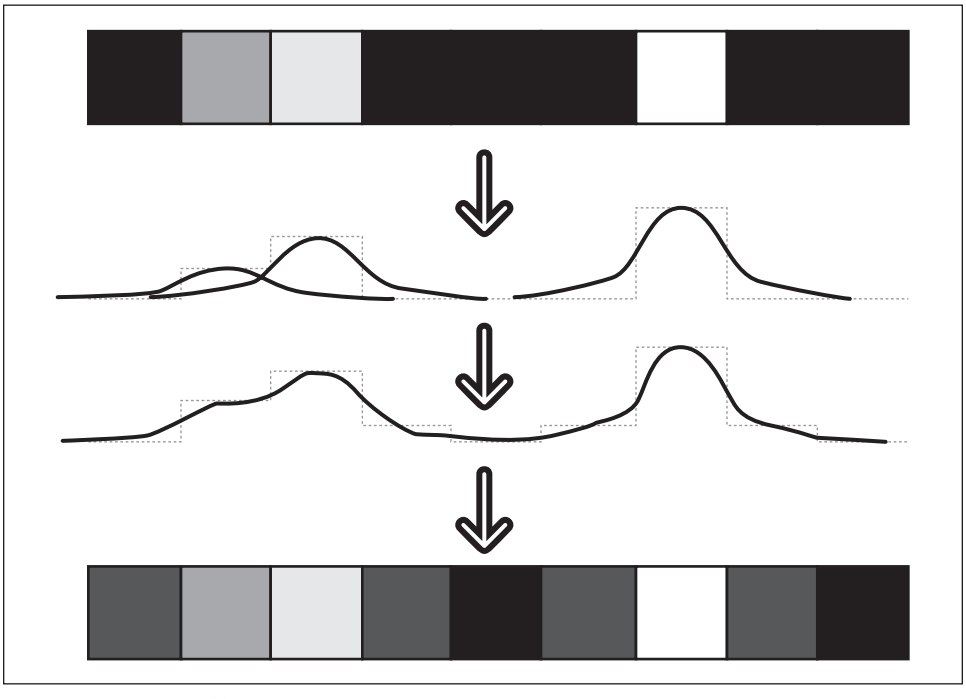

Figure 5-3. Gaussian blur on 1D pixel array

the "standard" sigma (i.e., param3 = 0.0) give better performance than other kernels. Gaussian blur supports single- or three-channel images in either 8-bit or 32-bit floating-point formats, and it can be done in place. Results of Gaussian blurring are shown in Figure 5-4.

The fifth and final form of smoothing supported by OpenCV is called *bilateral filtering* [Tomasi98], an example of which is shown in Figure 5-5. Bilateral filtering is one operation from a somewhat larger class of image analysis operators known as *edge-preserving smoothing*. Bilateral filtering is most easily understood when contrasted to Gaussian smoothing. A typical motivation for Gaussian smoothing is that pixels in a real image should vary slowly over space and thus be correlated to their neighbors, whereas random noise can be expected to vary greatly from one pixel to the next (i.e., noise is not spatially correlated). It is in this sense that Gaussian smoothing reduces noise while preserving signal. Unfortunately, this method breaks down near edges, where you do expect pixels to be uncorrelated with their neighbors. Thus Gaussian smoothing smoothes away the edges. At the cost of a little more processing time, bilateral filtering provides us a means of smoothing an image without smoothing away the edges.

Like Gaussian smoothing, bilateral filtering constructs a weighted average of each pixel and its neighboring components. The weighting has two components, the first of which is the same weighting used by Gaussian smoothing. The second component is also a Gaussian weighting but is based not on the spatial distance from the center pixel

Figure 5-4. Gaussian blurring

but rather on the difference in intensity* from the center pixel.† You can think of bilateral filtering as Gaussian smoothing that weights more similar pixels more highly than less similar ones. The effect of this filter is typically to turn an image into what appears to be a watercolor painting of the same scene.‡ This can be useful as an aid to segmenting the image.

Bilateral filtering takes two parameters. The first is the width of the Gaussian kernel used in the spatial domain, which is analogous to the sigma parameters in the Gaussian filter. The second is the width of the Gaussian kernel in the color domain. The larger this second parameter is, the broader is the range of intensities (or colors) that will be included in the smoothing (and thus the more extreme a discontinuity must be in order to be preserved).

* In the case of multichannel (i.e., color) images, the difference in intensity is replaced with a weighted sum over colors. This weighting is chosen to enforce a Euclidean distance in the CIE color space.

† Technically, the use of Gaussian distribution functions is not a necessary feature of bilateral filtering. The implementation in OpenCV uses Gaussian weighting even though the method is general to many possible weighting functions.

‡ This effect is particularly pronounced after multiple iterations of bilateral filtering.

Figure 5-5. Results of bilateral smoothing

Image Morphology

OpenCV provides a fast, convenient interface for doing *morphological transformations* [Serra83] on an image. The basic morphological transformations are called *dilation* and *erosion*, and they arise in a wide variety of contexts such as removing noise, isolating individual elements, and joining disparate elements in an image. Morphology can also be used to find intensity bumps or holes in an image and to find image gradients.

Dilation and Erosion

Dilation is a convolution of some image (or region of an image), which we will call A, with some *kernel*, which we will call B. The kernel, which can be any shape or size, has a single defined *anchor point*. Most often, the kernel is a small solid square or disk with the anchor point at the center. The kernel can be thought of as a template or mask, and its effect for dilation is that of a *local maximum* operator. As the kernel B is scanned over the image, we compute the maximal pixel value overlapped by B and replace the image pixel under the anchor point with that maximal value. This causes bright regions within an image to grow as diagrammed in Figure 5-6. This growth is the origin of the term "dilation operator".

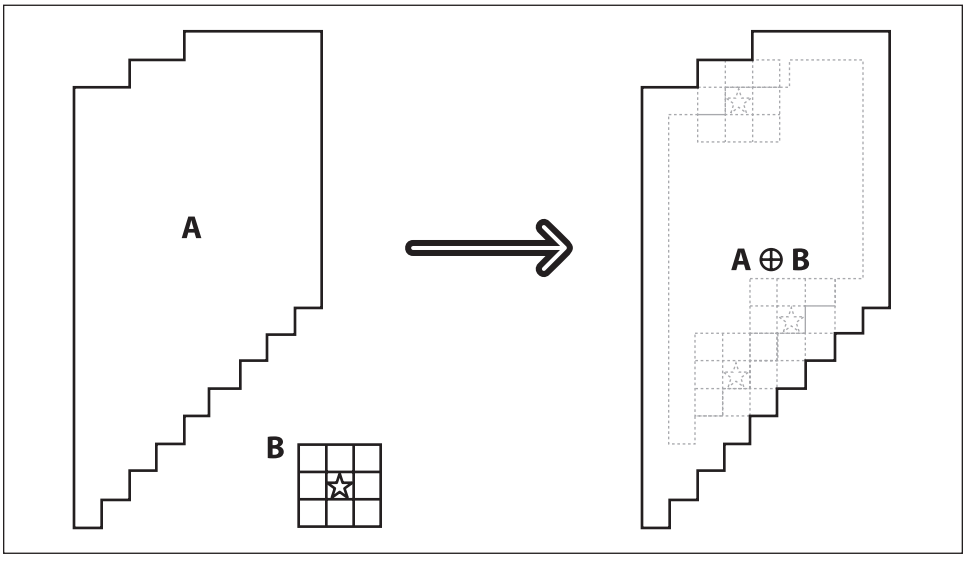

Figure 5-6. Morphological dilation: take the maximum under the kernel B

Erosion is the converse operation. The action of the erosion operator is equivalent to computing a *local minimum* over the area of the kernel. Erosion generates a new image from the original using the following algorithm: as the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value.* Erosion is diagrammed in Figure 5-7.

Image morphology is often done on binary images that result from thresholding. However, because dilation is just a max operator and erosion is just a min operator, morphology may be used on intensity images as well.

In general, whereas dilation expands region A, erosion reduces region A. Moreover, dilation will tend to smooth concavities and erosion will tend to smooth away protrusions. Of course, the exact result will depend on the kernel, but these statements are generally true for the filled convex kernels typically used.

In OpenCV, we effect these transformations using the `cvErode()` and `cvDilate()` functions:

```
void cvErode(
   IplImage*        src,
   IplImage*        dst,
   IplConvKernel*   B          = NULL,
   int              iterations = 1
);
```

* To be precise, the pixel in the destination image is set to the value equal to the minimal value of the pixels under the kernel in the source image.

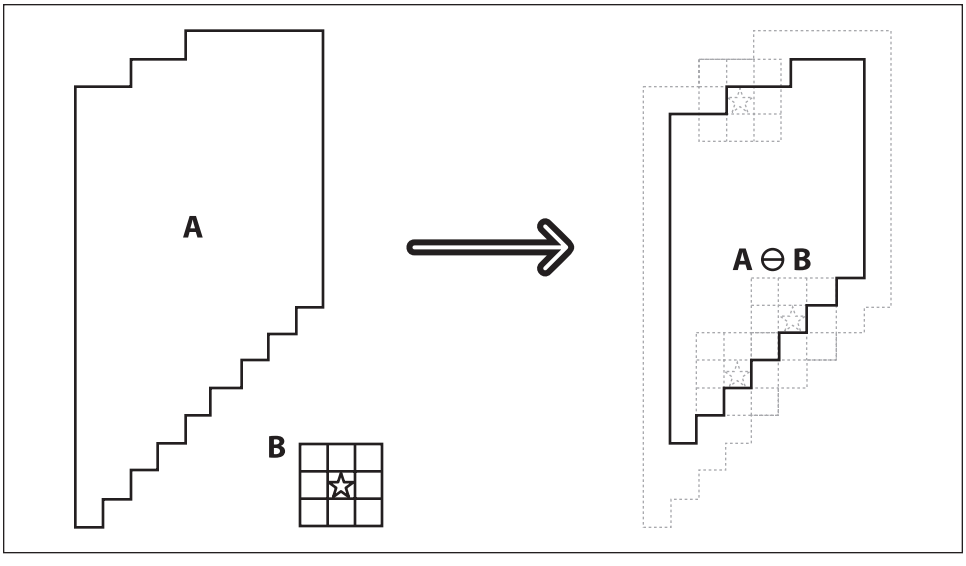

Figure 5-7. Morphological erosion: take the minimum under the kernel B

```
void cvDilate(
   IplImage*        src,
   IplImage*        dst,
   IplConvKernel*   B          = NULL,
   int              iterations = 1
);
```

Both `cvErode()` and `cvDilate()` take a source and destination image, and both support "in place" calls (in which the source and destination are the same image). The third argument is the kernel, which defaults to `NULL`. In the `NULL` case, the kernel used is a 3-by-3 kernel with the anchor at its center (we will discuss shortly how to create your own kernels). Finally, the fourth argument is the number of iterations. If not set to the default value of 1, the operation will be applied multiple times during the single call to the function. The results of an erode operation are shown in Figure 5-8 and those of a dilation operation in Figure 5-9. The erode operation is often used to eliminate "speckle" noise in an image. The idea here is that the speckles are eroded to nothing while larger regions that contain visually significant content are not affected. The dilate operation is often used when attempting to find *connected components* (i.e., large discrete regions of similar pixel color or intensity). The utility of dilation arises because in many cases a large region might otherwise be broken apart into multiple components as a result of noise, shadows, or some other similar effect. A small dilation will cause such components to "melt" together into one.

