

# Advanced Edge Detection Techniques:

## The Canny and the Shen-Castan Methods

---

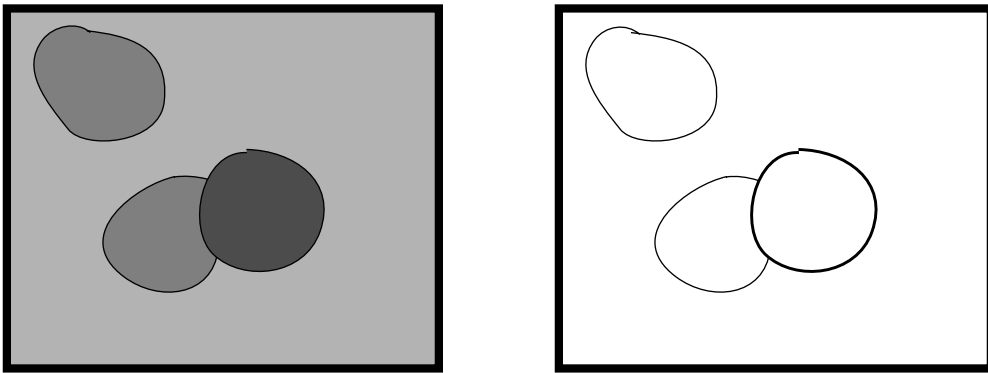
### 1.1 The Purpose of Edge Detection

---

Edge detection is one of the most commonly used operations in image analysis, and there are probably more algorithms in the literature for enhancing and detecting edges than any other single subject. The reason for this is that edges form the outline of an object. An edge is the boundary between an object and the background, and indicates the boundary between overlapping objects. This means that if the edges in an image can be identified accurately, all of the objects can be located and basic properties such as area, perimeter, and shape can be measured. Since computer vision involves the identification and classification of objects in an image, edge detection is an essential tool.

A straightforward example of edge detection is illustrated in Figure 1.1. There are two overlapping objects in the original picture (a), which has a uniform grey background. The edge enhanced version of the same image (b) has dark lines outlining the three objects. Note that there is no way to tell which parts of the image are background and which are object; only the boundaries between the regions are identified. However, given that the blobs in the image are the regions, it can be determined that the blob numbered '3' covers up a part of blob '2', and is therefore closer to the camera.

Edge detection is part of a process called *segmentation* - the identification of regions within an image. The regions that may be objects in Figure 1



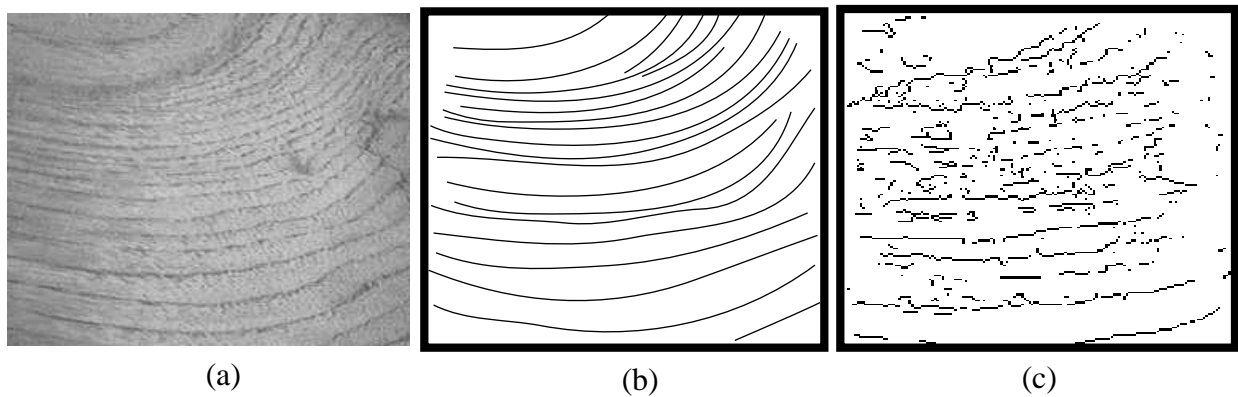
**FIGURE 1.1** - Example of edge detection. (a) Synthetic image with blobs on a grey background. (b) Edge enhanced image showing only the outlines of the objects.

have been isolated, and further processing may determine what kind of object each region represents. While in this example edge detection is merely a step in the segmentation process, it is sometimes all that is needed, especially when the objects in an image are lines.

Consider the image in Figure 1.2, which is a photograph of a cross-section of a tree. The growth rings are the objects of interest in this image. Each ring represents a year of the tree's life, and the number of rings is therefore the same as the age of the tree. Enhancing the rings using an edge detector, as shown in Figure 1.2b, is all that is needed to segment the image into foreground (objects = rings) and background (everything else).

Technically, *edge detection* is the process of locating the edge pixels, and *edge enhancement* will increase the contrast between the edges and the background so that the edges become more visible. In practice the terms are used interchangeably, since most edge detection programs also set the edge pixel values to a specific grey level or color so that they can be easily seen. In addition, *edge tracing* is the process of following the edges, usually collecting the edge pixels into a list. This is done in a consistent direction, either clockwise or counter-clockwise around the objects. Chain coding is one example of a method of edge tracing. The result is a non-raster representation of the objects which can be used to compute shape measures or otherwise identify or classify the object.

The remainder of this chapter will discuss the theory of edge detection, including a few traditional methods. Then two methods of special interest will be described and compared. These methods, the Canny edge detector and the Shen-Castan, or ISEF, edge detector have received a lot of atten-



**FIGURE 1.2** - A cross section of a tree. (a) Original grey-level image. (b) Ideal edge enhanced image, showing the growth rings. (c) The edge enhancement that one might expect using a real algorithm.

tion lately, and justifiably so. Both are based solidly on theoretical considerations, and both claim a degree of optimality; that is, both claim to be the best that can be done under certain specified circumstances. These claims will be examined, both in theory and in practice.

---

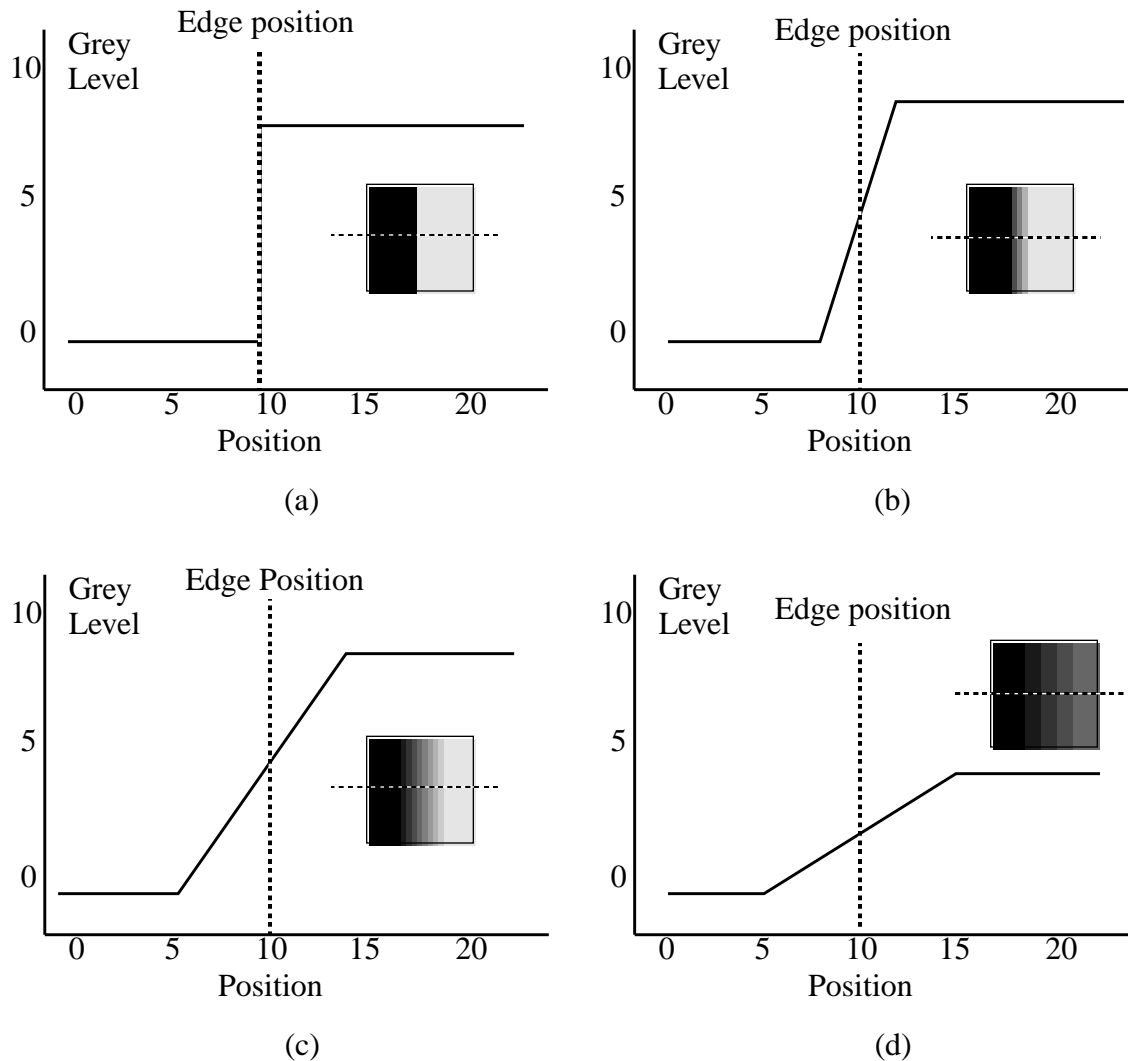
## 1.2 Traditional Approaches and Theory

---

Most good algorithms begin with a clear statement of the problem to be solved, and a cogent analysis of the possible methods of solution and the conditions under which the methods will operate correctly. Using this paradigm to define an edge detection algorithm means first defining what an edge is, then using this definition to suggest methods of enhancement.

As usual, there are a number of possible definitions of an edge, each being applicable in various specific circumstances. One of the most common and most general definitions is the *ideal step edge*, illustrated in Figure 1.3. In this one-dimensional example, the edge is simply a change in grey level occurring at one specific location. The greater the change in level the easier the edge is to detect, but in the ideal case *any* level change can be seen quite easily.

The first complication occurs because of digitization. It is unlikely that the image will be sampled in such a way that all of the edges happen to correspond exactly with a pixel boundary. Indeed, the change in level may extend across some number of pixels (Figures 1.3b-d). The actual position of the edge is considered to be the center of the *ramp* connecting the low grey level to the high one. This is a ramp in the mathematical



**FIGURE 1.3** - Step edges. (a) The change in level occurs exactly at pixel 10. (b) The same level change as before, but over 4 pixels centered at pixel 10. This is a *ramp* edge. (c) Same level change but over 10 pixels, centered at 10. (d) A smaller change over 10 pixels. The insert shows the way the image would appear, and the dotted line shows where the image was sliced to give the illustrated cross-section.

world only, since after the image has been made digital (sampled) the ramp has the jagged appearance of a staircase.

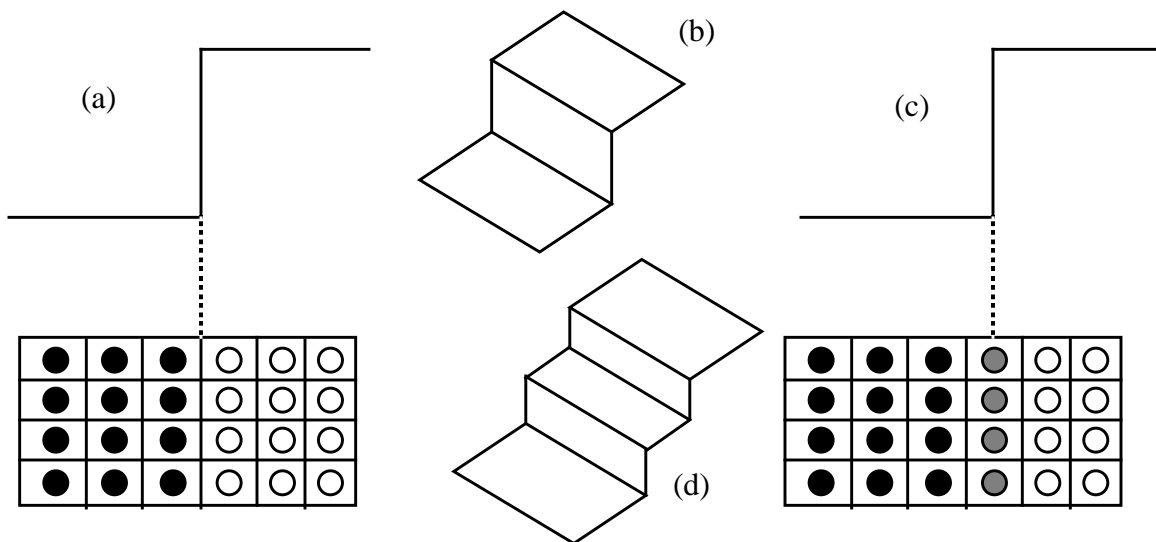
The second complication is the ubiquitous problem of *noise*. Due to a great many factors such as light intensity, type of camera and lens, motion, temperature, atmospheric effects, dust, and others, it is very unlikely that two pixels that correspond to precisely the same grey level in the scene will have the same level in the image. Noise is a random

effect, and is characterizable only statistically. The result of noise on the image is to produce a random variation in level from pixel to pixel, and so the smooth lines and ramps of the ideal edges are never encountered in real images.

### 1.2.1 Models of Edges

The step edge of Figure 1.3a is ideal because it is easy to detect: in the absence of noise, any significant change in grey level would indicate an edge. A step edge never really occurs in an image because: a) objects rarely have such a sharp outline; b) a scene is never sampled so that edges occur exactly at the margin of a pixel; and c) due to noise, as mentioned previously.

Noise will be discussed in the next section, and object outlines vary quite a bit from image to image, so let us concentrate for a moment on sampling. Figure 1.4a shows an ideal step edge and the set of pixels involved. Note that the edge occurs on the extreme left side of the white edge pixels. As the camera moves to the left by amounts smaller than one pixel width the edge moves to the right. In Figure 1.4c the edge has moved by one half of a pixel, and the pixels along the edge now contain some part of the image that is black and some part that is white. This will be reflected in the grey level as a weighted average:



**FIGURE 1.4** - The effect of sampling on a step edge. (a) An ideal step edge. (b) Three dimensional view of the step edge. (c) Step edge sampled at the center of a pixel, instead of on a margin. (d) The result, in three dimensions, has the appearance of a staircase.

$$v = \frac{(v_w a_w + v_b a_b)}{a_w + a_b} \quad (\text{EQ 1.1})$$

where  $v_w$  and  $v_b$  are the grey levels of the white and black regions, and  $a_w$  and  $a_b$  are the areas of the white and black parts of the edge pixel. For example, if the white level is 100 and the black level is 0, then the value of an edge pixel for which the edge runs through the middle will be 50. The result is a double step instead of a step edge.