To recap: when OpenCV processes the `cvErode()` function, what happens beneath the hood is that the value of some point p is set to the minimum value of all of the points covered by the kernel when aligned at p; for the dilation operator, the equation is the same except that max is considered rather than min:

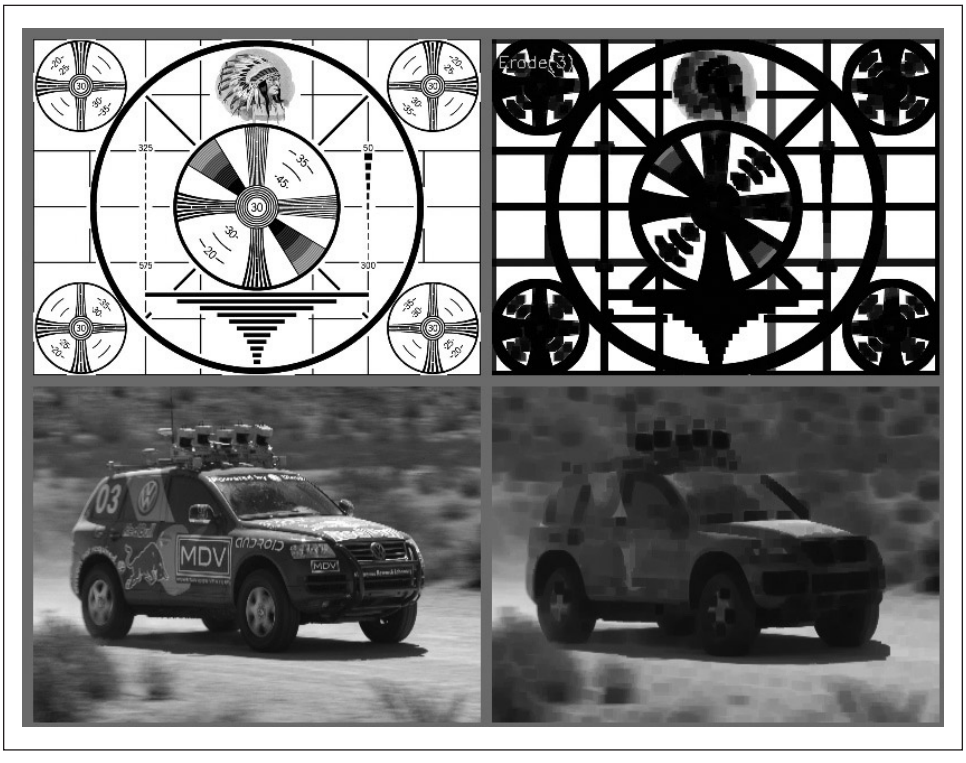

Figure 5-8. Results of the erosion, or "min", operator: bright regions are isolated and shrunk

$$\text{erode}(x,y) = \min_{(x',y')\in\text{kernel}} \text{src}(x+x', y+y')$$

$$\text{dilate}(x,y) = \max_{(x',y')\in\text{kernel}} \text{src}(x+x', y+y')$$

You might be wondering why we need a complicated formula when the earlier heuristic description was perfectly sufficient. Some readers actually prefer such formulas but, more importantly, the formulas capture some generality that isn't apparent in the qualitative description. Observe that if the image is not binary then the min and max operators play a less trivial role. Take another look at Figures 5-8 and 5-9, which show the erosion and dilation operators applied to two real images.

Making Your Own Kernel

You are not limited to the simple 3-by-3 square kernel. You can make your own custom morphological kernels (our previous "kernel B") using `IplConvKernel`. Such kernels are allocated using `cvCreateStructuringElementEx()` and are released using `cvReleaseStructuringElement()`.

```
IplConvKernel* cvCreateStructuringElementEx(
   int          cols,
   int          rows,
```

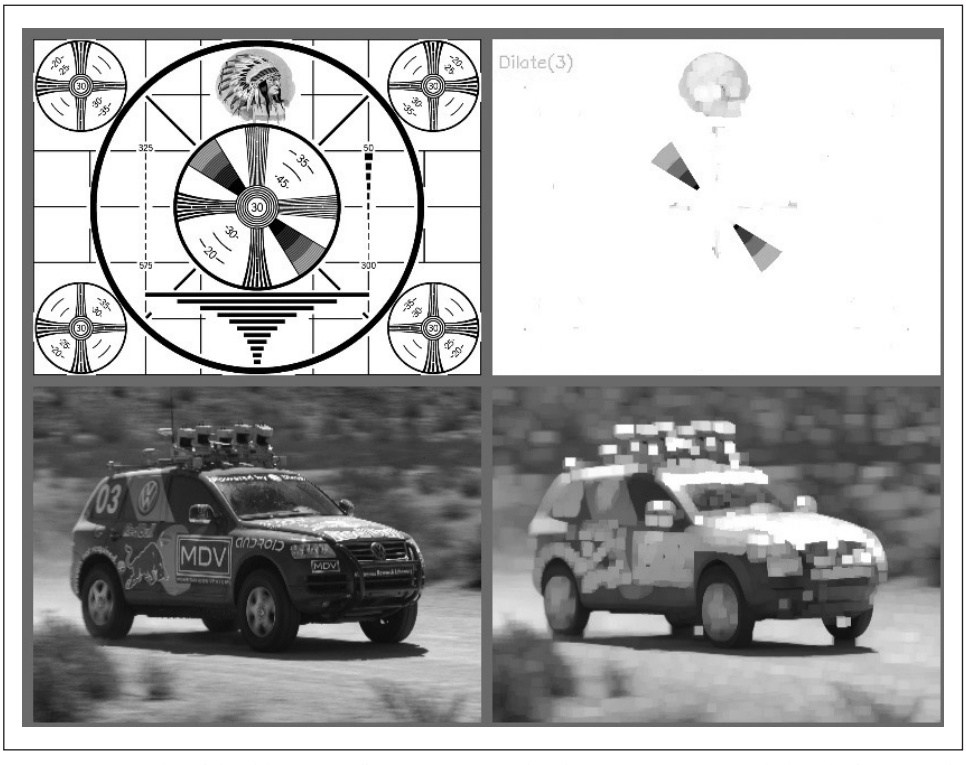

Figure 5-9. Results of the dilation, or "max", operator: bright regions are expanded and often joined

```
    int          anchor_x,
    int          anchor_y,
    int          shape,
    int*         values=NULL
);

void cvReleaseStructuringElement( IplConvKernel** element );
```

A morphological kernel, unlike a convolution kernel, doesn't require any numerical values. The elements of the kernel simply indicate where the max or min computations take place as the kernel moves around the image. The anchor point indicates how the kernel is to be aligned with the source image and also where the result of the computation is to be placed in the destination image. When creating the kernel, `cols` and `rows` indicate the size of the rectangle that holds the structuring element. The next parameters, `anchor_x` and `anchor_y`, are the (*x*, *y*) coordinates of the anchor point within the enclosing rectangle of the kernel. The fifth parameter, `shape`, can take on values listed in Table 5-2. If CV_SHAPE_CUSTOM is used, then the integer vector `values` is used to define a custom shape of the kernel within the `rows`-by-`cols` enclosing rectangle. This vector is read in raster scan order with each entry representing a different pixel in the enclosing rectangle. Any nonzero value is taken to indicate that the corresponding pixel

should be included in the kernel. If `values` is `NULL` then the custom shape is interpreted to be all nonzero, resulting in a rectangular kernel.*

Table 5-2. Possible IplConvKernel shape values

Shape value	Meaning
CV_SHAPE_RECT	The kernel is rectangular
CV_SHAPE_CROSS	The kernel is cross shaped
CV_SHAPE_ELLIPSE	The kernel is elliptical
CV_SHAPE_CUSTOM	The kernel is user-defined via `values`

More General Morphology

When working with Boolean images and image masks, the basic erode and dilate operations are usually sufficient. When working with grayscale or color images, however, a number of additional operations are often helpful. Several of the more useful operations can be handled by the multi-purpose `cvMorphologyEx()` function.

```
void cvMorphologyEx(
   const CvArr*   src,
   CvArr*         dst,
   CvArr*         temp,
   IplConvKernel* element,
   int            operation,
   int            iterations   = 1
);
```

In addition to the arguments `src`, `dst`, `element`, and `iterations`, which we used with previous operators, `cvMorphologyEx()` has two new parameters. The first is the `temp` array, which is required for some of the operations (see Table 5-3). When required, this array should be the same size as the source image. The second new argument—the really interesting one—is `operation`, which selects the morphological operation that we will do.

Table 5-3. cvMorphologyEx() operation options

Value of operation	Morphological operator	Requires temp image?
CV_MOP_OPEN	Opening	No
CV_MOP_CLOSE	Closing	No
CV_MOP_GRADIENT	Morphological gradient	Always
CV_MOP_TOPHAT	Top Hat	For in-place only (`src = dst`)
CV_MOP_BLACKHAT	Black Hat	For in-place only (`src = dst`)

Opening and closing

The first two operations in Table 5-3, *opening* and *closing*, are combinations of the erosion and dilation operators. In the case of opening, we erode first and then dilate (Figure 5-10).

* If the use of this strange integer vector strikes you as being incongruous with other OpenCV functions, you are not alone. The origin of this syntax is the same as the origin of the IPL prefix to this function—another instance of archeological code relics.

Opening is often used to count regions in a binary image. For example, if we have thresholded an image of cells on a microscope slide, we might use opening to separate out cells that are near each other before counting the regions. In the case of closing, we dilate first and then erode (Figure 5-12). Closing is used in most of the more sophisticated connected-component algorithms to reduce unwanted or noise-driven segments. For connected components, usually an erosion or closing operation is performed first to eliminate elements that arise purely from noise and then an opening operation is used to connect nearby large regions. (Notice that, although the end result of using open or close is similar to using erode or dilate, these new operations tend to preserve the area of connected regions more accurately.)

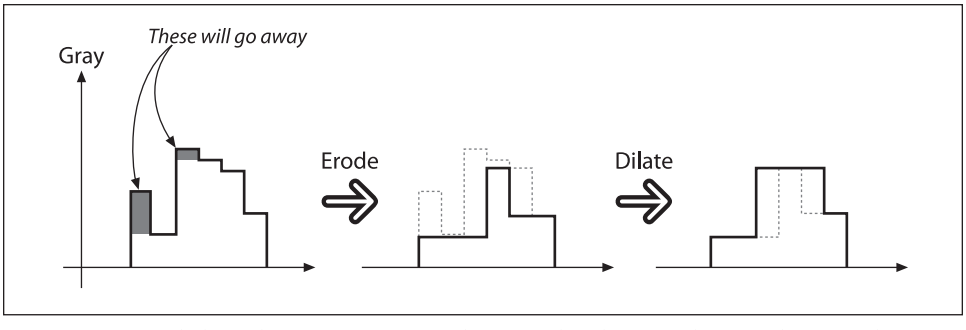

Figure 5-10. Morphological opening operation: the upward outliers are eliminated as a result

Both the opening and closing operations are approximately area-preserving: the most prominent effect of closing is to eliminate lone outliers that are lower than their neighbors whereas the effect of opening is to eliminate lone outliers that are higher than their neighbors. Results of using the opening operator are shown in Figure 5-11, and of the closing operator in Figure 5-13.

One last note on the opening and closing operators concerns how the `iterations` argument is interpreted. You might expect that asking for two iterations of closing would yield something like dilate-erode-dilate-erode. It turns out that this would not be particularly useful. What you really want (and what you get) is dilate-dilate-erode-erode. In this way, not only the single outliers but also neighboring pairs of outliers will disappear.

Morphological gradient

Our next available operator is the *morphological gradient*. For this one it is probably easier to start with a formula and then figure out what it means:

$$\text{gradient(src)} = \text{dilate(src)} - \text{erode(src)}$$

The effect of this operation on a Boolean image would be simply to isolate perimeters of existing blobs. The process is diagrammed in Figure 5-14, and the effect of this operator on our test images is shown in Figure 5-15.