If the effect of a blurred outline is to spread out the grey level change over a number of pixels then the single stair becomes a *staircase*. The ramp is a model of what the edge must have originally looked like in order to produce a staircase, and so is an idealization, an interpolation of the data actually encountered.

Although the ideal step edge and ramp edge models were generally used to devise new edge detectors in the past, the model was recognized to be a simplification, and newer edge detection schemes incorporate noise into the model and are tested on staircases and noisy edges.

### 1.2.2 Noise

All image acquisition processes are subject to noise of some type, so there is little point in ignoring it; the ideal situation of no noise never occurs in practice. Noise cannot be predicted accurately because of its random nature, and cannot even be measured accurately from a noisy image, since the contribution to the grey levels of the noise can't be distinguished from the pixel data. However, noise can sometimes be characterized by its effect on the image, and is usually expressed as a probability distribution with a specific mean and standard deviation.

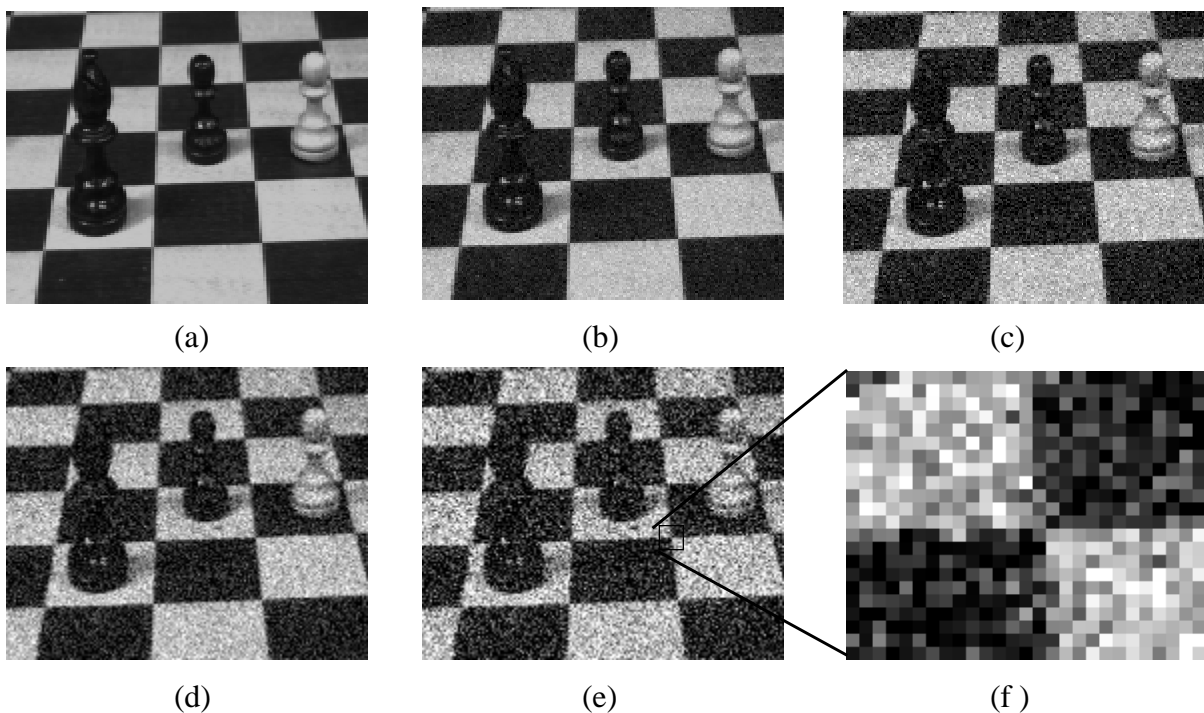
There are two types of noise that are of specific interest in image analysis. *Signal independent* noise is a random set of greys levels, statistically independent of the image data, that is added to the pixels in the image to give the resulting noisy image. This kind of noise occurs when an image is transmitted electronically from one place to another. If  $A$  is a perfect image and  $N$  is the noise that occurs during transmission, then the final image  $B$  is:

$$B = A + N \quad (\text{EQ 1.2})$$

A and N are unrelated to each other. The noise image N could have any statistical properties, but a common assumption is that it follows the Normal distribution with a mean of zero and some measured or presumed standard deviation.

It is a simple matter to create an artificially noisy image having known characteristics, and such images are very useful tools for experimenting with edge detection algorithms. Figure 1.5 shows an image of a chess board that has been subjected to various degrees of artificial noise. For a Normal distribution with zero mean the amount of noise is specified by the standard deviation; values of 10, 20, 30, and 50 are shown in the figure.

For these images it is possible to obtain an estimate of the noise. The scene contains a number of small regions that should have a uniform grey level - the squares on the chess board. If the noise is consistent over the entire image, then the noise in any one square will be a sample of the noise in the whole image, and since the level is constant over the square then any variation can be assumed to be caused by the noise alone. In this



**FIGURE 1.5** - Normally distributed noise and its effect on an image. (a) Original image. (b) Noise having  $\sigma = 10$ . (c) Noise having  $\sigma = 20$ . (d) Noise having  $\sigma = 30$ . (e) Noise having  $\sigma = 50$ . (f) Expanded view of an intersection of four regions in the  $\sigma = 50$  image.

case, the mean and standard deviation of the grey levels in any square can be computed; the standard deviation of the grey levels will be close to that of the noise. To make sure that this is working properly, we can now use the mean already computed as the grey level of the square and compute the mean and standard deviation of the *difference of each grey level from the mean*; this new mean should be near to zero, and the standard deviation close to that of that noise (and to the previously computed standard deviation).

A program that does this appears in Figure 1.6. As a simple test, a black square and a white square were isolated from the image in Figure 1.5c and this program was used to estimate the noise. The results were:

Black region:

Image mean is 31.63629 Standard deviation is 19.52933

Noise mean is 0.00001 Standard deviation is 19.52933

White region:

Image mean is 188.60692 Standard deviation is 19.46295

Noise mean is -0.00000 Standard deviation is 19.47054

In both cases the noise mean was very close to zero (although we have assumed this), and the standard deviation was very close to 20, which was the value used to create the noisy image.

The second major type of noise is called *signal dependant noise*. In this case the level of the noise value at each point in the image is a function of the grey level there. The grain seen in some photographs is an example of this sort of noise, and it is generally harder to deal with. Fortunately it is less often of importance, and becomes manageable if the photograph is sampled properly.

Figure 1.7 shows a step edge subjected to noise of a type that can be characterized by a normal distribution. This is an artificial edge generated by computer, so its exact location is known. It is difficult to see this in all of the random variations, but a good edge detector should be able to determine the edge position in even this situation.

Returning, with less confidence, to the case of the ideal step edge, the question of how to identify the location of the edge remains. An edge, based on the previous discussion, is defined by a grey level (or color) contour. If this contour is crossed then the level changes rapidly; following the contour leads to more subtle, possibly random, level changes. This leads to the conclusion that an edge has a measurable direction. Also, although it is by the large level change observed when crossing the



```

/* Measure the Normally distributed noise in a small region.
   Assume that the mean is zero.*/

#include <stdio.h>
#include <math.h>
#define MAX
#include "lib.h"

main(int argc, char *argv[])
{
    IMAGE im;
    int i,j,k;
    float x, y, z;
    double mean, sd;

    im = Input_PBM (argv[1]);

/* Measure */
    k = 0;
    x = y = 0.0;
    for (i=0; i<im->info->nr; i++)
        for (j=0; j<im->info->nc; j++)
        {
            x += (float)(im->data[i][j]);
            y += (float)(im->data[i][j]) * (float)(im->data[i][j]);
            k += 1;
        }

/* Compute estimate - mean noise is 0 */
    sd = (double)(y - x*x/(float)k)/(float)(k-1);
    mean = (double)(x/(float)k);
    sd = sqrt(sd);
    printf ("Image mean is %10.5f Standard deviation is %10.5f\n",
           mean, sd);

/* Now assume that the uniform level is the mean, and compute the
   mean and SD of the differences from that!*/
    x = y = z = 0.0;
    for (i=0; i<im->info->nr; i++)
        for (j=0; j<im->info->nc; j++)
        {
            z = (float)(im->data[i][j] - mean);
            x += z;
            y += z*z;
        }
    sd = (double)(y - x*x/(float)k)/(float)(k-1);
    mean = (double)(x/(float)k);
    sd = sqrt(sd);
    printf ("Noise mean is %10.5f Standard deviation is %10.5f\n",
           mean, sd);
}

```

---

**FIGURE 1.6** - A C program for estimating the noise in an image. The input image is sampled from the image to be measured, and must be a region that would ordinarily have a constant grey level.

---

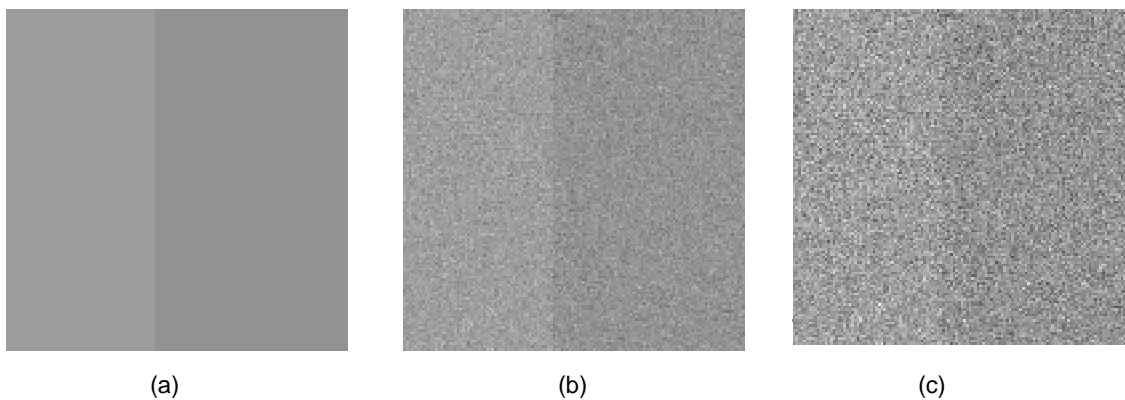
contour that an edge pixel can first be identified, it is the fact that such pixels connect to form a contour that permits the separation of noise from edge pixels. Noise pixels also show a large change in level.

There are essentially three common types of operators for locating edges. The first type is a derivative operator designed to identify places where there are large intensity changes. The second resembles a template matching scheme, where the edge is modeled by a small image showing the abstracted properties of a perfect edge. Finally there are operators that use a mathematical model of the edge; the best of these use a model of the noise also, and make an effort to take it into account. Our interest is mainly in the latter types, but examples of the first two types will be explored first.

### 1.2.3 Derivative Operators

Since an edge is defined by a change in grey level, an operator that is sensitive to this change will operate as an edge detector. A derivative operator does this; one interpretation of a derivative is as the rate of change of a function, and the rate of change of the grey levels in an image is large near an edge and small in constant areas.

Since images are two dimensional, it is important to consider level changes in many directions. For this reason, the partial derivatives of the image are used, with respect to the principal directions  $x$  and  $y$ . An estimate of the actual edge direction can be obtained by using the derivatives in  $x$  and  $y$  as the components of the actual direction along the axes, and



**FIGURE 1.7** - (a) A step edge subjected to Gaussian (normal distribution) noise. (b) Standard deviation is 10. (c) Standard deviation is 20. Note that the edge is getting lost in the random noise

computing the vector sum. The operator involved happens to be the *gradient*, and if the image is thought of as a function of two variables  $A(\mathbf{x}, \mathbf{y})$  then the gradient is defined as:

$$\nabla A(x, y) = \left( \frac{\partial A}{\partial x}, \frac{\partial A}{\partial y} \right) \quad (\text{EQ 1.3})$$

which is a two dimensional vector.

Of course, an image is not a function, and can not be differentiated in the usual way. Because an image is discrete, we use *differences* instead; that is, the derivative at a pixel is approximated by the difference in grey levels over some local region. The simplest such approximation is the operator  $\nabla_1$  :

$$\begin{aligned} \nabla_{x1} A(x, y) &= A(x, y) - A(x - 1, y) \\ \nabla_{y1} A(x, y) &= A(x, y) - A(x, y - 1) \end{aligned} \quad (\text{EQ 1.4})$$

The assumption in this case is that the grey levels vary linearly between the pixels, so that no matter where the derivative is taken its value is the slope of the line. One problem with this approximation is that it does not compute the gradient at the point  $(\mathbf{x}, \mathbf{y})$ , but at  $(\mathbf{x}-1/2, \mathbf{y}-1/2)$ . The edge locations would therefore be shifted by one half of a pixel in the  $-\mathbf{x}$  and  $-\mathbf{y}$  directions. A better choice for an approximation might be  $\nabla_2$  :

$$\begin{aligned} \nabla_{x2} A &= A(x + 1, y) - A(x - 1, y) \\ \nabla_{y2} A &= A(x, y + 1) - A(x, y - 1) \end{aligned} \quad (\text{EQ 1.5})$$

This operator is symmetrical with respect to the pixel  $(\mathbf{x}, \mathbf{y})$ , although it does not consider the value of the pixel at  $(\mathbf{x}, \mathbf{y})$ .

Whichever operator is used to compute the gradient, the resulting vector contains information about how strong the edge is at that pixel and what its direction is. The magnitude of the gradient vector is the length of the hypotenuse of the right triangle having sides  $\nabla_x$  and  $\nabla_y$ , and this reflects the strength of the edge, or *edge response*, at any given pixel. The direction of the edge at the same pixel is the angle that the hypotenuse makes with the axis.

Mathematically, the edge response is given by:

$$G_{mag} = \sqrt{\left(\frac{\partial A}{\partial x}\right)^2 + \left(\frac{\partial A}{\partial y}\right)^2} \quad (\text{EQ 1.6})$$

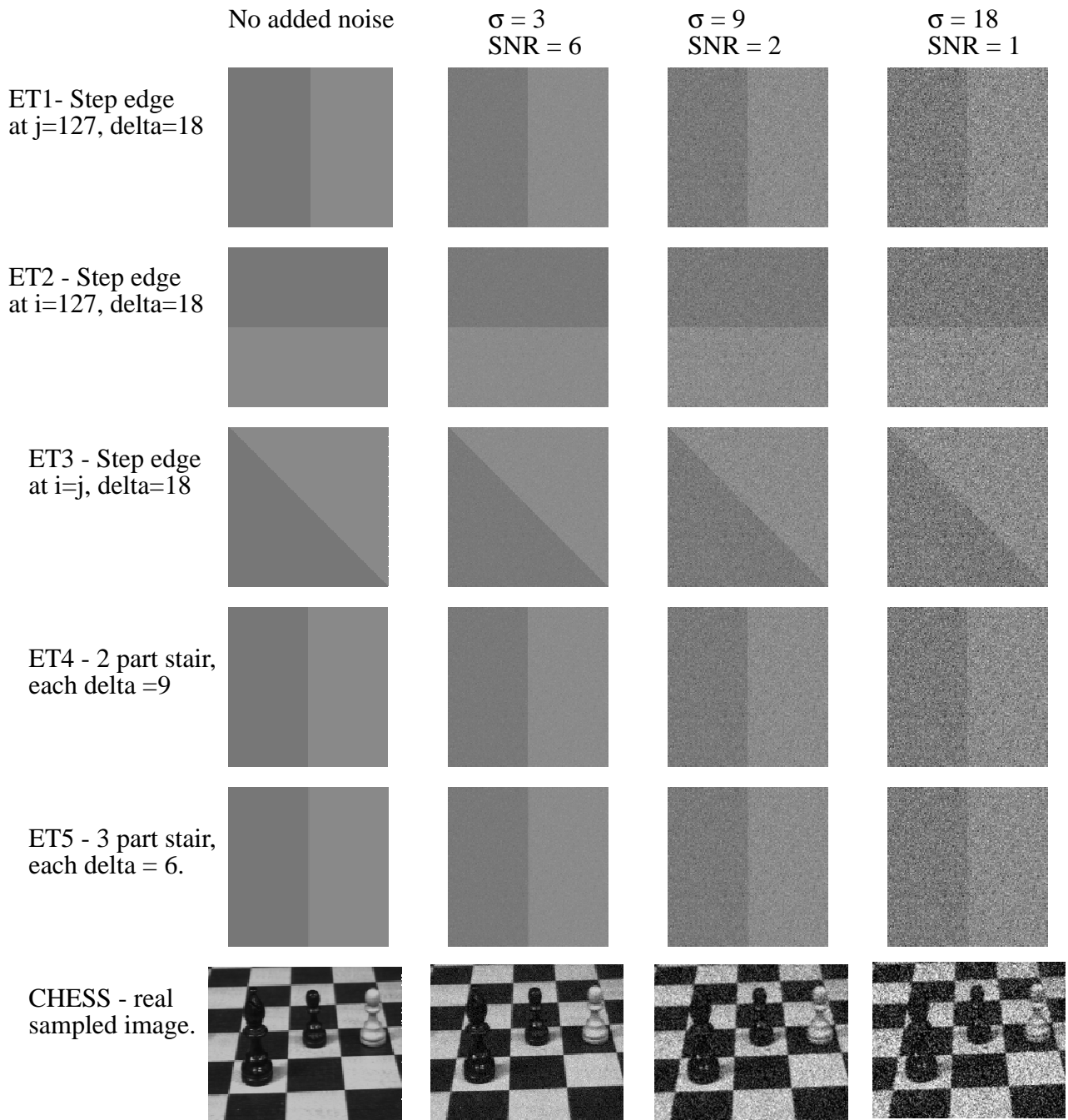
and the direction of the edge is approximately:

$$G_{dir} = \text{atan}\left(\frac{\frac{\partial A}{\partial y}}{\frac{\partial A}{\partial x}}\right) \quad (\text{EQ 1.7})$$

The edge magnitude will be a real number, and is usually converted to an integer by rounding. Any pixel having a gradient that exceeds a specified threshold value is said to be an edge pixel, and others are not. Technically, an edge detector will report the edge pixels only, while edge enhancement draws the edge pixels over the original image. This distinction will not be important in the further discussion. The two edge detectors evaluated here will use the middle value in the range of grey levels as a threshold.

At this point it would be useful to see the results of the two gradient operators applied to an image. For the purposes of evaluation of all of the methods to be presented a standard set of test images is suggested; the basic set appears in Figure 1.8, and noisy versions of these will also be used. Noise will be normal, and have standard deviations of 3, 9, and 18. For edge gradient of 18 grey levels, these correspond to signal to noise ratios of 6, 2, and 1. The appearance of the edge enhanced test images will give a rough cue about how successful the edge detection algorithm is.

In addition, it would be nice to have a numerical measure of how successful an edge detection scheme is in an absolute sense. There is no such measure in general, but something usable can be constructed by thinking about the ways in which an edge detector can fail, or be wrong. First, an edge detector can report an edge where none exists; this can be due to noise, or simply poor design or thresholding, and is called a *false positive*. In addition, an edge detector could fail to report an edge pixel that does exist; this is a *false negative*. Finally, the position of the edge pixel could be wrong. An edge detector that reports edge pixels in their proper positions is obviously better than one that does not, and this must be measured somehow. Since most of the test images will have known numbers



**FIGURE 1.8** - Standard test images for edge detector evaluation. There are three step edges and two stairs, plus a real sampled image; all have been subjected to normally distributed zero mean noise with known standard deviations of 3, 9, and 18.

and positions of edge pixels, and will have noise of a known type and quantity applied, the application of the edge detectors to the standard images will give an approximate measure of their effectiveness.

One possible way to evaluate an edge detector, based on the above discussion, was proposed by Pratt[1978], who suggested the following function:

$$E_1 = \frac{\sum_{i=1}^{I_A} \left( \frac{1}{1 + \alpha d(i)^2} \right)}{\max(I_A, I_I)} \quad (\text{EQ 1.8})$$

where  $I_A$  is the number of edge pixels found by the edge detector,  $I_I$  is the actual number of edge pixels in the test image, and the function  $d(i)$  is the distance between the actual  $i$ th pixel and the one found by the edge detector. The value  $\alpha$  is used for scaling, and should be kept constant for any set of trials. A value of  $1/9$  will be used here, as it was in Pratt's work. This metric is, as discussed previously, a function of the distance between correct and measured edge positions, but is only indirectly related to the false positives and negatives.

Kitchen and Rosenfeld[1981] also present an evaluation scheme, this one based on *local edge coherence*. It does not concern itself with the actual position of an edge, and so is a supplement to Pratt's metric. It does concern how well the edge pixel fits into the local neighborhood of edge pixels. The first step is the definition of a function that measures how well an edge pixel is continued on the left; this function is:

$$L(k) = \begin{cases} a(d, d_k) a\left(\frac{k\pi}{4}, d + \frac{\pi}{2}\right) & \text{if neighbor } k \text{ is an edge pixel} \\ 0 & \text{Otherwise} \end{cases} \quad (\text{EQ 1.9})$$

where  $d$  is the edge direction at the pixel being tested,  $d_0$  is the edge direction at its neighbor to the right,  $d_1$  is the direction of the upper-right neighbor, and so on counterclockwise about the pixel involved. The function  $a$  is a measure of the angular difference between any two angles:

$$a(\alpha, \beta) = \frac{\pi - |\alpha - \beta|}{\pi} \quad (\text{EQ 1.10})$$

A similar function measures directional continuity on the right of the pixel being evaluated:

$$R(k) = \begin{cases} a(d, d_k) a\left(\frac{k\pi}{4}, d - \frac{\pi}{2}\right) & \text{if neighbor } k \text{ is an edge pixel} \\ 0 & \text{Otherwise} \end{cases} \quad (\text{EQ 1.11})$$

The overall continuity measure is taken to be the average of the best (largest) value of  $L(k)$  and the best value of  $R(k)$ ; this measure is called  $C$ .

Then a measure of thinness is applied. An edge should be a thin line, one pixel wide. Lines of a greater width imply that false positives exist, probably because the edge detector has responded more than once to the same edge. The thinness measure  $T$  is the fraction of the six pixels in the  $3 \times 3$  region centered at the pixel being measured, not counting the center and the two pixels found by  $L(k)$  and  $R(k)$ , that are edge pixels. The overall evaluation of the edge detector is:

$$E_2 = \gamma C + (1 - \gamma) T \quad (\text{EQ 1.12})$$

where  $\gamma$  is a constant: we will use the value 0.8 here.

We are now prepared to evaluate the two gradient operators. Each of the operators was applied to each of the 24 test images. Then both the Pratt and the KR metric was taken on the results, with the following outcome. For  $\nabla_1$ :

---

**TABLE 1.1**

Evaluation of the  $\nabla_1$  operator.

Image	Evaluator	No Noise	SNR = 6	SNR = 2	SNR=1
ET1	Eval 1	0.9650	0.5741	0.0510	0.0402
	Eval 2	1.0000	0.6031	0.3503	0.3494
ET2	Eval 1	0.9650	0.6714	0.0484	0.0392
	Eval 2	1.0000	0.6644	0.3491	0.3493
ET3	Eval 1	0.9726	0.7380	0.0818	0.0564
	Eval 2	0.9325	0.6743	0.3532	0.3493
ET4	Eval 1	0.4947	0.0839	0.0375	0.0354
	Eval 2	0.8992	0.3338	0.3473	0.3489
ET5	Eval 1	0.4772	0.0611	0.0365	0.0354
	Eval 2	0.7328	0.3163	0.34614	0.3485

The drop in quality for ET4 and ET5 is due to the operator giving a response to each step, rather than a single overall response to the edge.

For  $\nabla_2$  we found:

**TABLE 1.2**

Evaluation of the  $\nabla_2$  operator

Image	Evaluator	No Noise	SNR = 6	SNR = 2	SNR = 1
ET1	Eval 1	0.9727	0.8743	0.0622	0.0421
	Eval 2	0.8992	0.6931	0.4167	0.4049
ET2	Eval 1	0.9726	0.9454	0.0612	0.0400
	Eval 2	0.8992	0.6696	0.4032	0.4049
ET3	Eval 1	0.9726	0.9707	0.1000	0.0623
	Eval 2	0.9325	0.9099	0.4134	0.4058
ET4	Eval 1	0.5158	0.4243	0.0406	0.0320
	Eval 2	1.000	0.5937	0.4158	0.4043
ET5	Eval 1	0.5062	0.1963	0.0360	0.0316
	Eval 2	0.8992	0.4097	0.4147	0.4046

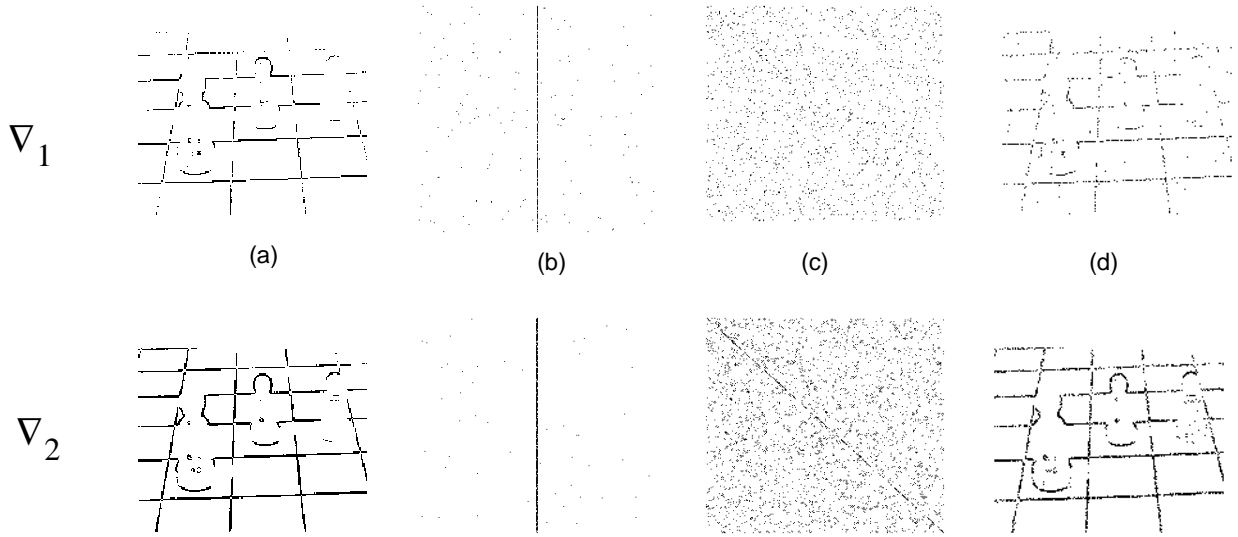
This operator gave two edge pixels at each point along the edge, one in each region. As a result, each of the two pixels contributes to the distance to the actual edge. This duplication of edge pixels should have been penalized in one of the evaluations, but E1 does not penalize extra edge pixels as much as it does missing ones.

It is not possible to show all of the edge enhanced images, since in this case alone there are 48 of them. Figure 1.9 shows a selection of the results from both operators, and from these images, and from the evaluations, it can be concluded that  $\nabla_2$  is slightly superior, especially where the noise is higher.

#### 1.2.4 Template Based Edge Detection

The idea behind template based edge detection is to use a small, discrete template as a model of an edge instead of using a derivative operator directly, as in the previous section, or a complex, more global model, as in the next section. The template can be either an attempt to model the level changes in the edge, or an attempt to approximate a derivative operator; the latter appears to be most common.





**FIGURE 1.9** - Sample results from the gradient edge detectors. (a) Chess image ( $\sigma=3$ ). (b) ET1 image (SNR=6). (c) ET3 image (SNR=2). (d) Chess image ( $\sigma=18$ ).

There is a vast array of template based edge detectors. Two were chosen to be examined here, simply because they provide the best sets of edge pixels while using a small template. The first of these is the Sobel edge detector, which uses templates in the form of convolution masks having the following values:

$$\begin{array}{cc} \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix} = S_y & \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} = S_x \end{array}$$

One way to view these templates is as an approximation to the gradient at the pixel corresponding to the center of the template. Note that the weights on the diagonal elements is smaller than the weights on the horizontal and vertical. The  $x$  component of the Sobel operator is  $S_x$ , and the  $y$  component is  $S_y$ ; considering these as components of the gradient means that the magnitude and direction of the edge pixel is given by Equations 1.6 and 1.7.

For a pixel at image coordinates  $(i,j)$ ,  $S_x$  and  $S_y$  can be computed by:

$$S_x = I[i-1][j+1] + 2I[i][j+1] + I[i+1][j+1] - (I[i-1][j-1] + 2I[i][j-1] + I[i+1][j-1])$$

$$S_y = I[i+1][j+1] + 2I[i+1][j] + I[i+1][j-1] - (I[i-1][j+1] + 2I[i-1][j] + I[i-1][j-1])$$

which is equivalent to applying the operator  $\nabla_1$  to each 2x2 portion of the 3x3 region, and then averaging the result. After  $S_x$  and  $S_y$  are computed for every pixel in an image, the resulting magnitudes must be thresholded. All pixels will have some response to the templates, but only the very large responses will correspond to edges. The best way to compute the magnitude is by using Equation 1.6, but this involves a square root calculation that is both intrinsically slow and requires the use of floating point numbers. Optionally we could use the sum of the absolute values of  $S_x$  and  $S_y$  (that is:  $|S_x| + |S_y|$ ) or even the largest of the two values. Thresholding could be done using almost any standard method. Sections 1.4 and 1.5 describe some techniques that are specifically intended for use on edges.

The second example of the use of templates is the one described by Kirsch, and were selected as an example here because these templates have a different motivation than Sobel's. For the 3x3 case the templates are:

$$\begin{array}{cccc}
 \begin{array}{ccc} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{array} & 
 \begin{array}{ccc} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{array} & 
 \begin{array}{ccc} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{array} & 
 \begin{array}{ccc} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{array} \\
 K0 = & K1 = & K2 = & K3 = \\
 \begin{array}{ccc} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{array} & 
 \begin{array}{ccc} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{array} & 
 \begin{array}{ccc} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{array} & 
 \begin{array}{ccc} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{array} \\
 K4 = & K5 = & K6 = & K7 =
 \end{array}$$

These masks are an effort to model the kind of grey level change seen near an edge having various orientations, rather than an approximation to the gradient. There is one mask for each of eight compass directions. For example, a large response to mask **K0** implies a vertical edge (horizontal gradient) at the pixel corresponding to the center of the mask. To find the edges, an image  $I$  is convolved with all of the masks at each pixel position. The response of the operator at a pixel is the *maximum* of the responses of any of the eight masks. The direction of the edge pixel is quantized into eight possibilities here, and is  $\pi/4 * i$ , where  $i$  is the number of the mask having the largest response.