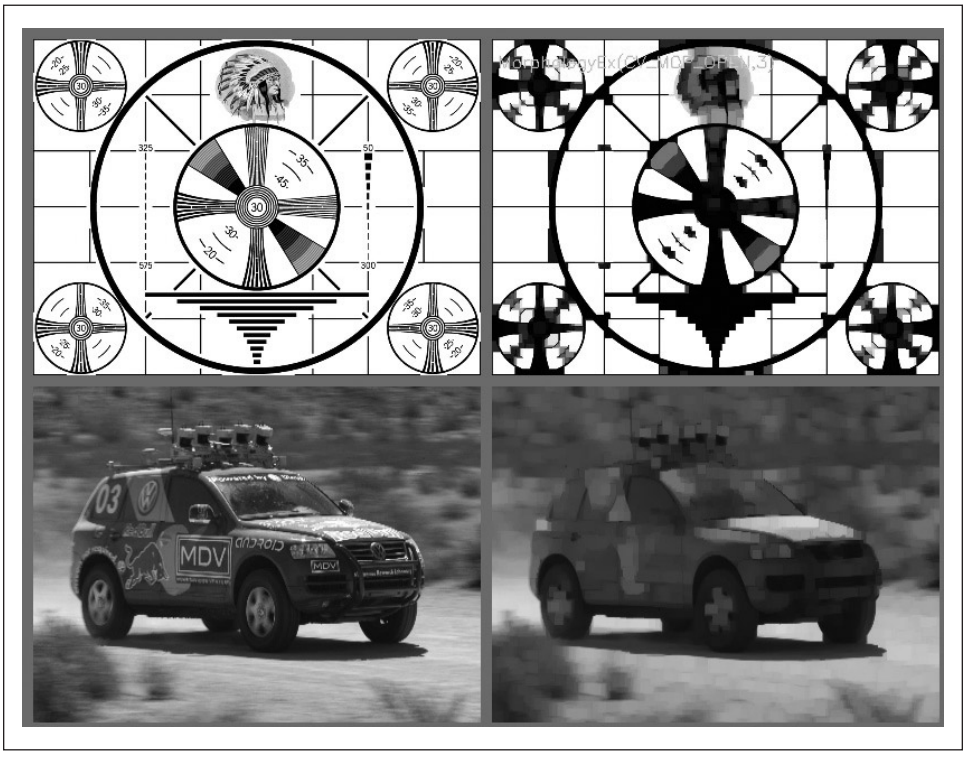

Figure 5-11. Results of morphological opening on an image: small bright regions are removed, and the remaining bright regions are isolated but retain their size

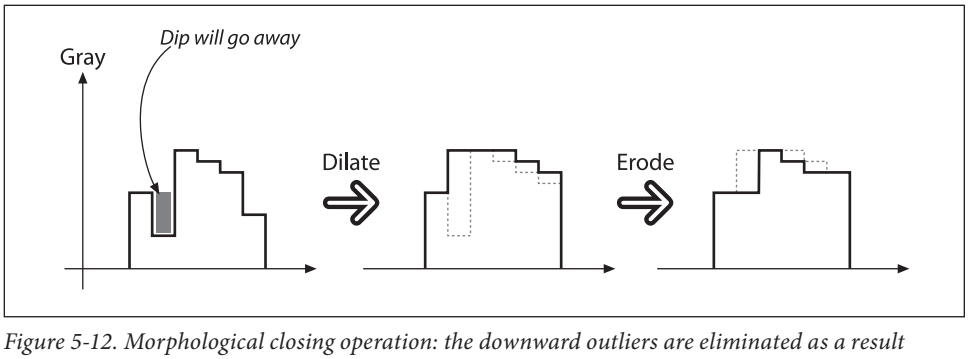

Figure 5-12. Morphological closing operation: the downward outliers are eliminated as a result

With a grayscale image we see that the value of the operator is telling us something about how fast the image brightness is changing; this is why the name "morphological gradient" is justified. Morphological gradient is often used when we want to isolate the perimeters of bright regions so we can treat them as whole objects (or as whole parts of objects). The complete perimeter of a region tends to be found because an expanded version is subtracted from a contracted version of the region, leaving a complete perimeter

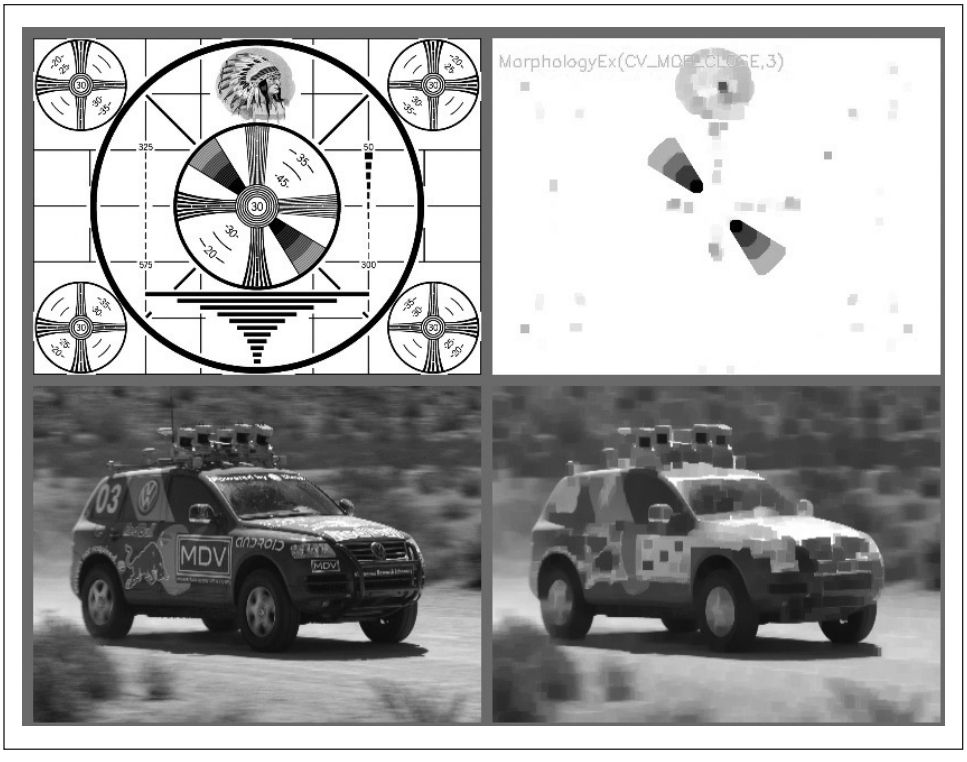

Figure 5-13. Results of morphological closing on an image: bright regions are joined but retain their basic size

edge. This differs from calculating a gradient, which is much less likely to work around the full perimeter of an object.*

Top Hat and Black Hat

The last two operators are called *Top Hat* and *Black Hat* [Meyer78]. These operators are used to isolate patches that are, respectively, brighter or dimmer than their immediate neighbors. You would use these when trying to isolate parts of an object that exhibit brightness changes relative only to the object to which they are attached. This often occurs with microscope images of organisms or cells, for example. Both operations are defined in terms of the more primitive operators, as follows:

$$\text{TopHat(src)} = \text{src} - \text{open(src)}$$

$$\text{BlackHat(src)} = \text{close(src)} - \text{src}$$

As you can see, the Top Hat operator subtracts the opened form of A from A. Recall that the effect of the open operation was to exaggerate small cracks or local drops. Thus,

* We will return to the topic of gradients when we introduce the Sobel and Scharr operators in the next chapter.

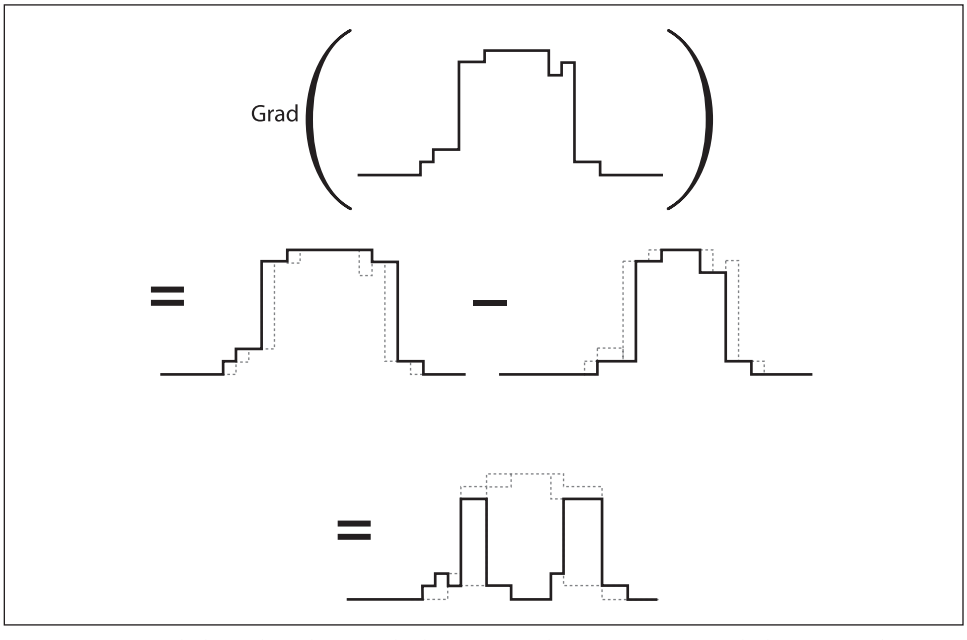

Figure 5-14. Morphological gradient applied to a grayscale image: as expected, the operator has its highest values where the grayscale image is changing most rapidly

subtracting open(A) from A should reveal areas that are lighter then the surrounding region of A, relative to the size of the kernel (see Figure 5-16); conversely, the Black Hat operator reveals areas that are darker than the surrounding region of A (Figure 5-17). Summary results for all the morphological operators discussed in this chapter are assembled in Figure 5-18.*

Flood Fill

Flood fill [Heckbert00; Shaw04; Vandevenne04] is an extremely useful function that is often used to mark or isolate portions of an image for further processing or analysis. Flood fill can also be used to derive, from an input image, masks that can be used for subsequent routines to speed or restrict processing to only those pixels indicated by the mask. The function `cvFloodFill()` itself takes an optional mask that can be further used to control where filling is done (e.g., when doing multiple fills of the same image).

In OpenCV, flood fill is a more general version of the sort of fill functionality which you probably already associate with typical computer painting programs. For both, a *seed point* is selected from an image and then all similar neighboring points are colored with a uniform color. The difference here is that the neighboring pixels need not all be

* Both of these operations (Top Hat and Black Hat) make more sense in grayscale morphology, where the structuring element is a matrix of real numbers (not just a binary mask) and the matrix is added to the current pixel neighborhood before taking a minimum or maximum. Unfortunately, this is not yet implemented in OpenCV.

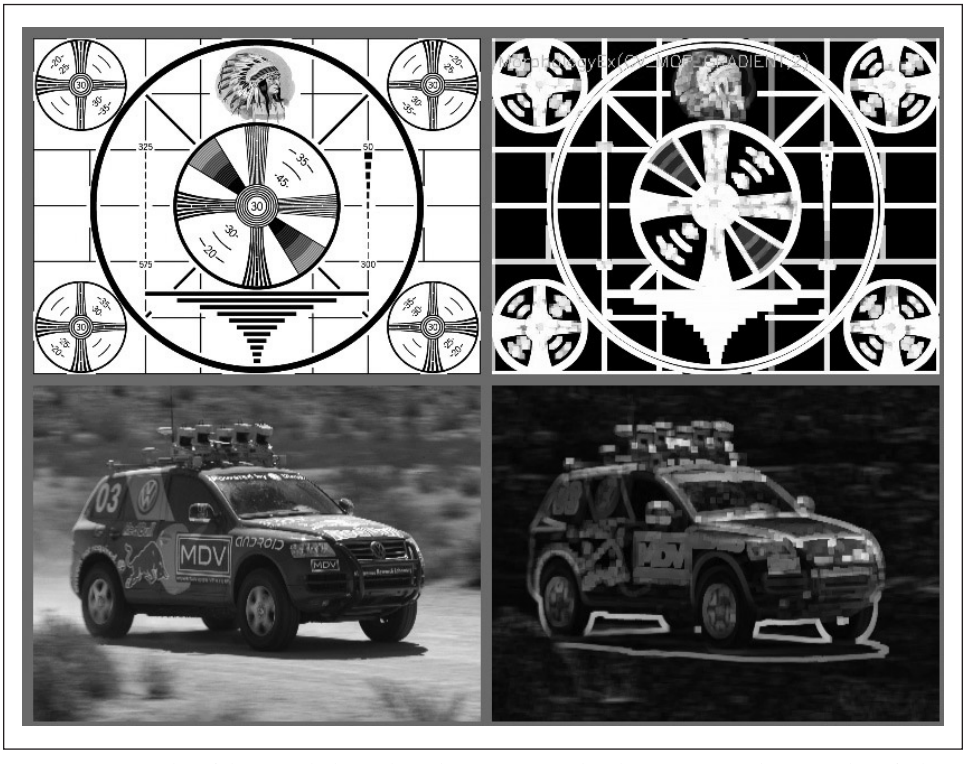

Figure 5-15. Results of the morphological gradient operator: bright perimeter edges are identified

identical in color.* The result of a flood fill operation will always be a single contiguous region. The `cvFloodFill()` function will color a neighboring pixel if it is within a specified range (`loDiff` to `upDiff`) of either the current pixel or if (depending on the settings of `flags`) the neighboring pixel is within a specified range of the original `seedPoint` value. Flood filling can also be constrained by an optional mask argument. The prototype for the flood fill routine is:

```
void cvFloodFill(
   IplImage*          img,
   CvPoint            seedPoint,
   CvScalar           newVal,
   CvScalar           loDiff    = cvScalarAll(0),
   CvScalar           upDiff    = cvScalarAll(0),
   CvConnectedComp*   comp      = NULL,
   int                flags     = 4,
   CvArr*             mask      = NULL
);
```

The parameter `img` is the input image, which can be 8-bit or floating-point and one-channel or three-channel. We start the flood filling from `seedPoint`, and `newVal` is the

* Users of contemporary painting and drawing programs should note that most now employ a filling algorithm very much like `cvFloodFill()`.