Both of these edge detectors were evaluated using the test images of Figure 1.8. The results, in tabular form as before, are:

---

**TABLE 1.3** Evaluation of the Sobel edge detector

Image	Evaluator	No Noise	SNR=6	SNR=2	SNR=1
ET1	Eval 1	0.9727	0.9690	0.1173	0.0617
	Eval 2	0.8992	0.8934	0.4474	0.4263
ET2	Eval 1	0.9726	0.9706	0.1609	0.0526
	Eval 2	0.8992	0.8978	0.4215	0.4255
ET3	Eval 1	0.9726	0.9697	0.1632	0.0733
	Eval 2	0.9325	0.9186	0.4349	0.4240
ET4	Eval 1	0.4860	0.4786	0.0595	0.0373
	Eval 2	0.7328	0.6972	0.4426	0.4266
ET5	Eval 1	0.4627	0.3553	0.0480	0.0355
	Eval 2	0.7496	0.6293	0.4406	0.4250

And for the Kirsch operator:

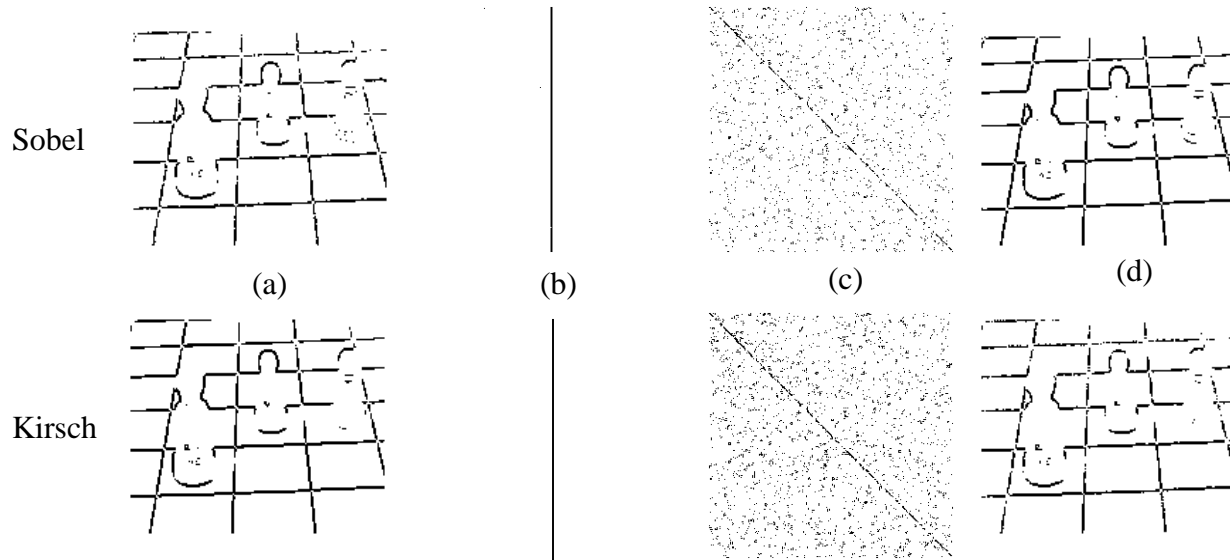
---

**TABLE 1.4** Evaluation of the Kirsch edge detector

Image	Evaluator	No Noise	SNR=6	SNR=2	SNR=1
ET1	Eval 1	0.9727	0.9727	0.1197	0.0490
	Eval 2	0.8992	0.8992	0.4646	0.4922
ET2	Eval 1	0.9726	0.9726	0.1517	0.0471
	Eval 2	0.8992	0.8992	0.4528	0.4911
ET3	Eval 1	0.9726	0.9715	0.1458	0.0684
	Eval 2	0.9325	0.9200	0.4708	0.4907
ET4	Eval 1	0.4860	0.4732	0.0511	0.0344
	Eval 2	0.7328	0.7145	0.4819	0.4907
ET5	Eval 1	0.4627	0.3559	0.0412	0.0339
	Eval 2	0.7496	0.6315	0.5020	0.4894

Figure 1.10 shows the response of these templates applied to a selection of the test images. Based on the evaluations and the appearance of the test images the Kirsch operator appears to be the best of the two template operators, although the two are very close. Both template operators are both superior to the simple derivative operators, especially as the noise increases.

It should be pointed out that in all cases studied so far there are unspecified aspects to the edge detection methods that will have an impact on their efficacy. Principal among these is the thresholding method used, but



**FIGURE 1.10** - Sample results from the template edge detectors. (a) Chess image, noise  $\sigma = 3$ . (b) ET1, SNR=6. (c) ET3, SNR=2. (d) Chess image, noise  $\sigma = 18$ .

sometimes simple noise removal is done beforehand and edge thinning is done afterward. The model based methods that follow generally include these features, sometimes as part of the edge model.

### 1.3 Edge Models: Marr-Hildreth Edge Detection

---

In the late 1970's, David Marr attempted to combine what was known about biological vision into a model that could be used for machine vision. According to Marr, "... *the purpose of early visual processing is to construct a primitive but rich description of the image that is to be used to determine the reflectance and illumination of the visible surfaces, and their orientation and distance relative to the viewer*" [Marr1980]. The lowest level description he called the *primal sketch*, a major component of which are the edges.

Marr studied the literature on mammalian visual systems and summarized these in five major points:

1. In natural images, features of interest occur at a variety of scales. No single operator can function at all of these scales, so the result of operators at each of many scales should be combined.

2. A natural scene does not appear to consist of diffraction patterns or other wave-like effects, and so some form of local averaging (*smoothing*) must take place.
3. The optimal smoothing filter that matches the observed requirements of biological vision (smooth and localized in the spatial domain and smooth and band-limited in the frequency domain) is the *Gaussian*.
4. When a change in intensity (an edge) occurs there is an extreme value in the first derivative or intensity. This corresponds to a *zero crossing* in the second derivative.
5. The orientation independent differential operator of lowest order is the *Laplacian*.

Each of these points is either supported by the observation of vision systems or derived mathematically, but the overall grounding of the resulting edge detector is still a little loose. However, based on the five points above, an edge detection algorithm can be stated as follows:

1. Convolve the image  $I$  with a two dimensional Gaussian function.
2. Compute the Laplacian of the convolved image; call this  $L$ .
3. Edges pixels are those for which there is a zero crossing in  $L$ .

The results of convolutions with Gaussians having a variety of standard deviations are combined to form a single edge image. Standard deviation is a measure of scale in this instance.

The algorithm is not difficult to implement, although it is more difficult than the methods seen so far. A convolution in two dimensions can be expressed as:

$$I * G(i, j) = \sum_n \sum_m I(n, m) G(i - n, j - m) \quad (\text{EQ 1.13})$$

The function  $G$  being convolved with the image is a two dimensional Gaussian, which is:

$$G_{\sigma}(x, y) = \sigma^2 e^{\frac{-(x^2 + y^2)}{\sigma^2}} \quad (\text{EQ 1.14})$$

To perform the convolution on a digital image the Gaussian must be sampled to create a small two dimensional image. After the convolution, the Laplacian operator can be applied. This is:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (\text{EQ 1.15})$$

and could be computed using differences. However, since order does not matter in this case, we could compute the Laplacian of the Gaussian analytically and sample that function, creating a convolution mask that can be applied to the image to yield the same result. The Laplacian of a Gaussian (LoG) is:

$$\nabla^2 G_\sigma = \left( \frac{r^2 - 2\sigma^2}{\sigma^4} \right) e^{\left( \frac{-r^2}{2\sigma^2} \right)} \quad (\text{EQ 1.16})$$

where  $r = \sqrt{x^2 + y^2}$ . This latter approach is the one taken in the C code implementing this operator, which appears at the end of this chapter.

This program first creates a two dimensional, sampled version of the Laplacian of the Gaussian (called **lgau** in the function **marr**) and convolves this in the obvious way with the input image (function **convolution**). Then the zero crossings are identified and pixels at those positions are marked.

A zero crossing at a pixel P implies that the values of the two opposing neighboring pixels in some direction have different signs. For example, if the edge through P is vertical then the pixel to the left of P will have a different sign than the one to the right of P. There are four cases to test: up/down, left/right, and the two diagonals. This test is performed for each pixel in the Laplacian of the Gaussian by the function **zero\_cross**.

In order to ensure that a variety of scales are used, the program uses two different Gaussians, and selects the pixels that have zero crossings in both scales as output edge pixels. More than two Gaussians could be used, of course. The program accepts a standard deviation value  $\sigma$  as a parameter, either from the command line or from the parameter file 'marr.par'. It then uses both  $\sigma+0.8$  and  $\sigma-0.8$  as standard deviation values, does two convolutions, locates two sets of zero crossings, and

merges the resulting edge pixels into a single image. The program is called ‘marr’, and can be invoked as

```
marr input.pgm 2.0
```

which would read in the image file named ‘input.pgm’ and apply the Marr-Hildreth edge detection algorithm using 1.2 and 2.8 as standard deviations.

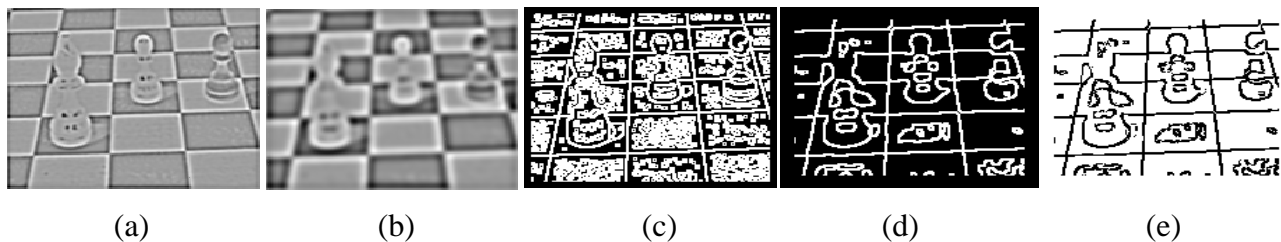
Figure 1.11 illustrates the steps in this process, using the chess image (no noise) as an example. Figures 1.11a and b shows the original image after being convolved with the Laplacian of the Gaussians, having  $\sigma$  values of 1.2 and 2.8 respectively. Figures 1.11c and 1.11d are the responses from these two different values of  $\sigma$ , and Figure 1.11e shows the result of merging the edge pixels in these two images.

Figure 1.12 shows the result of the Marr-Hildreth edge detector applied to the all of the test images of Figure 1.8. In addition, the evaluation of this operator is:

**TABLE 1.5**

Evaluation of the Marr-Hildreth edge detector

Image	Evaluator	No Noise	SNR = 6	SNR = 2	SNR = 1
ET1	Eval 1	0.8968	0.7140	0.7154	0.2195
	Eval 2	0.9966	0.7832	0.6988	0.7140
ET2	Eval 1	0.6948	0.6948	0.6404	0.1956
	Eval 2	0.9966	0.7801	0.7013	0.7121
ET3	Eval 1	0.7362	0.7319	0.7315	0.2671
	Eval 2	0.9133	0.7766	0.7052	0.7128
ET4	Eval 1	0.4194	0.4117	0.3818	0.1301
	Eval 2	0.8961	0.7703	0.6981	0.7141



**FIGURE 1.11** - Steps in the computation of the Marr-Hildreth edge detector. (a) Convolution of the original image with the Laplacian of a Gaussian having  $\sigma = 1.2$ . (b) Convolution of the image with the Laplacian of a Gaussian having  $\sigma = 2.8$ . (c) Zero crossings found in (a). (d) Zero crossings found in (b). (e) Result, found by using zero crossings common to both.

TABLE 1.5

Evaluation of the Marr-Hildreth edge detector

Image	Evaluator	No Noise	SNR = 6	SNR = 2	SNR = 1
ET5	Eval 1	0.3694	0.3822	0.3890	0.1290
	Eval 2	0.9966	0.7626	0.6995	0.7141

The evaluations above tend to be low. Because of the width of the Gaussian filter, the pixels that are a distance less than about  $4\sigma$  from the boundary of the image are not processed, and hence E1 thinks of these as missing edge pixels. When this is taken into account the evaluation using ET1 with no noise, as an example, becomes 0.9727. Some of the other low evaluations are, on the other hand, the fault of the method. Locality is not especially good, and the edges are not always thin. Still, this edge detector is much better than the previous ones in cases of low signal to noise ratio.

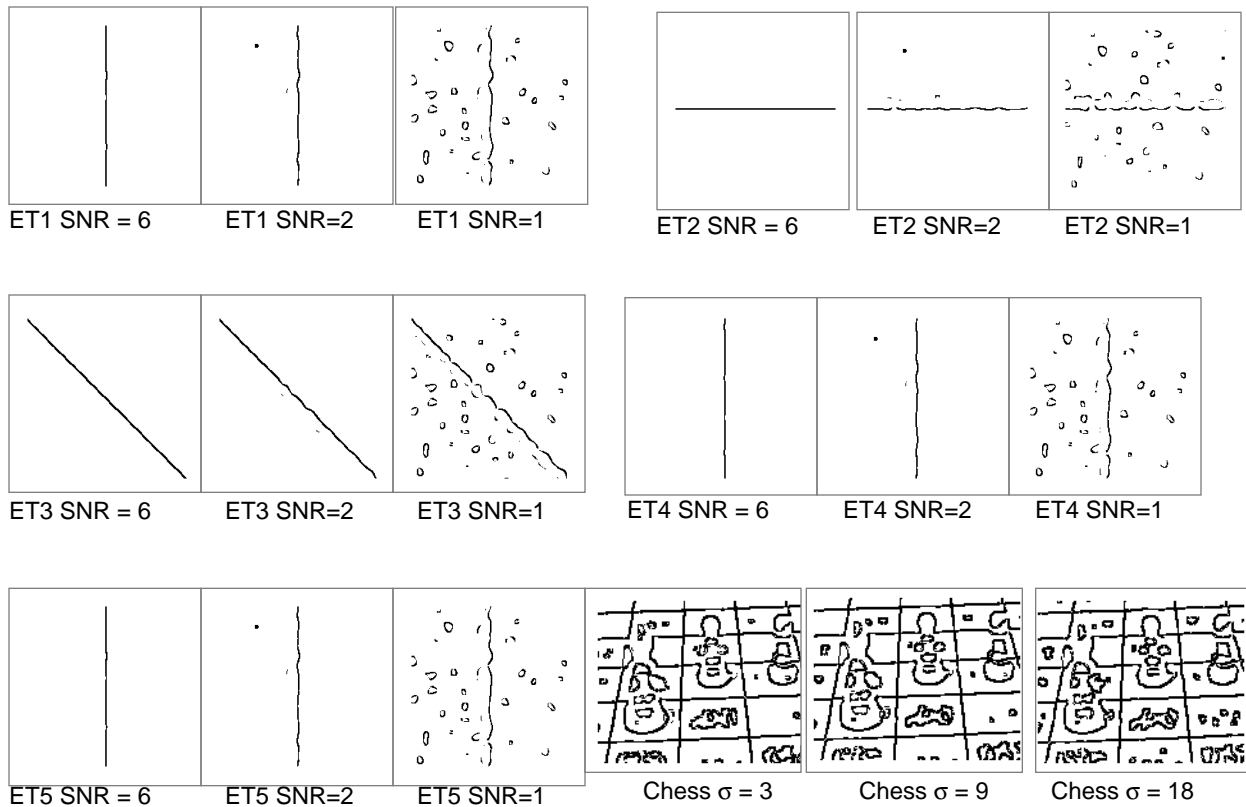


FIGURE 1.12 - Edges from the test images as found by the Marr-Hildreth algorithm, using two resolution values.



---

## 1.4 The Canny Edge Detector

---

In 1986, John Canny defined a set of goals for an edge detector and described an optimal method for achieving them.

Canny specified three issues that an edge detector must address. In plain English, these are:

1. **Error rate** - the edge detector should respond only to edges, and should find all of them; no edges should be missed.
2. **Localization** - the distance between the edge pixels as found by the edge detector and the actual edge should be as small as possible.
3. **Response** - the edge detector should not identify multiple edge pixels where only a single edge exists.

These seem reasonable enough, especially since the first two have already been discussed and used to evaluate edge detectors. The response criterion seems very similar to a false positive, at first glance.

Canny assumed a step edge subject to white Gaussian noise. The edge detector was assumed to be a convolution filter  $\mathbf{f}$  which would smooth the noise and locate the edge. The problem is to identify the one filter that optimizes the three edge detection criteria.

In one dimension, the response of the filter  $\mathbf{f}$  to an edge  $\mathbf{G}$  is given by a convolution integral:

$$H = \int_{-W}^W G(-x)f(x)dx \quad (\text{EQ 1.17})$$

The filter is assumed to be zero outside of the region  $[-W, W]$ . Mathematically, the three criteria are expressed as:

$$SNR = \frac{A \left| \int_{-W}^0 f(x)dx \right|}{n_0 \sqrt{\int_{-W}^W f^2(x)dx}} \quad (\text{EQ 1.18})$$

$$Localization = \frac{A|f'(0)|}{n_0 \sqrt{\int_{-W}^W f'^2 dx}} \quad (\text{EQ 1.19})$$

$$x_{zc} = \pi \left( \frac{\int_{-\infty}^{\infty} f'^2(x) dx}{\int_{-\infty}^{\infty} f''^2(x) dx} \right)^{\frac{1}{2}} \quad (\text{EQ 1.20})$$

The value of **SNR** is the output signal to noise ratio (error rate), and should be as large as possible: we need lots of signal and little noise. The **localization** value represents the reciprocal of the distance of the located edge from the true edge, and should also be as large as possible, which means that the distance would be as small as possible. The value  $\mathbf{x}_{zc}$  is a constraint; it represents the mean distance between zero crossings of  $\mathbf{f}'$ , and is essentially a statement that the edge detector  $\mathbf{f}$  will not have too many responses to the same edge in a small region.

Canny attempts to find the filter  $\mathbf{f}$  that maximizes the product **SNR\*localization** subject to the multiple response constraint, and while the result is too complex to be solved analytically, an efficient approximation turns out to be the *first derivative of a Gaussian* function. recall that a Gaussian has the form:

$$G(x) = e^{-\frac{x^2}{2\sigma^2}} \quad (\text{EQ 1.21})$$

The derivative with respect to  $\mathbf{x}$  is therefore

$$G'(x) = \left(-\frac{x}{\sigma^2}\right) e^{-\left(\frac{x^2}{2\sigma^2}\right)} \quad (\text{EQ 1.22})$$

In two dimensions, a Gaussian is given by

$$G(x, y) = \sigma^2 e^{-\left(\frac{x^2 + y^2}{2\sigma^2}\right)} \quad (\text{EQ 1.23})$$

and  $\mathbf{G}$  has derivatives in both the  $\mathbf{x}$  and  $\mathbf{y}$  directions. The approximation to Canny's optimal filter for edge detection is  $\mathbf{G}'$ , and so by convolving the input image with  $\mathbf{G}'$  we obtain an image  $\mathbf{E}$  that has enhanced edges, even in the presence of noise, which has been incorporated into the model of the edge image.

A convolution is fairly simple to implement, but is expensive computationally, especially a two dimensional convolution. This was seen in the Marr edge detector. However, a convolution with a two dimensional Gaussian can be separated into two convolutions with one-dimensional Gaussians, and the differentiation can be done afterwards. Indeed, the differentiation can also be done by convolutions in one dimension, giving two images: one is the  $x$  component of the convolution with  $\mathbf{G}'$  and the other is the  $y$  component.

Thus, the Canny edge detection algorithm to this point is:

1. Read in the image to be processed,  $\mathbf{I}$ .
2. Create a 1D Gaussian mask  $\mathbf{G}$  to convolve with  $\mathbf{I}$ . The standard deviation ( $s$ ) of this Gaussian is a parameter to the edge detector.
3. Create a 1D mask for the first derivative of the Gaussian in the  $\mathbf{x}$  and  $\mathbf{y}$  directions; call these  $\mathbf{G}_x$  and  $\mathbf{G}_y$ . The same  $s$  value is used as in step 2 above.
4. Convolve the image  $\mathbf{I}$  with  $\mathbf{G}$  along the rows to give the  $\mathbf{x}$  component image  $\mathbf{I}_x$ , and down the columns to give the  $\mathbf{y}$  component image  $\mathbf{I}_y$ .
5. Convolve  $\mathbf{I}_x$  with  $\mathbf{G}_x$  to give  $\mathbf{I}_x'$ , the  $\mathbf{x}$  component of  $\mathbf{I}$  convolved with the derivative of the Gaussian, and convolve  $\mathbf{I}_y$  with  $\mathbf{G}_y$  to give  $\mathbf{I}_y'$ .
6. If you want to view the result at this point the  $x$  and  $y$  components must be combined. The magnitude of the result is computed at each pixel  $(x, y)$  as:

$$M(x, y) = \sqrt{I'_x(x, y)^2 + I'_y(x, y)^2}$$

The magnitude is computed in the same manner as it was for the gradient, which is in fact what is being computed.

A complete C program for a Canny edge detector is given at the end of this chapter, but some explanation is relevant at this point. The main program opens the image file and reads it, and also reads in the parameters, such as  $\sigma$ . It then calls the function **canny**, which does most of the actual work. The first thing **canny** does is to compute the Gaussian filter mask (called **gau** in the program) and the derivative of a Gaussian filter mask (called **dgau**). The size of the mask to be used depends on  $\sigma$ ; for small  $\sigma$  the Gaussian will quickly become zero, resulting in a small mask. The program determines the needed mask size automatically.

Next, the function computes the convolution as in step 4 above. The C function **separable\_convolution** does this, being given the input image and the mask, and returning the x and y parts of the convolution (called **smx** and **smy** in the program; these are floating point 2D arrays). The convolution of step 5 above is then calculated by calling the C function **dx\_seperable\_convolution** twice, once for x and once for y. The resulting real images (called **dx** and **dy** in the program) are the x and y components of the image convolved with **G'**. The function **norm** will calculate the magnitude given any pair of x and y components.

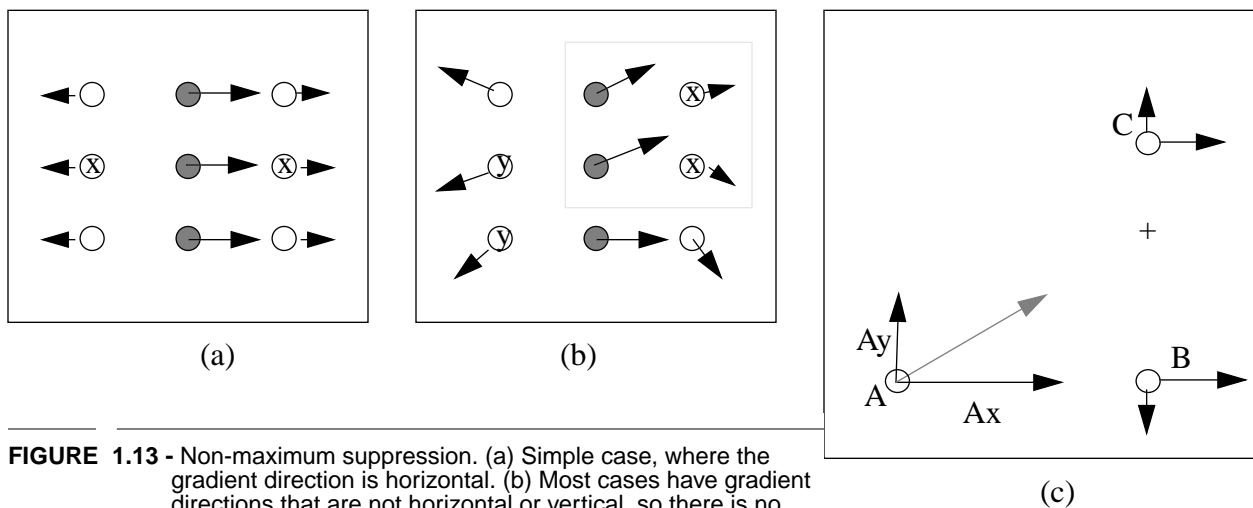
The final step in the edge detector is a little curious at first, and needs some explanation. The value of the pixels in **M** is large if they are edge pixels and smaller if not, so thresholding could be used to show the edge pixels as white and the background as black. This does not give very good results; what must be done is to threshold the image based partly on the direction of the gradient at each pixel. The basic idea is that edge pixels have a direction associated with them; the magnitude of the gradient at an edge pixel should be greater than the magnitude of the gradient of the pixels on each side of the edge. The final step in the Canny edge detector is a *non-maximum suppression* step, where pixels that are not local maxima are removed.

Figure 1.13 attempts to shed light on this process by using geometry. Part a of this figure shows a 3x3 region centered on an edge pixel which in this case is vertical. The arrows indicate the direction of the gradient at each pixel, and the length of the arrows is proportional to the magnitude of the gradient. Here, non-maximal suppression means that the center pixel, the one under consideration, must have a larger gradient magnitude than its neighbors *in the gradient direction*; these are the two pixels marked with an 'x'. That is: from the center pixel, travel in the direction

of the gradient until another pixel is encountered; this is the first neighbor. Now, again starting at the center pixel, travel in the direction opposite to that of the gradient until another pixel is encountered; this is the second neighbor. Moving from one of these to the other passes through the edge pixel in a direction that crosses the edge, so the gradient magnitude should be largest at the edge pixel.

In this specific case the situation is clear. The direction of the gradient is horizontal, and the neighboring pixels used in the comparison are exactly the left and right neighbors. Unfortunately this does not happen very often. If the gradient direction is arbitrary then following that direction will usually take you to a point in between two pixels. What is the gradient there? Its value cannot be known for certain, but it can be estimated from the gradients of the neighboring pixels. It is assumed that the gradient changes continuously as a function of position, and that the gradient at the pixel coordinates are simply sampled from the continuous case. If it is further assumed that the change in the gradient between any two pixels is a linear function, then the gradient at any point between the pixels can be approximated by a linear interpolation.

A more general case is shown in Figure 1.13b. Here the gradients all point in different directions, and following the gradient from the center pixel now takes us in between the pixels marked 'x'. Following the direction opposite to the gradient takes us between the pixels marked 'y'. Let's consider only the case involving the 'x' pixels as shown in Figure 1.13c, since the other case is really the same. The pixel named A is the one



**FIGURE 1.13** - Non-maximum suppression. (a) Simple case, where the gradient direction is horizontal. (b) Most cases have gradient directions that are not horizontal or vertical, so there is no exact gradient at the desired point. (c) Gradients at pixels neighboring A are used to estimate the gradient at the location marked with '+';

under consideration, and pixels **B** and **C** are the neighbors in the direction of the positive gradient. The vector components of the gradient at **A** are  $A_x$  and  $A_y$ , and the same naming convention will be used for **B** and **C**.

Each pixel lies on a grid line having an integer  $x$  and  $y$  coordinate. This means that pixels **A** and **B** differ by one distance unit in the  $x$  direction. It must be determined which grid line will be crossed first when moving from **A** in the gradient direction. Then the gradient magnitude will be linearly interpolated using the two pixels on that grid line and on opposite sides of the crossing point, which is at location  $(P_x, P_y)$ . In Figure 1.xxc the crossing point is marked with a '+', and is in between **B** and **C**. The gradient magnitude at this point is estimated as

$$G = (P_y - C_y)Norm(C) + (B_y - P_y)Norm(B) \quad (\text{EQ 1.24})$$

where the **norm** function computes the gradient magnitude.

Every pixel in the filtered image is processed in this way; the gradient magnitude is estimated for two locations, one on each side of the pixel, and the magnitude at the pixel must be greater than its neighbors'. In the general case there are eight major cases to check for, and some short cuts that can be made for efficiency's sake, but the above method is essentially what is used in most implementations of the Canny edge detector. The function **nonmax\_suppress** in the C source at the end of the chapter computes a value for the magnitude at each pixel based on this method, and sets the value to zero unless the pixel is a local maximum.

It would be possible to stop at this point, and use the method to enhance edges. Figure 1.14 shows the various stages in processing the chess board test image of Figure 1.8 (no added noise). The stages are: computing the result of convolving with a Gaussian in the  $x$  and  $y$  directions (Figures 1.14a and b); computing the derivatives in the  $x$  and  $y$  directions (Figure 1.14c and d); the magnitude of the gradient before non-maximal suppression (Figure 1.14e); and the magnitude after non-maximal suppression (Figure 1.14f). This last image still contains grey level values, and needs to be thresholded to determine which pixels are edge pixels and which are not. As an extra, but novel, step, Canny suggests thresholding using *hysteresis* rather than simply selecting a threshold value to apply everywhere.

Hysteresis thresholding uses a high threshold  $T_h$  and a low threshold  $T_l$ . Any pixel in the image that has a value greater than  $T_h$  is presumed to be an edge pixel, and is marked as such immediately. Then, any pixels that

are connected to this edge pixel and that have a value greater than  $T_1$  are also selected as edge pixels, and are marked too. The marking of neighbors can be done recursively, as it is in the function **hysteresis**, or by performing multiple passes through the image.