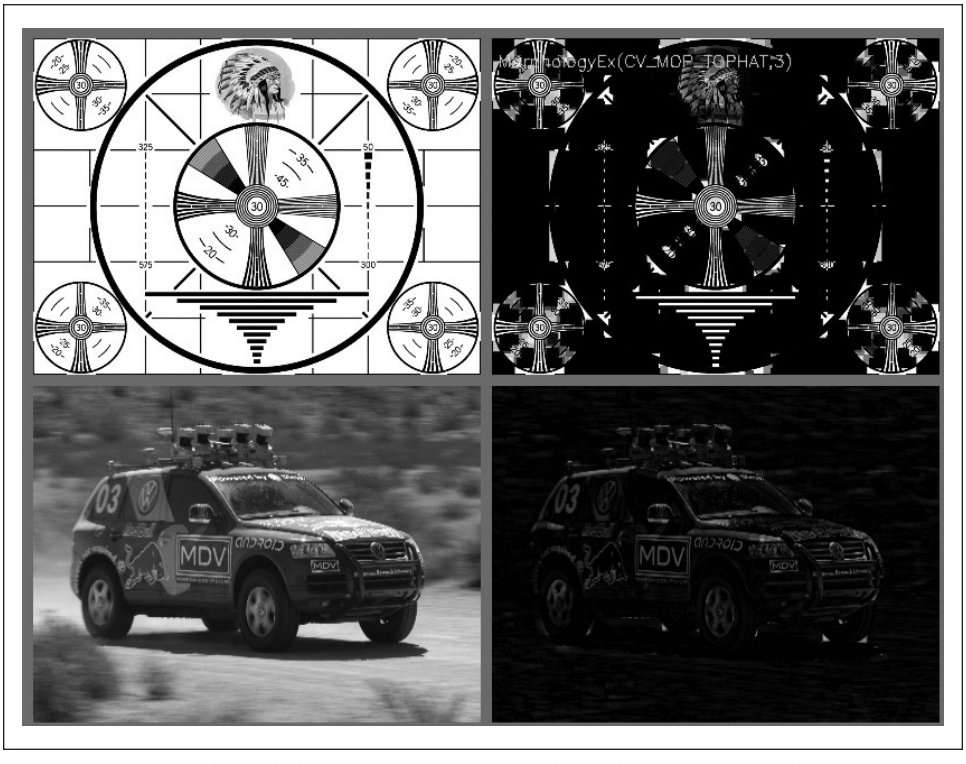

Figure 5-16. Results of morphological Top Hat operation: bright local peaks are isolated

value to which colorized pixels are set. A pixel will be colorized if its intensity is not less than a colorized neighbor's intensity minus `loDiff` and not greater than the colorized neighbor's intensity plus `upDiff`. If the `flags` argument includes `CV_FLOODFILL_FIXED_RANGE`, then a pixel will be compared to the original seed point rather than to its neighbors. If non-`NULL`, `comp` is a `CvConnectedComp` structure that will hold statistics about the areas filled.* The `flags` argument (to be discussed shortly) is a little tricky; it controls the connectivity of the fill, what the fill is relative to, whether we are filling only a mask, and what values are used to fill the mask. Our first example of flood fill is shown in Figure 5-19.

The argument `mask` indicates a mask that can function both as input to `cvFloodFill()` (in which case it constrains the regions that can be filled) and as output from `cvFloodFill()` (in which case it will indicate the regions that actually were filled). If set to a non-`NULL` value, then `mask` must be a one-channel, 8-bit image whose size is exactly two pixels larger in width and height than the source image (this is to make processing easier and faster for the internal algorithm). Pixel ($x + 1$, $y + 1$) in the mask image corresponds to image pixel (x, y) in the source image. Note that `cvFloodFill()` will not flood across

* We will address the specifics of a "connected component" in the section "Image Pyramids". For now, just think of it as being similar to a mask that identifies some subsection of an image.

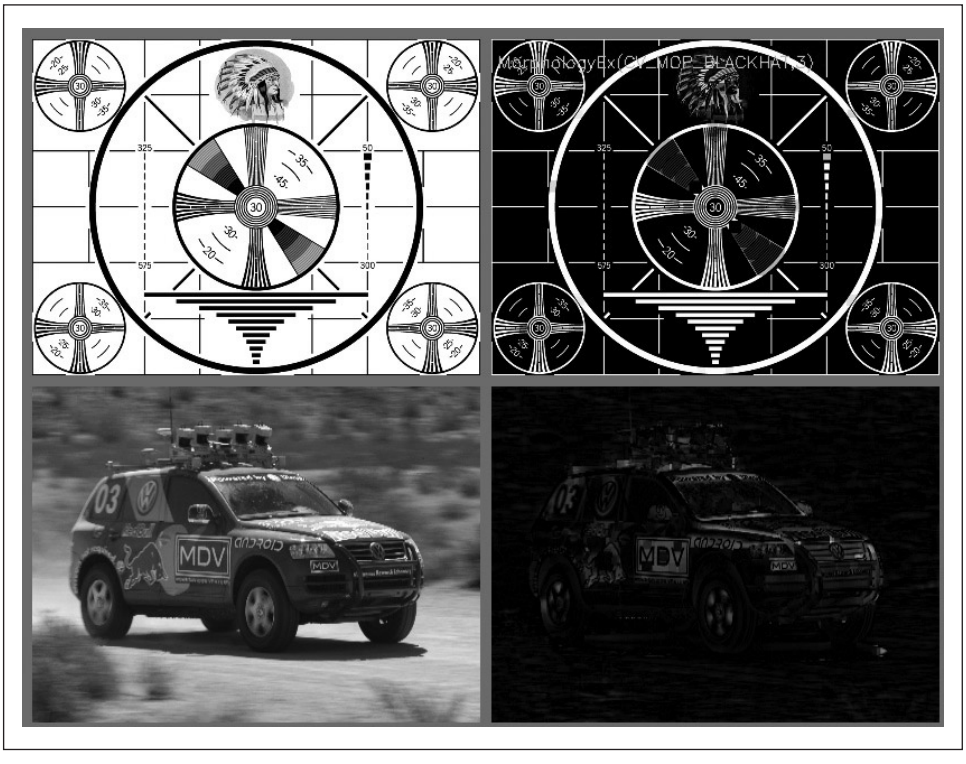

Figure 5-17. Results of morphological Black Hat operation: dark holes are isolated

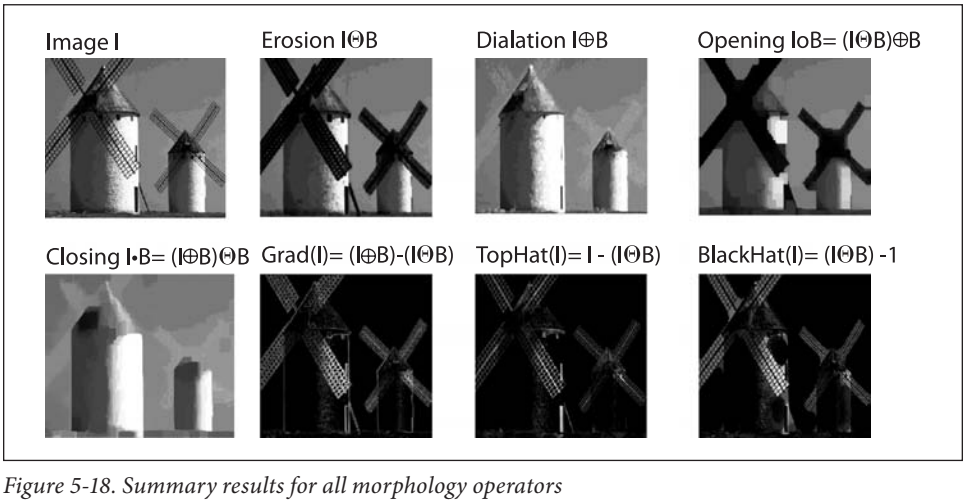

Figure 5-18. Summary results for all morphology operators

nonzero pixels in the mask, so you should be careful to zero it before use if you don't want masking to block the flooding operation. Flood fill can be set to colorize either the source image `img` or the mask image `mask`.

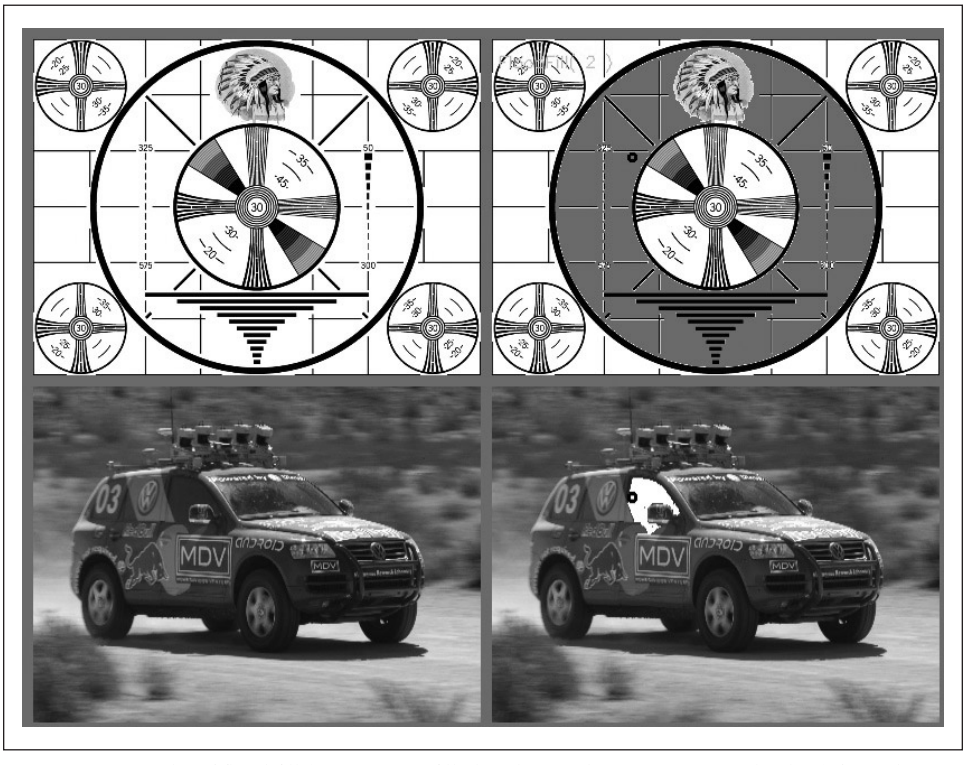

Figure 5-19. Results of flood fill (top image is filled with gray, bottom image with white) from the dark circle located just off center in both images; in this case, the hiDiff and loDiff parameters were each set to 7.0

If the flood-fill mask is set to be marked, then it is marked with the values set in the middle bits (8–15) of the `flags` value (see text). If these bits are not set then the mask is set to 1 as the default value. Don't be confused if you fill the mask and see nothing but black upon display; the filled values (if the middle bits of the flag weren't set) are 1s, so the mask image needs to be rescaled if you want to display it visually.

It's time to clarify the `flags` argument, which is tricky because it has three parts. The *low* 8 bits (0–7) can be set to 4 or 8. This controls the connectivity considered by the filling algorithm. If set to 4, only horizontal and vertical neighbors to the current pixel are considered in the filling process; if set to 8, flood fill will additionally include diagonal neighbors. The *high* 8 bits (16–23) can be set with the flags `CV_FLOODFILL_FIXED_RANGE` (fill relative to the seed point pixel value; otherwise, fill relative to the neighbor's value), and/or `CV_FLOODFILL_MASK_ONLY` (fill the mask location instead of the source image location). Obviously, you must supply an appropriate mask if `CV_FLOODFILL_MASK_ONLY` is set. The *middle* bits (8–15) of `flags` can be set to the value with which you want the mask to be filled. If the middle bits of `flags` are 0s, the mask will be filled with 1s. All these flags may be linked together via OR. For example, if you want an 8-way connectivity fill,

filling only a fixed range, filling the mask not the image, and filling using a value of 47, then the parameter to pass in would be:

```
flags = 8
      | CV_FLOODFILL_MASK_ONLY
      | CV_FLOODFILL_FIXED_RANGE
      | (47<<8);
```

Figure 5-20 shows flood fill in action on a sample image. Using `CV_FLOODFILL_FIXED_RANGE` with a wide range resulted in most of the image being filled (starting at the center). We should note that `newVal`, `loDiff`, and `upDiff` are prototyped as type `CvScalar` so they can be set for three channels at once (i.e., to encompass the RGB colors specified via `CV_RGB()`). For example, `lowDiff = CV_RGB(20,30,40)` will set `lowDiff` thresholds of 20 for red, 30 for green, and 40 for blue.

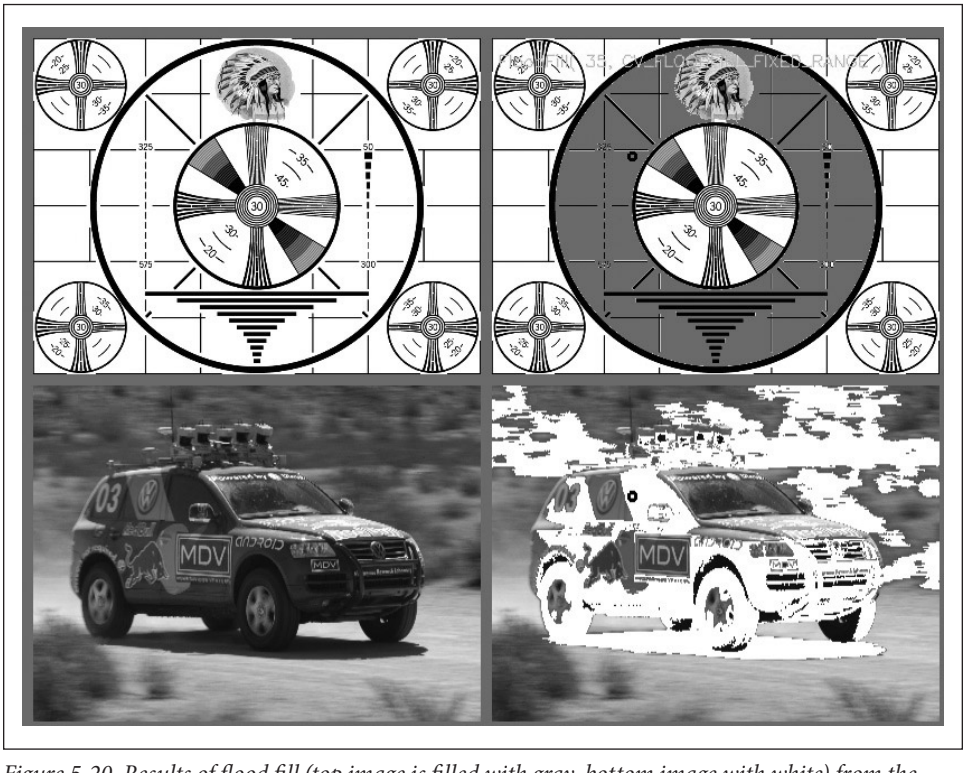

Figure 5-20. Results of flood fill (top image is filled with gray, bottom image with white) from the dark circle located just off center in both images; in this case, flood fill was done with a fixed range and with a high and low difference of 25.0

Resize

We often encounter an image of some size that we would like to convert to an image of some other size. We may want to upsize (zoom in) or downsize (zoom out) the image; we can accomplish either task by using `cvResize()`. This function will fit the source

image exactly to the destination image size. If the ROI is set in the source image then that ROI will be resized to fit in the destination image. Likewise, if an ROI is set in the destination image then the source will be resized to fit into the ROI.

```
void cvResize(
   const CvArr*   src,
   CvArr*         dst,
   int            interpolation = CV_INTER_LINEAR
);
```

The last argument is the interpolation method, which defaults to linear interpolation. The other available options are shown in Table 5-4.

Table 5-4. cvResize() interpolation options

Interpolation	Meaning
CV_INTER_NN	Nearest neighbor
CV_INTER_LINEAR	Bilinear
CV_INTER_AREA	Pixel area re-sampling
CV_INTER_CUBIC	Bicubic interpolation

In general, we would like the mapping from the source image to the resized destination image to be as smooth as possible. The argument `interpolation` controls exactly how this will be handled. Interpolation arises when we are shrinking an image and a pixel in the destination image falls in between pixels in the source image. It can also occur when we are expanding an image and need to compute values of pixels that do not directly correspond to any pixel in the source image. In either case, there are several options for computing the values of such pixels. The easiest approach is to take the resized pixel's value from its closest pixel in the source image; this is the effect of choosing the `interpolation` value `CV_INTER_NN`. Alternatively, we can linearly weight the 2-by-2 surrounding source pixel values according to how close they are to the destination pixel, which is what `CV_INTER_LINEAR` does. We can also virtually place the new resized pixel over the old pixels and then average the covered pixel values, as done with `CV_INTER_AREA`.* Finally, we have the option of fitting a cubic spline between the 4-by-4 surrounding pixels in the source image and then reading off the corresponding destination value from the fitted spline; this is the result of choosing the `CV_INTER_CUBIC` interpolation method.

Image Pyramids

Image pyramids [Adelson84] are heavily used in a wide variety of vision applications. An image pyramid is a collection of images—all arising from a single original image—that are successively downsampled until some desired stopping point is reached. (Of course, this stopping point could be a single-pixel image!)

* At least that's what happens when `cvResize()` shrinks an image. When it expands an image, `CV_INTER_AREA` amounts to the same thing as `CV_INTER_NN`.

There are two kinds of image pyramids that arise often in the literature and in application: the Gaussian [Rosenfeld80] and Laplacian [Burt83] pyramids [Adelson84]. The *Gaussian pyramid* is used to downsample images, and the Laplacian pyramid (to be discussed shortly) is required when we want to reconstruct an upsampled image from an image lower in the pyramid.

To produce layer (i+1) in the Gaussian pyramid (we denote this layer G_{i+1}) from layer G_i of the pyramid, we first convolve G_i with a Gaussian kernel and then remove every even-numbered row and column. Of course, from this it follows immediately that each image is exactly one-quarter the area of its predecessor. Iterating this process on the input image G_0 produces the entire pyramid. OpenCV provides us with a method for generating each pyramid stage from its predecessor:

```
void cvPyrDown(
   IplImage*   src,
   IplImage*   dst,
   IplFilter   filter = IPL_GAUSSIAN_5x5
);
```

Currently, the last argument `filter` supports only the single (default) option of a 5-by-5 Gaussian kernel.

Similarly, we can convert an existing image to an image that is twice as large in each direction by the following analogous (but not inverse!) operation:

```
void cvPyrUp(
   IplImage*   src,
   IplImage*   dst,
   IplFilter   filter = IPL_GAUSSIAN_5x5
);
```

In this case the image is first upsized to twice the original in each dimension, with the new (even) rows filled with 0s. Thereafter, a convolution is performed with the given filter (actually, a filter twice as large in each dimension than that specified*) to approximate the values of the "missing" pixels.

We noted previously that the operator `PyrUp()` is not the inverse of `PyrDown()`. This should be evident because `PyrDown()` is an operator that loses information. In order to restore the original (higher-resolution) image, we would require access to the information that was discarded by the downsampling. This data forms the *Laplacian pyramid*. The ith layer of the Laplacian pyramid is defined by the relation:

$$L_i = G_i - \mathrm{UP}(G_{i+1}) \otimes \mathcal{G}_{5\times5}$$

Here the operator `UP()` upsizes by mapping each pixel in location (x, y) in the original image to pixel ($2x + 1$, $2y + 1$) in the destination image; the $\otimes$ symbol denotes convolution; and $\mathcal{G}_{5\times5}$ is a 5-by-5 Gaussian kernel. Of course, $G_i - \mathrm{UP}(G_{i+1}) \otimes \mathcal{G}_{5\times5}$ is the definition

* This filter is also normalized to four, rather than to one. This is appropriate because the inserted rows have 0s in all of their pixels before the convolution.

of the PyrUp() operator provided by OpenCv. Hence, we can use OpenCv to compute the Laplacian operator directly as:

$$L_i = G_i - \text{PyrUp}(G_{i+1})$$

The Gaussian and Laplacian pyramids are shown diagrammatically in Figure 5-21, which also shows the inverse process for recovering the original image from the sub-images. Note how the Laplacian is really an approximation that uses the difference of Gaussians, as revealed in the preceding equation and diagrammed in the figure.

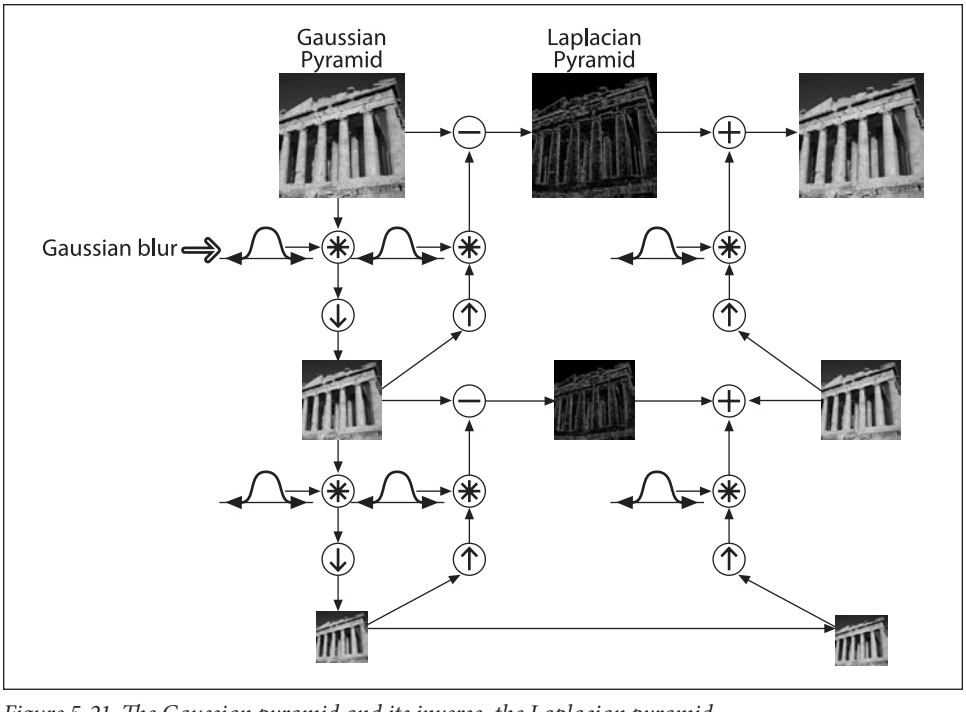

Figure 5-21. The Gaussian pyramid and its inverse, the Laplacian pyramid

There are many operations that can make extensive use of the Gaussian and Laplacian pyramids, but a particularly important one is image segmentation (see Figure 5-22). In this case, one builds an image pyramid and then associates to it a system of parent–child relations between pixels at level G_{i+1} and the corresponding reduced pixel at level G_i. In this way, a fast initial segmentation can be done on the low-resolution images high in the pyramid and then can be refined and further differentiated level by level.