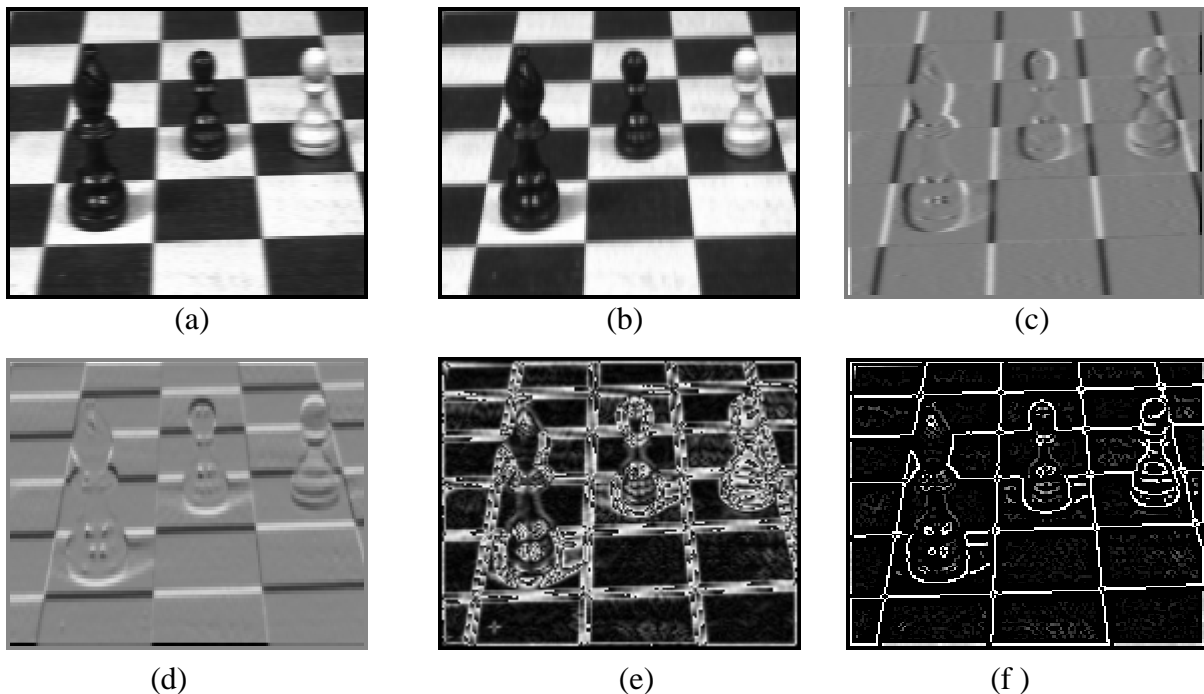
Figure 1.15 shows the result of adding hysteresis thresholding after non-maximum suppression. 1.15a is an expanded piece of Figure 1.14f, showing the pawn in the center of the board. The grey levels have been slightly scaled so that the smaller values can be seen clearly. A low threshold (1.15b) and a high threshold (1.15c) have been globally applied to the magnitude image, and the result of hysteresis thresholding is given in Figure 1.15d.

Examples of results from this edge detector will be seen in section 1.6.

## 1.5 The Shen-Castan (ISEF) Edge Detector

---

Canny's edge detector defined optimality with respect to a specific set of criteria. While these criteria seem reasonable enough, there is no compel-



**FIGURE 1.14** - Intermediate results from the Canny edge detector. (a) X component of the convolution with a Gaussian. (b) Y component of the convolution with a Gaussian. (c) X component of the image convolved with the derivative of a Gaussian. (d) Y component of the image convolved with the derivative of a Gaussian. (e) Resulting magnitude image. (f) After non-maximum suppression.

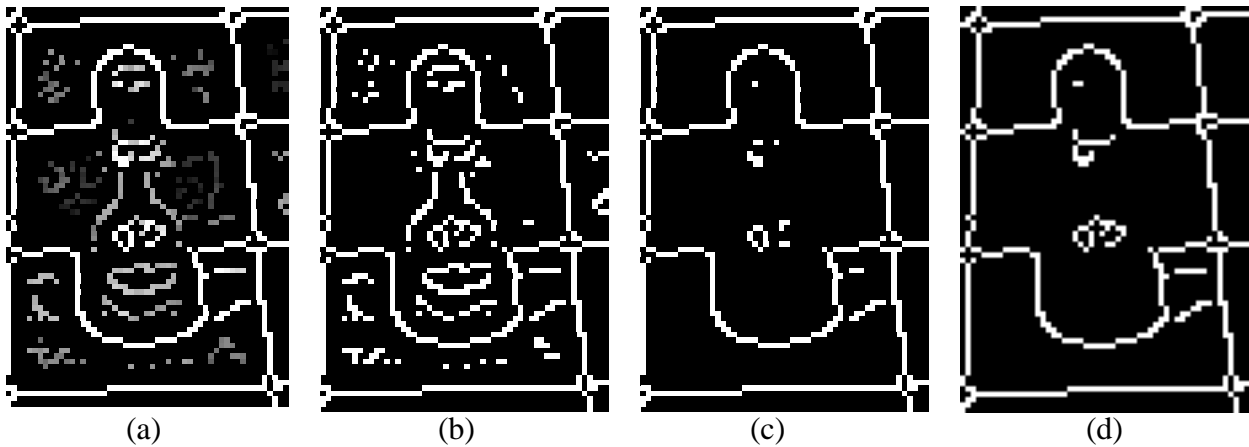
ling reason to think that they are the only possible ones. This means that the concept of optimality is a relative one, and that a better (in some circumstances) edge detector than Canny's is a possibility. In fact, sometimes it seems as if the comparison taking place is between definitions of optimality, rather than between edge detection schemes.

Shen and Castan agree with Canny about the general form of the edge detector: a convolution with a smoothing kernel followed by a search for edge pixels. However their analysis yields a different function to optimize: namely, they suggest minimizing (in one dimension):

$$C_N^2 = \frac{\int_0^\infty f^2(x)dx \cdot \int_0^\infty f'^2(x)dx}{f^4(0)} \quad (\text{EQ 1.25})$$

That is: the function that minimizes  $C_N$  is the optimal smoothing filter for an edge detector. The optimal filter function they came up with is the *infinite symmetric exponential filter* (ISEF):

$$f(x) = \frac{p}{2} e^{-p|x|} \quad (\text{EQ 1.26})$$




---

**FIGURE 1.15** - Hysteresis thresholding. (a) Enlarged portion of Figure 1.14f. (b) This portion after thresholding with a single low threshold. (c) After thresholding with a single high threshold. (d) After hysteresis thresholding.



Shen and Castan maintain that this filter gives better signal to noise ratios than Canny's filter, and provides better localization. This could be because the implementation of Canny's algorithm *approximates* his optimal filter by the derivative of a Gaussian, whereas Shen and Castan *use the optimal filter directly*, or could be due to a difference in the way the different optimality criteria are reflected in reality. On the other hand, Shen and Castan do not address the multiple response criterion, and as a result it is possible that their method will create spurious responses to noisy and blurred edges.

In two dimensions the ISEF is:

$$f(x, y) = a \cdot e^{-p(|x| + |y|)} \quad (\text{EQ 1.27})$$

which can be applied to an image in much the same way as was the derivative of Gaussian filter, as a 1D filter in the x direction, then in the y direction. However, Shen and Castan went one step further and gave a realization of their filter as one dimensional *recursive filters*. While a detailed discussion of recursive filters is beyond the scope of this book, a quick summary of this specific case may be useful.

The filter function f above is a real, continuous function. It can be rewritten for the discrete, sampled case as

$$f[i, j] = \frac{(1-b)b^{|x| + |y|}}{1+b} \quad (\text{EQ 1.28})$$

where the result is now normalized, as well. To convolve an image with this filter, recursive filtering in the x direction is done first, giving r[i,j]:

$$\begin{aligned} y_1[i, j] &= \frac{1-b}{1+b} I[i, j] + b y_1[i, j-1], j = 1 \dots N, i = 1 \dots M \\ y_2[i, j] &= b \frac{1-b}{1+b} I[i, j] + b y_1[i, j+1], j = N \dots 1, i = 1 \dots M \\ r[i, j] &= y_1[i, j] + y_2[i, j+1] \end{aligned} \quad (\text{EQ 1.29})$$

with the boundary conditions:

$$\begin{aligned}
 I[i, 0] &= 0 \\
 y_1[i, 0] &= 0 \\
 y_2[i, M + 1] &= 0
 \end{aligned}
 \tag{EQ 1.30}$$

Then filtering is done in the **y** direction, operating on  $r[i,j]$  to give the final output of the filter,  $y[i,j]$ :

$$\begin{aligned}
 y_1[i, j] &= \frac{1-b}{1+b}I[i, j] + by_1[i-1, j], i = 1 \dots M, j = 1 \dots N \\
 y_2[i, j] &= b\frac{1-b}{1+b}I[i, j] + by_1[i+1, j], i = N \dots 1, j = 1 \dots N \\
 y[i, j] &= y_1[i, j] + y_2[i+1, j]
 \end{aligned}
 \tag{EQ 1.31}$$

with the boundary conditions:

$$\begin{aligned}
 I[0, j] &= 0 \\
 y_1[0, j] &= 0 \\
 y_2[N+1, j] &= 0
 \end{aligned}
 \tag{EQ 1.32}$$

The use of recursive filtering speeds up the convolution greatly. In the ISEF implementation at the end of the chapter the filtering is performed by the function **ISEF**, which calls **ISEF\_vert** to filter the rows (Equation 29) and **ISEF\_horiz** to filter the columns (Equation 1.31). The value of **b** is a parameter to the filter, and is specified by the user.

All of the work to this point simply computes the filtered image. Edges are located in this image by finding zero crossings of the Laplacian, a process similar to that undertaken in the Marr-Hildreth algorithm. An approximation to the Laplacian can be obtained quickly by simply subtracting the original image from the smoothed image. That is, if the filtered image is **S** and the original is **I** we have:

$$S[i, j] - I[i, j] \approx \frac{1}{4a^2}I[i, j] * \nabla^2 f(i, j)
 \tag{EQ 1.33}$$

The resulting image  $\mathbf{B} = \mathbf{S} - \mathbf{I}$  is the *band-limited Laplacian* of the image. From this the *binary Laplacian image* (**BLI**) is obtained by setting all of the positive valued pixels in  $\mathbf{B}$  to 1 and all others to 0; this is calculated by the C function `compute_bli` in the ISEF source code provided. The candidate edge pixels are on the boundaries of the regions in **BLI**, which correspond to the zero crossings. These could be used as edges, but some additional enhancements improve the quality of the edge pixels identified by the algorithm.

The first improvement is the use of *false zero-crossing suppression*, which is related to the non-maximum suppression performed in the Canny approach. At the location of an edge pixel there will be a zero crossing in the second derivative of the filtered image. This means that the gradient at that point is either a maximum or a minimum. If the second derivative changes sign from positive to negative this is called a *positive zero crossing*, and if it changes from negative to positive it is called a *negative zero crossing*. We will allow positive zero crossings to have a positive gradient, and negative zero crossings to have a negative gradient. All other zero crossings are assumed to be false (spurious) and are not considered to correspond to an edge. This is implemented in the function `is_candidate_edge` in the ISEF code.

In situations where the original image is very noisy a standard thresholding method may not be sufficient. The edge pixels could be thresholded using a global threshold applied to the gradient, but Shen and Castan suggest an *adaptive gradient method*. A window with fixed width  $\mathbf{W}$  is centered at candidate edge pixels found in the **BLI**. If this is indeed an edge pixel, then the window will contain two regions of differing grey level separated by an edge (zero crossing contour). The best estimate of the gradient at that point should be the difference in level between the two regions, where one region corresponds to the zero pixels in the BLI and the other corresponds to the one-valued pixels. The function `compute_adaptive_gradient` performs this activity.

Finally, a hysteresis thresholding method is applied to the edges. This algorithm is basically the same as the one used in the Canny algorithm, adapted for use on an image where edges are marked by zero crossings. The C function `threshold_edges` performs hysteresis thresholding.

## 1.6 A Comparison of Two Optimal Edge Detectors

---

The two signal edge detectors examined in this chapter are the Canny operator and the Shen-Castan method. A good way to end the discussion of edge detection may be to compare these two approaches against each other.

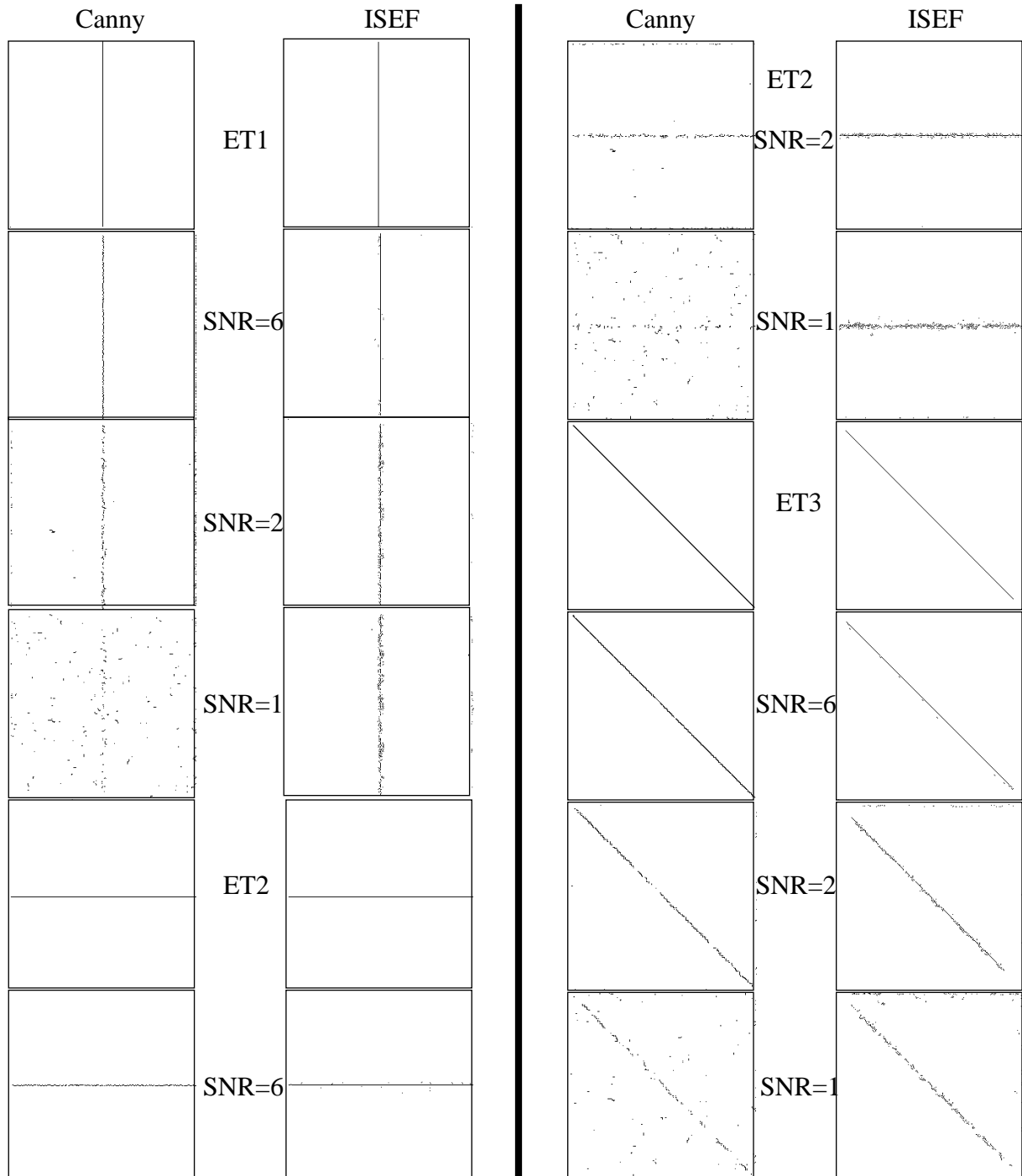
To summarize the two methods, the Canny algorithm convolves the image with the derivative of a Gaussian, then performs non-maximum suppression and hysteresis thresholding; the Shen-Castan algorithm convolves the image with the Infinite Symmetric Exponential Filter, computes the binary Laplacian image, suppresses false zero crossings, performs adaptive gradient thresholding, and finally also applies hysteresis thresholding. In both methods, as with Marr and Hildreth, the authors suggest the use of multiple resolutions.

Both algorithms offer user specified parameters, which can be useful for tuning the method to a particular class of images. The parameters are:

<b>Canny</b>	<b>Shen-Castan (ISEF)</b>
Sigma (standard deviation)	$0 \leq b \leq 1.0$ (smoothing factor)
High hysteresis threshold	High hysteresis threshold
Low hysteresis threshold	Low hysteresis threshold
	Width of window for adaptive gradient
	Thinning factor

The algorithms were implemented according to the specification laid out in the original articles describing them. It should be pointed out that the various parts of the algorithms could be applied to both methods; for example, a thinning factor could be added to Canny's algorithm, or it could be implemented using recursive filters. Exploring all possible permutations and combinations would be a massive undertaking.

Figure 1.16 shows the result of applying the Canny and the Shen-Castan edge detectors to the test images. Because the Canny implementation uses a wrap-around scheme when performing the convolution, the areas near the boundary of the image are occupied with black pixels, although sometimes with what appears to be noise. The ISEF implementation uses recursive filters, and the wrap-around was more difficult to implement; it was not, in fact, implemented. Instead, the image was embedded in a larger one before processing. As a result, the boundary of these images is mostly white where the convolution mask exceeded the image.



**FIGURE 1.16** - Side by side comparison of the output of the Canny and Shen-Castan (ISEF) edge detectors. All of the test images from Figure 1.8 have been processed by both algorithms, and the output appears here and on the next page.

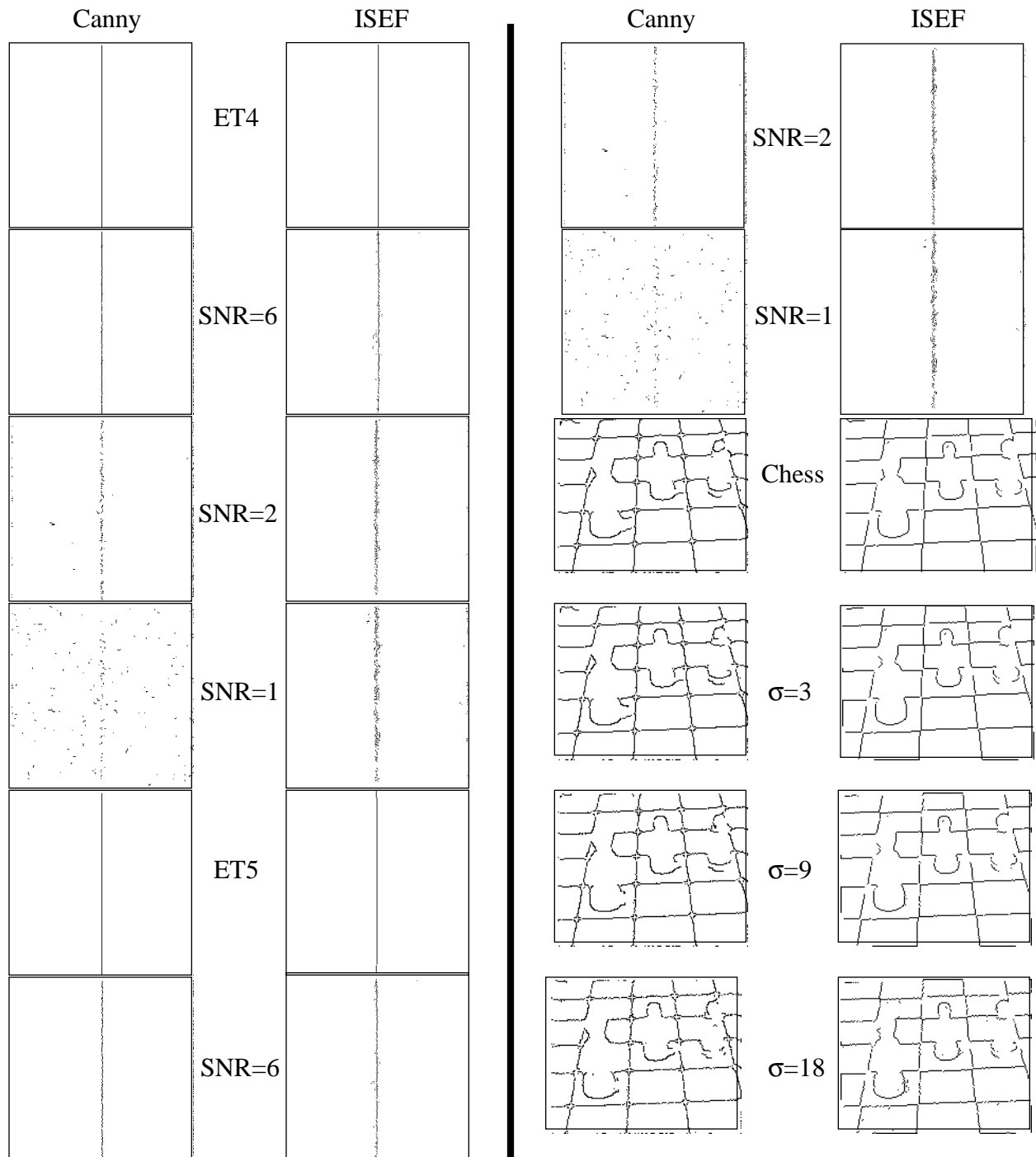


FIGURE 1.16 (continued) Comparison of Canny and Shen-Castan edge detectors.

The two methods were evaluated using E1 and E2 even though flaws have been found with E1. ISEF seems to have the advantage as noise

---

**TABLE 1.6** Evaluation of Canny VS ISEF: E1

Image	Algorithm	No Noise	SNR=6	SNR=2	SNR=1
ET1	Canny	0.9651	0.9498	0.5968	0.1708
	ISEF	0.9689	0.9285	0.7929	0.7036
ET2	Canny	0.9650	0.9155	0.6991	0.2530
	ISEF	0.9650	0.9338	0.8269	0.7170
ET3	Canny	0.9726	0.9641	0.8856	0.4730
	ISEF	0.8776	0.9015	0.7347	0.5238
ET4	Canny	0.5157	0.5092	0.3201	0.1103
	ISEF	0.4686	0.4787	0.4599	0.4227
ET5	Canny	0.5024	0.4738	0.3008	0.0955
	ISEF	0.4957	0.4831	0.4671	0.4074

becomes greater, at least for the E1 metric; Canny has the advantage using the E2 metric. Overall, the ISEF edge detector is ranked first by a slight margin over Canny, which is second. Marr-Hildreth is third, followed by Kirsch, Sobel,  $\nabla_2$ , and  $\nabla_1$  in that order. The comparison

---

**TABLE 1.7** Evaluation of Canny VS ISEF: E2

Image	Algorithm	No Noise	SNR=6	SNR=2	SNR=1
ET1	Canny	1.0000	0.5152	0.5402	0.5687
	ISEF	1.0000	0.9182	0.5756	0.5147
ET2	Canny	1.0000	0.6039	0.5518	0.5726
	ISEF	1.0000	0.9462	0.6018	0.5209
ET3	Canny	0.9291	0.7541	0.6032	0.5899
	ISEF	0.9965	0.9424	0.5204	0.4829
ET4	Canny	1.0000	0.7967	0.5396	0.5681
	ISEF	1.0000	0.5382	0.5193	0.5096
ET5	Canny	1.0000	0.5319	0.5269	0.5706
	ISEF	0.9900	0.6162	0.5243	0.5123

between Canny and ISEF does depend on the parameters selected in each case, and it is likely that better evaluations can be found that use a better choice of parameters. In some of these the Canny edge detector will come out ahead, and in some the ISEF method will win. The best set of parameters for a particular image is not known, and so ultimately the user is left to judge the methods.

---

## 1.7 Bibliography

---

Abdou, I.E., and Pratt, W.K., *Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors*, **Proceedings of the IEEE**, Vol. 67 No. 5, May 1979. Pp. 753-763.

Canny, J., *A Computational Approach to Edge Detection*, **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMI-8, No. 6, November, 1986. Pp. 679-698.

Deutsch, E. S., and Fram, J.R., *A Quantitative Study of Orientation Bias of Some Edge Detector Schemes*, **IEEE Transactions on Computers**, Vol. C-27, No. 3, March, 1978. Pp. 205-213.

Grimson, W.E.L., *From Images to Surfaces*, **MIT Press**, Cambridge, MA. 1981.

Kaplan, W., *Advanced Calculus* (2nd. edition), **Addison Wesley**, Reading, Mass. 1973.

Kirsch, R.A., *Computer Determination of the Constituent Structure of Biological Images*, **Computers and Biomedical Research**, Vol. 4, 1971. Pp. 315-328.

Kitchen, L. and Rosenfeld, A., *Edge Evaluation Using Local Edge Coherence*, **IEEE Transactions on Systems, Man, and Cybernetics**, Vol. SMC-11, No. 9, Sept. 1981. Pp. 597-605.

Marr, D. and Hildreth, E., *Theory of Edge Detection*, **Proceedings of the Royal Society of London**, Series B, Vol. 207, 1980. Pp. 187-217

Nalwa, V.S. and Binford, T.O., *On Detecting Edges*, **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMI-8, No. 6, Nov. 1986. Pp. 699-714.

Prager, J. M., *Extracting and Labeling Boundary Segments in Natural Scenes*, **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMI-2, No. 1, January, 1980. Pp. 16-27.

Pratt, W.K., *Digital Image Processing*, **John Wiley & Sons**, New York. 1978.



Shah, M., Sood, A., and Jain, R., *Pulse and Staircase Edge Models*, **Computer Vision, Graphics, and Image Processing**, Vol. 34, 1986. Pp. 321-343.

Shen, J. and Castan, S., *An Optimal Linear Operator for Step Edge Detection*, **Computer Vision, Graphics, and Image Processing: Graphical Models and Understanding**, Vol. 54, No. 2, March, 1992. Pp. 112-133.

Torre, V. and Poggio, T. A., *On Edge Detection*, **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMI-8, No. 2, March, 1986. Pp. 147-163.

## 1.8 Source code for the Marr-Hildreth edge detector

```

/* Marr/Hildreth edge detection */

#include <math.h>
#include <stdio.h>
#define MAX
#include "lib.h"

float ** f2d (int nr, int nc);
void convolution (IMAGE im, float **mask, int nr, int nc, float **res,
    int NR, int NC);
float gauss(float x, float sigma);
float LoG (float x, float sigma);
float meanGauss (float x, float sigma);
void marr (float s, IMAGE im);
void dolap (float **x, int nr, int nc, float **y);
void zero_cross (float **lapim, IMAGE im);
float norm (float x, float y);
float distance (float a, float b, float c, float d);

void main (int argc, char *argv[])
{
    int i,j,n;
    float s=1.0;
    FILE *params;
    IMAGE im1, im2;

    /* Read parameters from the file marr.par */
    if (argc > 2)
        sscanf (argv[2], "%f", &s);
    else
    {
        params = fopen ("marr.par", "r");
        if (params)
        {
            fscanf (params, "%f", &s); /* Gaussian standard deviation */
            fclose (params);
        }
    }
    printf ("Standard deviation= %lf\n", s);

    /* Command line: input file name */
    if (argc < 2)
    {
        printf ("USAGE: marr <filename> <standard deviation>\n");
        printf ("Marr edge detector - reads a PGM format file and\n");
        printf (" detects edges, creating 'marr.pgm'.\n");
        exit (1);
    }

    im1 = Input_PBM (argv[1]);
    if (im1 == 0)
    {
        printf ("No input image ('%s')\n", argv[1]);
        exit (2);
    }
}

```

```

    im2 = newimage (im1->info->nr, im1->info->nc);
    for (i=0; i<im1->info->nr; i++)
        for (j=0; j<im1->info->nc; j++)
            im2->data[i][j] = im1->data[i][j];

/* Apply the filter */
    marr (s-0.8, im1);
    marr (s+0.8, im2);

    for (i=0; i<im1->info->nr; i++)
        for (j=0; j<im1->info->nc; j++)
            if (im1->data[i][j] > 0 && im2->data[i][j] > 0)
                im1->data[i][j] = 0;
            else im1->data[i][j] = 255;

    Output_PBM (im1, "marr.pgm");
    printf ("Done. File is 'marr.pgm'.\n");
}

float norm (float x, float y)
{
    return (float) sqrt ( (double)(x*x + y*y) );
}

float distance (float a, float b, float c, float d)
{
    return norm ( (a-c), (b-d) );
}

void marr (float s, IMAGE im)
{
    int width;
    float **smx;
    int i,j,k,n;
    float **lgau, z;

/* Create a Gaussian and a derivative of Gaussian filter mask */
    width = 3.35*s + 0.33;
    n = width+width + 1;
    printf ("Smoothing with a Gaussian of size %dx%d\n", n, n);
    lgau = f2d (n, n);
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            lgau[i][j] = LoG (distance ((float)i, (float)j,
                                     (float)width, (float)width), s);

/* Convolution of source image with a Gaussian in X and Y directions */
    smx = f2d (im->info->nr, im->info->nc);
    printf ("Convolution with LoG:\n");
    convolution (im, lgau, n, n, smx, im->info->nr, im->info->nc);

/* Locate the zero crossings */
    printf ("Zero crossings:\n");
    zero_cross (smx, im);

/* Clear the boundary */
    for (i=0; i<im->info->nr; i++)
    {
        for (j=0; j<=width; j++) im->data[i][j] = 0;
    }
}

```

```

        for (j=im->info->nc-width-1; j<im->info->nc; j++)
            im->data[i][j] = 0;
    }
    for (j=0; j<im->info->nc; j++)
    {
        for (i=0; i<= width; i++) im->data[i][j] = 0;
        for (i=im->info->nr-width-1; i<im->info->nr; i++)
            im->data[i][j] = 0;
    }

    free(smx[0]); free(smx);
    free(lgau[0]); free(lgau);
}

/* Gaussian*/
float gauss(float x, float sigma)
{
    return (float)exp((double) ((-x*x)/(2*sigma*sigma)));
}

float meanGauss (float x, float sigma)
{
    float z;

    z = (gauss(x,sigma)+gauss(x+0.5,sigma)+gauss(x-0.5,sigma))/3.0;
    z = z/(PI*2.0*sigma*sigma);
    return z;
}

float LoG (float x, float sigma)
{
    float x1;

    x1 = gauss (x, sigma);
    return (x*x-2*sigma*sigma)/((sigma*sigma*sigma*sigma) * x1);
}

/*
float ** f2d (int nr, int nc)
{
    float **x, *y;
    int i;

    x = (float **)calloc ( nr, sizeof (float *) );
    y = (float *) calloc ( nr*nc, sizeof (float) );
    if ( (x==0) || (y==0) )
    {
        fprintf (stderr, "Out of storage: F2D.\n");
        exit (1);
    }
    for (i=0; i<nr; i++)
        x[i] = y+i*nc;
    return x;
}
*/

void convolution (IMAGE im, float **mask, int nr, int nc, float **res,
    int NR, int NC)
{