This algorithm (due to B. Jaehne [Jaehne95; Antonisse82]) is implemented in OpenCV as cvPyrSegmentation():

```
void cvPyrSegmentation(
   IplImage*      src,
   IplImage*      dst,
```

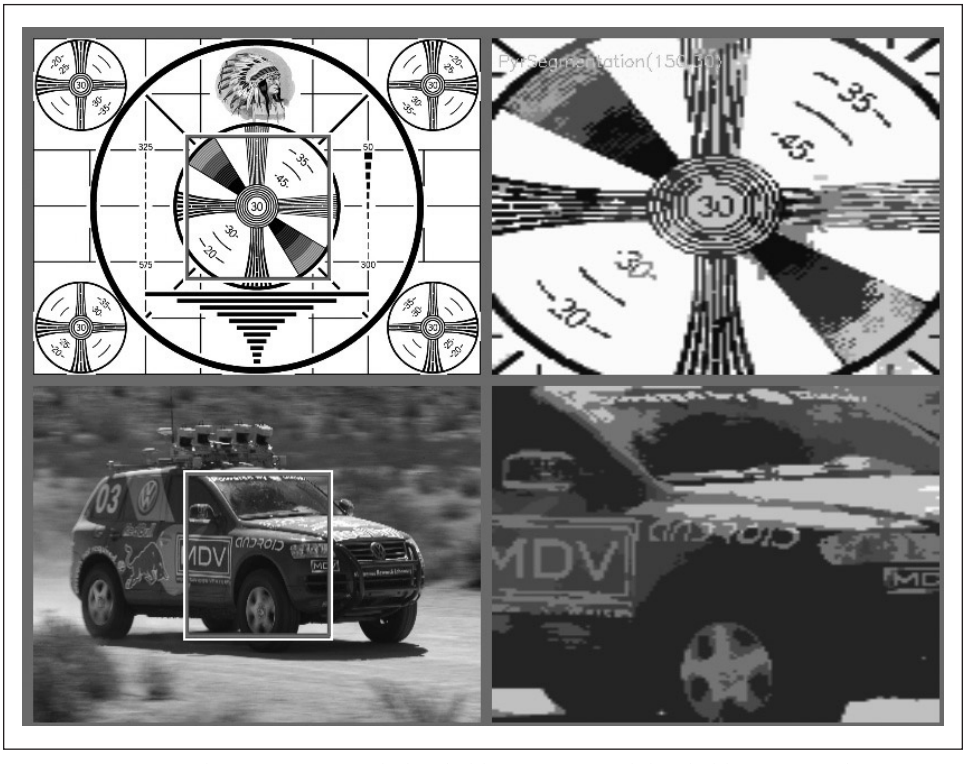

Figure 5-22. Pyramid segmentation with threshold1 set to 150 and threshold2 set to 30; the images on the right contain only a subsection of the images on the left because pyramid segmentation requires images that are N-times divisible by 2, where N is the number of pyramid layers to be computed (these are 512-by-512 areas from the original images)

```
        CvMemStorage*  storage,
        CvSeq**        comp,
        int            level,
        double         threshold1,
        double         threshold2
);
```

As usual, `src` and `dst` are the source and destination images, which must both be 8-bit, of the same size, and of the same number of channels (one or three). You might be wondering, "What destination image?" Not an unreasonable question, actually. The destination image `dst` is used as scratch space for the algorithm and also as a return visualization of the segmentation. If you view this image, you will see that each segment is colored in a single color (the color of some pixel in that segment). Because this image is the algorithm's scratch space, you cannot simply set it to `NULL`. Even if you do not want the result, you must provide an image. One important word of warning about `src` and `dst`: because all levels of the image pyramid must have integer sizes in both dimensions, the starting images must be divisible by two as many times as there are levels in the

pyramid. For example, for a four-level pyramid, a height or width of 80 ($2 \times 2 \times 2 \times 5$) would be acceptable, but a value of 90 ($2 \times 3 \times 3 \times 5$) would not.*

The pointer `storage` is for an OpenCV memory storage area. In Chapter 8 we will discuss such areas in more detail, but for now you should know that such a storage area is allocated with a command like†

```
CvMemStorage* storage = cvCreateMemStorage();
```

The argument `comp` is a location for storing further information about the resulting segmentation: a sequence of connected components is allocated from this memory storage. Exactly how this works will be detailed in Chapter 8, but for convenience here we briefly summarize what you'll need in the context of `cvPyrSegmentation()`.

First of all, a *sequence* is essentially a list of structures of a particular kind. Given a sequence, you can obtain the number of elements as well as a particular element if you know both its type and its number in the sequence. Take a look at the Example 5-1 approach to accessing a sequence.

Example 5-1. Doing something with each element in the sequence of connected components returned by cvPyrSegmentation()

```
void f(
  IplImage* src,
  IplImage* dst
) {
  CvMemStorage* storage = cvCreateMemStorage(0);
  CvSeq* comp = NULL;
  cvPyrSegmentation( src, dst, storage, &comp, 4, 200, 50 );
  int n_comp = comp->total;
  for( int i=0; i<n_comp; i++ ) {
    CvConnectedComp* cc = (CvConnectedComp*) cvGetSeqElem( comp, i );
    do_something_with( cc );
  }
  cvReleaseMemStorage( &storage );
}
```

There are several things you should notice in this example. First, observe the allocation of a *memory storage*; this is where `cvPyrSegmentation()` will get the memory it needs for the connected components it will have to create. Then the pointer `comp` is allocated as type `CvSeq*`. It is initialized to `NULL` because its current value means nothing. We will pass to `cvPyrSegmentation()` a pointer to `comp` so that `comp` can be set to the location of the sequence created by `cvPyrSegmentation()`. Once we have called the segmentation, we can figure out how many elements there are in the sequence with the member element `total`. Thereafter we can use the generic `cvGetSeqElem()` to obtain the *i*th element of `comp`; however, because `cvGetSeqElem()` is generic and returns only a void pointer, we must cast the return pointer to the appropriate type (in this case, `CvConnectedComp*`).

* Heed this warning! Otherwise, you will get a totally useless error message and probably waste hours trying to figure out what's going on.

† Actually, the current implementation of `cvPyrSegmentation()` is a bit incomplete in that it returns not the computed segments but only the bounding rectangles (as `CvSeq<CvConnectedComp>`).

Finally, we need to know that a connected component is one of the basic structure types in OpenCV. You can think of it as a way of describing a "blob" in an image. It has the following definition:

```
typedef struct CvConnectedComponent {
  double   area;
  CvScalar value;
  CvRect   rect;
  CvSeq*   contour;
};
```

The `area` is the area of the component. The `value` is the average color* over the area of the component and `rect` is a bounding box for the component (defined in the coordinates of the parent image). The final element, `contour`, is a pointer to another sequence. This sequence can be used to store a representation of the boundary of the component, typically as a sequence of points (type `CvPoint`).

In the specific case of `cvPyrSegmentation()`, the contour member is not set. Thus, if you want some specific representation of the component's pixels then you will have to compute it yourself. The method to use depends, of course, on the representation you have in mind. Often you will want a Boolean mask with nonzero elements wherever the component was located. You can easily generate this by using the `rect` portion of the connected component as a mask and then using `cvFloodFill()` to select the desired pixels inside of that rectangle.

Threshold

Frequently we have done many layers of processing steps and want either to make a final decision about the pixels in an image or to categorically reject those pixels below or above some value while keeping the others. The OpenCV function `cvThreshold()` accomplishes these tasks (see survey [Sezgin04]). The basic idea is that an array is given, along with a threshold, and then something happens to every element of the array depending on whether it is below or above the threshold.

```
double cvThreshold(
   CvArr*         src,
   CvArr*         dst,
   double         threshold,
   double         max_value,
   int            threshold_type
);
```

As shown in Table 5-5, each threshold type corresponds to a particular comparison operation between the ith source pixel (src_i) and the threshold (denoted in the table by T). Depending on the relationship between the source pixel and the threshold, the destination pixel dst_i may be set to 0, the src_i, or the `max_value` (denoted in the table by M).

* Actually the meaning of `value` is context dependant and could be just about anything, but it is typically a color associated with the component. In the case of `cvPyrSegmentation()`, `value` is the average color over the segment.

CHAPTER 6

Image Transforms

Overview

In the previous chapter we covered a lot of different things you could do with an image. The majority of the operators presented thus far are used to enhance, modify, or otherwise "process" one image into a similar but new image.

In this chapter we will look at *image transforms*, which are methods for changing an image into an alternate representation of the data entirely. Perhaps the most common example of a transform would be a something like a *Fourier transform*, in which the image is converted to an alternate representation of the data in the original image. The result of this operation is still stored in an OpenCV "image" structure, but the individual "pixels" in this new image represent spectral components of the original input rather than the spatial components we are used to thinking about.

There are a number of useful transforms that arise repeatedly in computer vision. OpenCV provides complete implementations of some of the more common ones as well as building blocks to help you implement your own image transforms.

Convolution

Convolution is the basis of many of the transformations that we discuss in this chapter. In the abstract, this term means something we do to every part of an image. In this sense, many of the operations we looked at in Chapter 5 can also be understood as special cases of the more general process of convolution. What a particular convolution "does" is determined by the form of the *Convolution kernel* being used. This kernel is essentially just a fixed size array of numerical coefficients along with an *anchor point* in that array, which is typically located at the center. The size of the array* is called the *support* of the kernel.

Figure 6-1 depicts a 3-by-3 convolution kernel with the anchor located at the center of the array. The value of the convolution at a particular point is computed by first placing

* For technical purists, the support of the kernel actually consists of only the nonzero portion of the kernel array.

the kernel anchor on top of a pixel on the image with the rest of the kernel overlaying the corresponding local pixels in the image. For each kernel point, we now have a value for the kernel at that point and a value for the image at the corresponding image point. We multiply these together and sum the result; this result is then placed in the resulting image at the location corresponding to the location of the anchor in the input image. This process is repeated for every point in the image by scanning the kernel over the entire image.

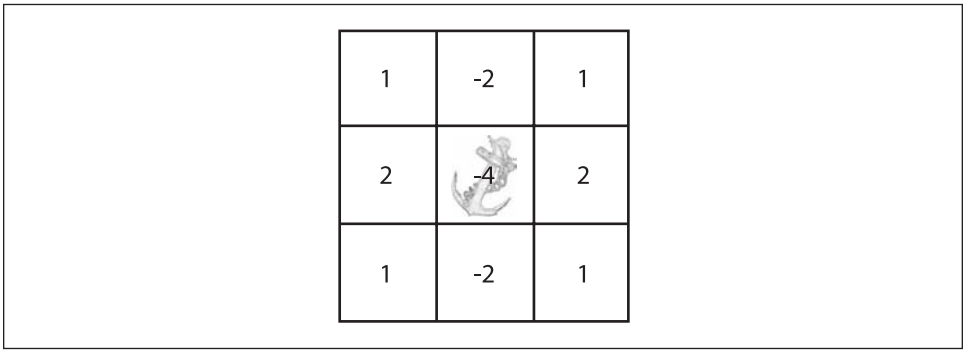

Figure 6-1. A 3-by-3 kernel for a Sobel derivative; note that the anchor point is in the center of the kernel

We can, of course, express this procedure in the form of an equation. If we define the image to be $I(x, y)$, the kernel to be $G(i, j)$ (where $0 < i < M_i - 1$ and $0 < j < M_j - 1$), and the anchor point to be located at (a_i, a_j) in the coordinates of the kernel, then the convolution $H(x, y)$ is defined by the following expression:

$$H(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j) G(i,j)$$

Observe that the number of operations, at least at first glance, seems to be the number of pixels in the image multiplied by the number of pixels in the kernel.* This can be a lot of computation and so is not something you want to do with some "for" loop and a lot of pointer de-referencing. In situations like this, it is better to let OpenCV do the work for you and take advantage of the optimizations already programmed into OpenCV. The OpenCV way to do this is with `cvFilter2D()`:

```
void cvFilter2D(
   const CvArr*     src,
   CvArr*           dst,
   const CvMat*     kernel,
```

* We say "at first glance" because it is also possible to perform convolutions in the frequency domain. In this case, for an N-by-N image and an M-by-M kernel with $N > M$, the computational time will be proportional to $N^2 \log(N)$ and not to the N^2M^2 that is expected for computations in the spatial domain. Because the frequency domain computation is independent of the size of the kernel, it is more efficient for large kernels. OpenCV automatically decides whether to do the convolution in the frequency domain based on the size of the kernel.

```
    CvPoint          anchor = cvPoint(-1,-1)
);
```

Here we create a matrix of the appropriate size, fill it with the coefficients, and then pass it together with the source and destination images into `cvFilter2D()`. We can also optionally pass in a `CvPoint` to indicate the location of the center of the kernel, but the default value (equal to `cvPoint(-1,-1)`) is interpreted as indicating the center of the kernel. The kernel can be of even size if its anchor point is defined; otherwise, it should be of odd size.

The `src` and `dst` images should be the same size. One might think that the `src` image should be larger than the `dst` image in order to allow for the extra width and length of the convolution kernel. But the sizes of the `src` and `dst` can be the same in OpenCV because, by default, prior to convolution OpenCV creates virtual pixels via replication past the border of the `src` image so that the border pixels in `dst` can be filled in. The replication is done as input($-dx$, y) = input(0, y), input($w + dx$, y) = input($w - 1$, y), and so forth. There are some alternatives to this default behavior; we will discuss them in the next section.

We remark that the coefficients of the convolution kernel should always be floating-point numbers. This means that you should use `CV_32FC1` when allocating that matrix.

Convolution Boundaries

One problem that naturally arises with convolutions is how to handle the boundaries. For example, when using the convolution kernel just described, what happens when the point being convolved is at the edge of the image? Most of OpenCV's built-in functions that make use of `cvFilter2D()` must handle this in one way or another. Similarly, when doing your own convolutions, you will need to know how to deal with this efficiently.

The solution comes in the form of the `cvCopyMakeBorder()` function, which copies a given image onto another slightly larger image and then automatically pads the boundary in one way or another:

```
void cvCopyMakeBorder(
    const CvArr*    src,
    CvArr*          dst,
    CvPoint         offset,
    int             bordertype,
    CvScalar        value     = cvScalarAll(0)
);
```

The `offset` argument tells `cvCopyMakeBorder()` where to place the copy of the original image within the destination image. Typically, if the kernel is N-by-N (for odd N) then you will want a boundary that is $(N - 1)/2$ wide on all sides or, equivalently, an image that is $N - 1$ wider and taller than the original. In this case you would set the offset to `cvPoint((N-1)/2,(N-1)/2)` so that the boundary would be even on all sides.*

* Of course, the case of N-by-N with N odd and the anchor located at the center is the simplest case. In general, if the kernel is N-by-M and the anchor is located at (a_x, a_y), then the destination image will have to be $N - 1$ pixels wider and $M - 1$ pixels taller than the source image. The offset will simply be (a_x, a_y).

The `bordertype` can be either `IPL_BORDER_CONSTANT` or `IPL_BORDER_REPLICATE` (see Figure 6-2). In the first case, the `value` argument will be interpreted as the value to which all pixels in the boundary should be set. In the second case, the row or column at the very edge of the original is replicated out to the edge of the larger image. Note that the border of the test pattern image is somewhat subtle (examine the upper right image in Figure 6-2); in the test pattern image, there's a one-pixel-wide dark border except where the circle patterns come near the border where it turns white. There are two other border types defined, `IPL_BORDER_REFLECT` and `IPL_BORDER_WRAP`, which are not implemented at this time in OpenCV but may be supported in the future.

Figure 6-2. Expanding the image border. The left column shows IPL_BORDER_CONSTANT where a zero value is used to fill out the borders. The right column shows IPL_BORDER_REPLICATE where the border pixels are replicated in the horizontal and vertical directions

We mentioned previously that, when you make calls to OpenCV library functions that employ convolution, those library functions call `cvCopyMakeBorder()` to get their work done. In most cases the border type called is `IPL_BORDER_REPLICATE`, but sometimes you will not want it to be done that way. This is another occasion where you might want to use `cvCopyMakeBorder()`. You can create a slightly larger image with the border you want, call whatever routine on that image, and then clip back out the part you were originally interested in. This way, OpenCV's automatic bordering will not affect the pixels you care about.

Gradients and Sobel Derivatives

One of the most basic and important convolutions is the computation of derivatives (or approximations to them). There are many ways to do this, but only a few are well suited to a given situation.

In general, the most common operator used to represent differentiation is the *Sobel derivative* [Sobel68] operator (see Figures 6-3 and 6-4). Sobel operators exist for any order of derivative as well as for mixed partial derivatives (e.g., $\partial^2/\partial x \partial y$).

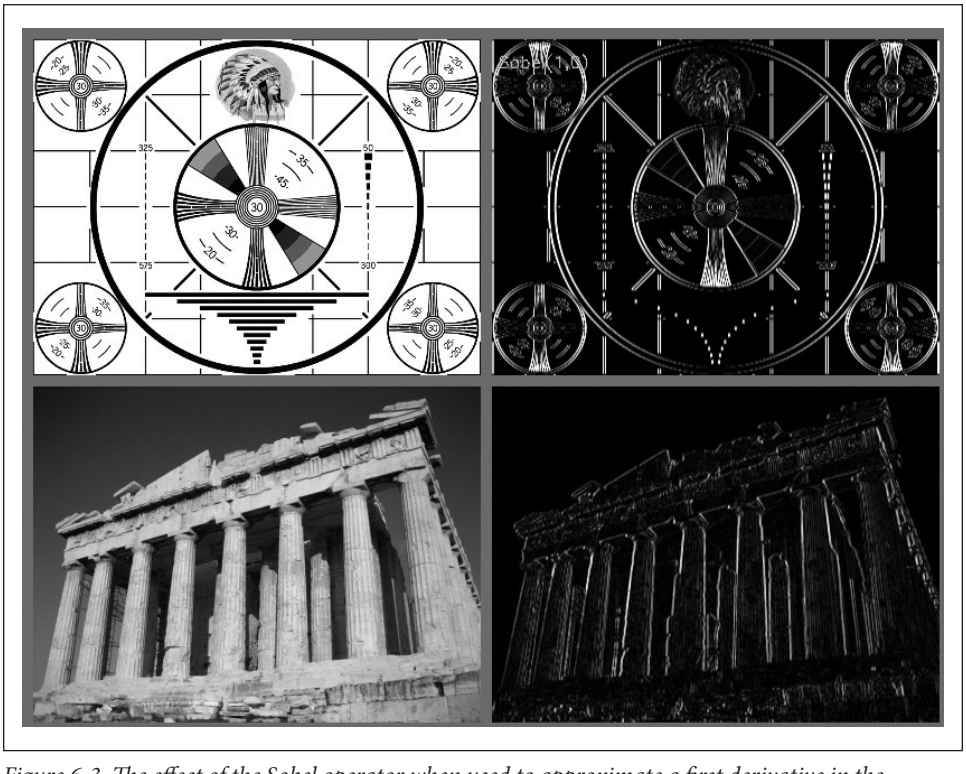

Figure 6-3. The effect of the Sobel operator when used to approximate a first derivative in the x-dimension

```
cvSobel(
      const CvArr* src,
      CvArr*       dst,
      int          xorder,
      int          yorder,
      int          aperture_size = 3
);
```

Here, `src` and `dst` are your image input and output, and `xorder` and `yorder` are the orders of the derivative. Typically you'll use 0, 1, or at most 2; a 0 value indicates no derivative

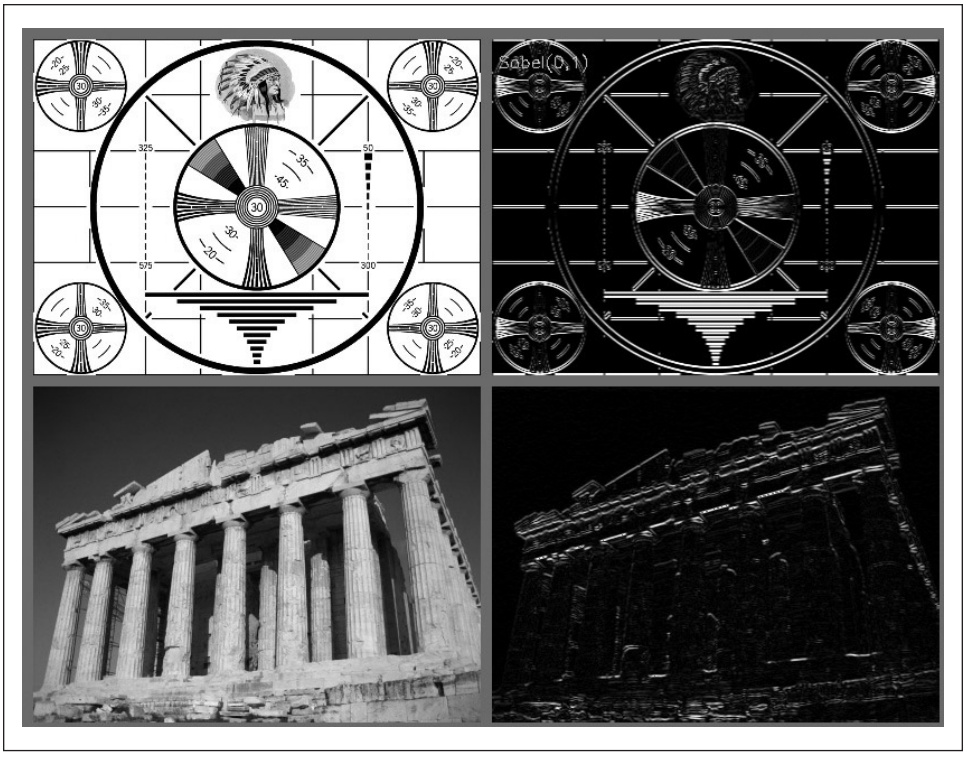

Figure 6-4. The effect of the Sobel operator when used to approximate a first derivative in the y-dimension

in that direction.* The `aperture_size` parameter should be odd and is the width (and the height) of the square filter. Currently, `aperture_sizes` of 1, 3, 5, and 7 are supported. If `src` is 8-bit then the `dst` must be of depth `IPL_DEPTH_16S` to avoid overflow.

Sobel derivatives have the nice property that they can be defined for kernels of any size, and those kernels can be constructed quickly and iteratively. The larger kernels give a better approximation to the derivative because the smaller kernels are very sensitive to noise.

To understand this more exactly, we must realize that a Sobel derivative is not really a derivative at all. This is because the Sobel operator is defined on a discrete space. What the Sobel operator actually represents is a fit to a polynomial. That is, the Sobel derivative of second order in the *x*-direction is not really a second derivative; it is a local fit to a parabolic function. This explains why one might want to use a larger kernel: that larger kernel is computing the fit over a larger number of pixels.

* Either `xorder` or `yorder` must be nonzero.

Scharr Filter

In fact, there are many ways to approximate a derivative in the case of a discrete grid. The downside of the approximation used for the Sobel operator is that it is less accurate for small kernels. For large kernels, where more points are used in the approximation, this problem is less significant. This inaccuracy does not show up directly for the *X* and *Y* filters used in `cvSobel()`, because they are exactly aligned with the *x*- and *y*-axes. The difficulty arises when you want to make image measurements that are approximations of *directional derivatives* (i.e., direction of the image gradient by using the arctangent of the *y*/*x* filter responses).

To put this in context, a concrete example of where you may want image measurements of this kind would be in the process of collecting shape information from an object by assembling a histogram of gradient angles around the object. Such a histogram is the basis on which many common shape classifiers are trained and operated. In this case, inaccurate measures of gradient angle will decrease the recognition performance of the classifier.

For a 3-by-3 Sobel filter, the inaccuracies are more apparent the further the gradient angle is from horizontal or vertical. OpenCV addresses this inaccuracy for small (but fast) 3-by-3 Sobel derivative filters by a somewhat obscure use of the special `aperture_size` value `CV_SCHARR` in the `cvSobel()` function. The Scharr filter is just as fast but more accurate than the Sobel filter, so it should always be used if you want to make image measurements using a 3-by-3 filter. The filter coefficients for the Scharr filter are shown in Figure 6-5 [Scharr00].

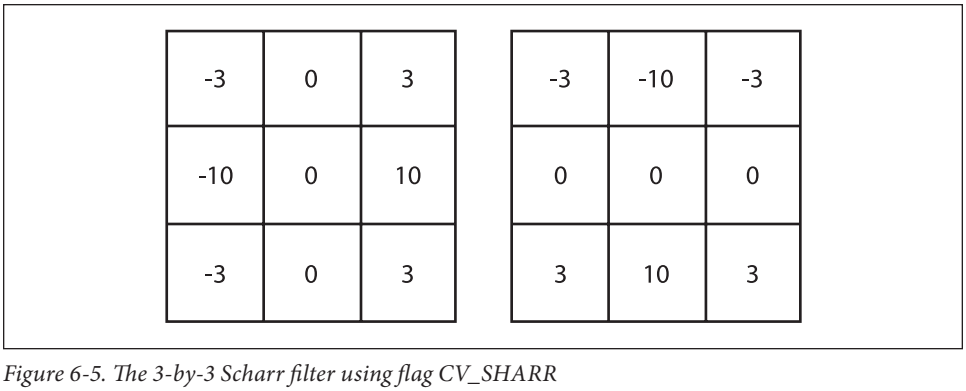

Figure 6-5. The 3-by-3 Scharr filter using flag CV_SHARR

Laplace

The OpenCV *Laplacian* function (first used in vision by Marr [Marr82]) implements a discrete analog of the Laplacian operator:*

* Note that the Laplacian operator is completely distinct from the Laplacian pyramid of Chapter 5.

$$\text{Laplace}(f) \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Because the Laplacian operator can be defined in terms of second derivatives, you might well suppose that the discrete implementation works something like the second-order Sobel derivative. Indeed it does, and in fact the OpenCV implementation of the Laplacian operator uses the Sobel operators directly in its computation.

```
void cvLaplace(
  const CvArr* src,
  CvArr*       dst,
  int          apertureSize = 3
);
```

The `cvLaplace()` function takes the usual source and destination images as arguments as well as an aperture size. The source can be either an 8-bit (unsigned) image or a 32-bit (floating-point) image. The destination must be a 16-bit (signed) image or a 32-bit (floating-point) image. This aperture is precisely the same as the aperture appearing in the Sobel derivatives and, in effect, gives the size of the region over which the pixels are sampled in the computation of the second derivatives.

The Laplace operator can be used in a variety of contexts. A common application is to detect "blobs." Recall that the form of the Laplacian operator is a sum of second derivatives along the *x*-axis and *y*-axis. This means that a single point or any small blob (smaller than the aperture) that is surrounded by higher values will tend to maximize this function. Conversely, a point or small blob that is surrounded by lower values will tend to maximize the negative of this function.

With this in mind, the Laplace operator can also be used as a kind of edge detector. To see how this is done, consider the first derivative of a function, which will (of course) be large wherever the function is changing rapidly. Equally important, it will grow rapidly as we approach an edge-like discontinuity and shrink rapidly as we move past the discontinuity. Hence the derivative will be at a local maximum somewhere within this range. Therefore we can look to the 0s of the second derivative for locations of such local maxima. Got that? Edges in the original image will be 0s of the Laplacian. Unfortunately, both substantial and less meaningful edges will be 0s of the Laplacian, but this is not a problem because we can simply filter out those pixels that also have larger values of the first (Sobel) derivative. Figure 6-6 shows an example of using a Laplacian on an image together with details of the first and second derivatives and their zero crossings.

Canny

The method just described for finding edges was further refined by J. Canny in 1986 into what is now commonly called the *Canny edge detector* [Canny86]. One of the differences between the Canny algorithm and the simpler, Laplace-based algorithm from the previous section is that, in the Canny algorithm, the first derivatives are computed in *x* and *y* and then combined into four directional derivatives. The points where these directional derivatives are local maxima are then candidates for assembling into edges.

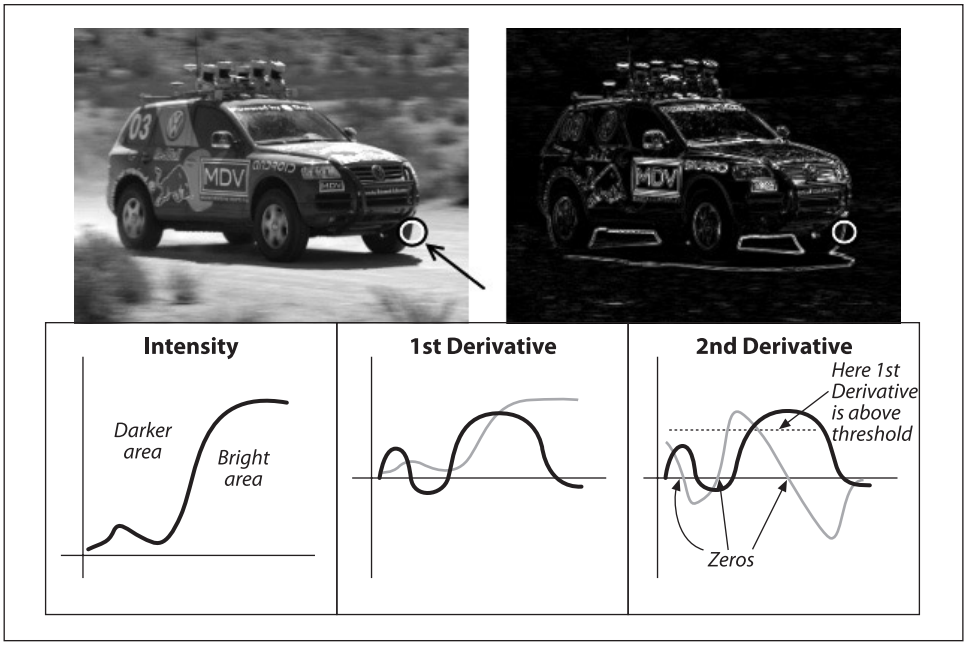

Figure 6-6. Laplace transform (upper right) of the racecar image: zooming in on the tire (circled in white) and considering only the x-dimension, we show a (qualitative) representation of the brightness as well as the first and second derivative (lower three cells); the 0s in the second derivative correspond to edges, and the 0 corresponding to a large first derivative is a strong edge

However, the most significant new dimension to the Canny algorithm is that it tries to assemble the individual edge candidate pixels into *contours*.* These contours are formed by applying an *hysteresis threshold* to the pixels. This means that there are two thresholds, an upper and a lower. If a pixel has a gradient larger than the upper threshold, then it is accepted as an edge pixel; if a pixel is below the lower threshold, it is rejected. If the pixel's gradient is between the thresholds, then it will be accepted only if it is connected to a pixel that is above the high threshold. Canny recommended a ratio of high:low threshold between 2:1 and 3:1. Figures 6-7 and 6-8 show the results of applying `cvCanny()` to a test pattern and a photograph using high:low hysteresis threshold ratios of 5:1 and 3:2, respectively.

```
void cvCanny(
   const CvArr* img,
   CvArr*       edges,
   double       lowThresh,
   double       highThresh,
   int          apertureSize = 3
);
```

* We'll have much more to say about contours later. As you await those revelations, though, keep in mind that the `cvCanny()` routine does not actually return objects of type `CvContour`; we will have to build those from the output of `cvCanny()` if we want them by using `cvFindContours()`. Everything you ever wanted to know about contours will be covered in Chapter 8.

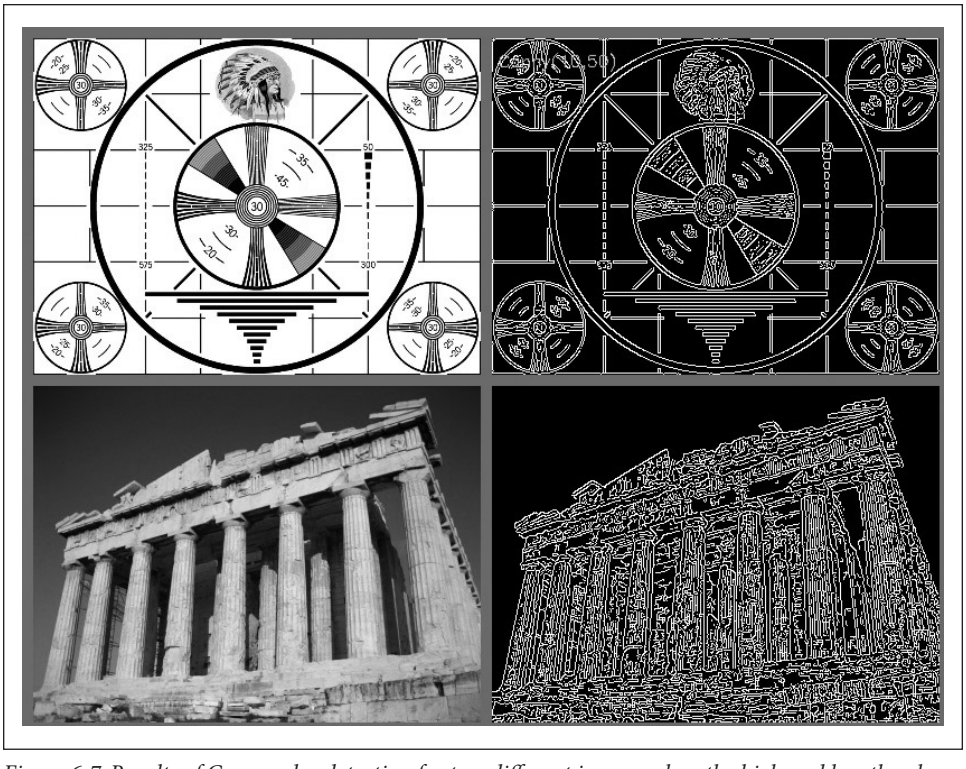

Figure 6-7. Results of Canny edge detection for two different images when the high and low thresholds are set to 50 and 10, respectively

The cvCanny() function expects an input image, which must be grayscale, and an output image, which must also be grayscale (but which will actually be a Boolean image). The next two arguments are the low and high thresholds, and the last argument is another aperture. As usual, this is the aperture used by the Sobel derivative operators that are called inside of the implementation of cvCanny().

Hough Transforms

The *Hough transform** is a method for finding lines, circles, or other simple forms in an image. The original Hough transform was a line transform, which is a relatively fast way of searching a binary image for straight lines. The transform can be further generalized to cases other than just simple lines.

Hough Line Transform

The basic theory of the Hough line transform is that any point in a binary image could be part of some set of possible lines. If we parameterize each line by, for example, a

* Hough developed the transform for use in physics experiments [Hough59]; its use in vision was introduced by Duda and Hart [Duda72].