```

```

int i,j,ii,jj, n, m, k, kk;
float x, y;

k = nr/2; kk = nc/2;
for (i=0; i<NR; i++)
    for (j=0; j<NC; j++)
    {
        x = 0.0;
        for (ii=0; ii<nr; ii++)
        {
            n = i - k + ii;
            if (n<0 || n>=NR) continue;
            for (jj=0; jj<nc; jj++)
            {
                m = j - kk + jj;
                if (m<0 || m>=NC) continue;
                x += mask[ii][jj] * (float)(im->data[n][m]);
            }
        }
        res[i][j] = x;
    }
}

void zero_cross (float **lapim, IMAGE im)
{
    int i,j,k,n,m, dx, dy;
    float x, y, z;
    int xi,xj,yi,yj, count = 0;
    IMAGE deriv;

    for (i=1; i<im->info->nr-1; i++)
        for (j=1; j<im->info->nc-1; j++)
        {
            im->data[i][j] = 0;
            if(lapim[i-1][j]*lapim[i+1][j]<0) {im->data[i][j]=255; continue;}
            if(lapim[i][j-1]*lapim[i][j+1]<0) {im->data[i][j]=255; continue;}
            if(lapim[i+1][j-1]*lapim[i-1][j+1]<0) {im->data[i][j]=255; continue;}
            if(lapim[i-1][j-1]*lapim[i+1][j+1]<0) {im->data[i][j]=255; continue;}
        }
}

/* An alternative way to compute a Laplacian*/
void dolap (float **x, int nr, int nc, float **y)
{
    int i,j,k,n,m;
    float u,v;

    for (i=1; i<nr-1; i++)
        for (j=1; j<nc-1; j++)
        {
            y[i][j] = (x[i][j+1]+x[i][j-1]+x[i-1][j]+x[i+1][j]) - 4*x[i][j];
            if (u>y[i][j]) u = y[i][j];
            if (v<y[i][j]) v = y[i][j];
        }
}

```

## 1.9 Source code for the Canny edge detector

```

#include <math.h>
#include <stdio.h>
#define MAX
#include "lib.h"

/* Scale floating point magnitudes and angles to 8 bits */
#define ORI_SCALE 40.0
#define MAG_SCALE 20.0

/* Biggest possible filter mask */
#define MAX_MASK_SIZE 20

/* Fraction of pixels that should be above the HIGH threshold */
float ratio = 0.1;
int WIDTH = 0;

float ** f2d (int nr, int nc);
int trace (int i, int j, int low, IMAGE im, IMAGE mag, IMAGE ori);
float gauss(float x, float sigma);
float dGauss (float x, float sigma);
float meanGauss (float x, float sigma);
void hysteresis (int high, int low, IMAGE im, IMAGE mag, IMAGE oriim);
void canny (float s, IMAGE im, IMAGE mag, IMAGE ori);
void seperable_convolution (IMAGE im, float *gau, int width,
                           float **smx, float **smy);
void dxy_seperable_convolution (float** im, int nr, int nc, float *gau,
                               int width, float **sm, int which);
void nonmax_suppress (float **dx, float **dy, int nr, int nc,
                     IMAGE mag, IMAGE ori);
void estimate_thresh (IMAGE mag, int *low, int *hi);

void main (int argc, char *argv[])
{
    int i,j,k,n;
    float s=1.0;
    int low= 0,high=-1;
    FILE *params;
    IMAGE im, magim, oriim;

    /* Command line: input file name */
    if (argc < 2)
    {
        printf ("USAGE: canny <filename>\n");
        printf ("Canny edge detector - reads a PGM format file and\n");
        printf ("detects edges, creating 'canny.pgm'.\n");
        exit (1);
    }
    printf ("CANNY: Apply the Canny edge detector to an image.\n");

    /* Read parameters from the file canny.par */
    params = fopen ("canny.par", "r");
    if (params)
    {
        fscanf (params, "%d", &low);/* Lower threshold */
        fscanf (params, "%d", &high);/* High threshold */
        fscanf (params, "%lf", &s);/* Gaussian standard deviation */
    }

```

```
        printf ("Parameters from canny.par: HIGH: %d LOW %d Sigma %f\n",
                high, low, s);
        fclose (params);
    }
    else printf ("Parameter file 'canny.par' does not exist.\n");

/* Read the input file */
im = Input_PBM (argv[1]);
if (im == 0)
{
    printf ("No input image ('%s')\n", argv[1]);
    exit (2);
}

/* Create local image space */
magim = newimage (im->info->nr, im->info->nc);
if (magim == NULL)
{
    printf ("Out of storage: Magnitude\n");
    exit (1);
}

oriim = newimage (im->info->nr, im->info->nc);
if (oriim == NULL)
{
    printf ("Out of storage: Orientation\n");
    exit (1);
}

/* Apply the filter */
canny (s, im, magim, oriim);

Output_PBM (magim, "mag.pgm");
Output_PBM (oriim, "ori.pgm");

/* Hysteresis thresholding of edge pixels */
hysteresis (high, low, im, magim, oriim);

for (i=0; i<WIDTH; i++)
    for (j=0; j<im->info->nc; j++)
        im->data[i][j] = 255;

for (i=im->info->nr-1; i>im->info->nr-1-WIDTH; i--)
    for (j=0; j<im->info->nc; j++)
        im->data[i][j] = 255;

for (i=0; i<im->info->nr; i++)
    for (j=0; j<WIDTH; j++)
        im->data[i][j] = 255;

for (i=0; i<im->info->nr; i++)
    for (j=im->info->nc-WIDTH-1; j<im->info->nc; j++)
        im->data[i][j] = 255;

Output_PBM (im, "canny.pgm");

printf ("Output files are:\n");
printf (" canny.pgm - edge-only image\n");
printf ("  mag.pgm   - magnitude after non-max suppression.\n");
```

```

        printf (" ori.pgm - angles associated with the edge pixels.\n");
    }

float norm (float x, float y)
{
    return (float) sqrt ( (double)(x*x + y*y) );
}

void canny (float s, IMAGE im, IMAGE mag, IMAGE ori)
{
    int width;
    float **smx,**smy;
    float **dx,**dy;
    int i,j,k,n;
    float gau[MAX_MASK_SIZE], dgau[MAX_MASK_SIZE], z;

    /* Create a Gaussian and a derivative of Gaussian filter mask */
    for(i=0; i<MAX_MASK_SIZE; i++)
    {
        gau[i] = meanGauss ((float)i, s);
        if (gau[i] < 0.005)
        {
            width = i;
            break;
        }
        dgau[i] = dGauss ((float)i, s);
    }

    n = width+width + 1;
    WIDTH = width/2;
    printf ("Smoothing with a Gaussian (width = %d) ...\n", n);

    smx = f2d (im->info->nr, im->info->nc);
    smy = f2d (im->info->nr, im->info->nc);

    /* Convolution of source image with a Gaussian in X and Y directions */
    seperable_convolution (im, gau, width, smx, smy);

    /* Now convolve smoothed data with a derivative */
    printf ("Convolution with the derivative of a Gaussian...\n");
    dx = f2d (im->info->nr, im->info->nc);
    dxy_seperable_convolution (smx, im->info->nr, im->info->nc,
        dgau, width, dx, 1);
    free(smx[0]); free(smx);

    dy = f2d (im->info->nr, im->info->nc);
    dxy_seperable_convolution (smy, im->info->nr, im->info->nc,
        dgau, width, dy, 0);
    free(smy[0]); free(smy);

    /* Create an image of the norm of dx,dy */
    for (i=0; i<im->info->nr; i++)
        for (j=0; j<im->info->nc; j++)
        {
            z = norm (dx[i][j], dy[i][j]);
            mag->data[i][j] = (unsigned char)(z*MAG_SCALE);
        }

    /* Non-maximum suppression - edge pixels should be a local max */

```



```

        nonmax_suppress (dx, dy, (int)im->info->nr, (int)im->info->nc, mag, ori);

        free(dx[0]); free(dx);
        free(dy[0]); free(dy);
    }

    /* Gaussian*/
    float gauss(float x, float sigma)
    {
        return (float)exp((double) ((-x*x)/(2*sigma*sigma)));
    }

    float meanGauss (float x, float sigma)
    {
        float z;

        z = (gauss(x,sigma)+gauss(x+0.5,sigma)+gauss(x-0.5,sigma))/3.0;
        z = z/(PI*2.0*sigma*sigma);
        return z;
    }

    /* First derivative of Gaussian*/
    float dGauss (float x, float sigma)
    {
        return -x/(sigma*sigma) * gauss(x, sigma);
    }

    /* HYSTERESIS thresholding of edge pixels. Starting at pixels with a
       value greater than the HIGH threshold, trace a connected sequence
       of pixels that have a value greater than the LOW threshold. */
    void hysteresis (int high, int low, IMAGE im, IMAGE mag, IMAGE oriim)
    {
        int i,j,k;

        printf ("Beginning hysteresis thresholding...\n");
        for (i=0; i<im->info->nr; i++)
            for (j=0; j<im->info->nc; j++)
                im->data[i][j] = 0;

        if (high<low)
        {
            estimate_thresh (mag, &high, &low);
            printf ("Hysteresis thresholds (from image): HI %d LOW %d\n",
                    high, low);
        }

        /* For each edge with a magnitude above the high threshold, begin
           tracing edge pixels that are above the low threshold. */

        for (i=0; i<im->info->nr; i++)
            for (j=0; j<im->info->nc; j++)
                if (mag->data[i][j] >= high)
                    trace (i, j, low, im, mag, oriim);

        /* Make the edge black (to be the same as the other methods) */
        for (i=0; i<im->info->nr; i++)
            for (j=0; j<im->info->nc; j++)
                if (im->data[i][j] == 0) im->data[i][j] = 255;
    }

```

```

        else im->data[i][j] = 0;
    }

    /* TRACE - recursively trace edge pixels that have a threshold > the low
       edge threshold, continuing from the pixel at (i,j). */

int trace (int i, int j, int low, IMAGE im, IMAGE mag, IMAGE ori)
{
    int n,m;
    char flag = 0;

    if (im->data[i][j] == 0)
    {
        im->data[i][j] = 255;
        flag=0;
        for (n= -1; n<=1; n++)
        {
            for(m= -1; m<=1; m++)
            {
                if (i==0 && m==0) continue;
                if (range(mag, i+n, j+m) && mag->data[i+n][j+m] >= low)
                if (trace(i+n, j+m, low, im, mag, ori))
                {
                    flag=1;
                    break;
                }
            }
            if (flag) break;
        }
        return(1);
    }
    return(0);
}

void seperable_convolution (IMAGE im, float *gau, int width,
                           float **smx, float **smy)
{
    int i,j,k, I1, I2, nr, nc;
    float x, y;

    nr = im->info->nr;
    nc = im->info->nc;

    for (i=0; i<nr; i++)
        for (j=0; j<nc; j++)
        {
            x = gau[0] * im->data[i][j]; y = gau[0] * im->data[i][j];
            for (k=1; k<width; k++)
            {
                I1 = (i+k)%nr; I2 = (i-k+nr)%nr;
                y += gau[k]*im->data[I1][j] + gau[k]*im->data[I2][j];
                I1 = (j+k)%nc; I2 = (j-k+nc)%nc;
                x += gau[k]*im->data[i][I1] + gau[k]*im->data[i][I2];
            }
            smx[i][j] = x; smy[i][j] = y;
        }
}

void dxy_seperable_convolution (float** im, int nr, int nc, float *gau,
```

```

        int width, float **sm, int which)
{
    int i,j,k, I1, I2;
    float x;

    for (i=0; i<nr; i++)
        for (j=0; j<nc; j++)
        {
            x = 0.0;
            for (k=1; k<width; k++)
            {
                if (which == 0)
                {
                    I1 = (i+k)%nr; I2 = (i-k+nr)%nr;
                    x += -gau[k]*im[I1][j] + gau[k]*im[I2][j];
                }
                else
                {
                    I1 = (j+k)%nc; I2 = (j-k+nc)%nc;
                    x += -gau[k]*im[i][I1] + gau[k]*im[i][I2];
                }
            }
            sm[i][j] = x;
        }
}

float ** f2d (int nr, int nc)
{
    float **x, *y;
    int i;

    x = (float **)calloc ( nr, sizeof (float *) );
    y = (float *) calloc ( nr*nc, sizeof (float) );
    if ( (x==0) || (y==0) )
    {
        fprintf (stderr, "Out of storage: F2D.\n");
        exit (1);
    }
    for (i=0; i<nr; i++)
        x[i] = y+i*nc;
    return x;
}

void nonmax_suppress (float **dx, float **dy, int nr, int nc,
                      IMAGE mag, IMAGE ori)
{
    int i,j,k,n,m;
    int top, bottom, left, right;
    float xx, yy, g2, g1, g3, g4, g, xc, yc;

    for (i=1; i<mag->info->nr-1; i++)
    {
        for (j=1; j<mag->info->nc-1; j++)
        {
            mag->data[i][j] = 0;

/* Treat the x and y derivatives as components of a vector */
            xc = dx[i][j];
            yc = dy[i][j];

```

```

        if (fabs(xc)<0.01 && fabs(yC)<0.01) continue;

        g = norm (xc, yc);

/* Follow the gradient direction, as indicated by the direction of
   the vector (xc, yc); retain pixels that are a local maximum.*/

        if (fabs(yC) > fabs(xc))
        {
/* The Y component is biggest, so gradient direction is basically UP/DOWN */
            xx = fabs(xc)/fabs(yC);
            yy = 1.0;

            g2 = norm (dx[i-1][j], dy[i-1][j]);
            g4 = norm (dx[i+1][j], dy[i+1][j]);
            if (xc*yC > 0.0)
            {
                g3 = norm (dx[i+1][j+1], dy[i+1][j+1]);
                g1 = norm (dx[i-1][j-1], dy[i-1][j-1]);
            } else
            {
                g3 = norm (dx[i+1][j-1], dy[i+1][j-1]);
                g1 = norm (dx[i-1][j+1], dy[i-1][j+1]);
            }

        } else
        {
/* The X component is biggest, so gradient direction is basically LEFT/RIGHT */
            xx = fabs(yC)/fabs(xc);
            yy = 1.0;

            g2 = norm (dx[i][j+1], dy[i][j+1]);
            g4 = norm (dx[i][j-1], dy[i][j-1]);
            if (xc*yC > 0.0)
            {
                g3 = norm (dx[i-1][j-1], dy[i-1][j-1]);
                g1 = norm (dx[i+1][j+1], dy[i+1][j+1]);
            } else
            {
                g1 = norm (dx[i-1][j+1], dy[i-1][j+1]);
                g3 = norm (dx[i+1][j-1], dy[i+1][j-1]);
            }

        }

/* Compute the interpolated value of the gradient magnitude */
        if ( (g > (xx*g1 + (yy-xx)*g2)) &&
            (g > (xx*g3 + (yy-xx)*g4)) )
        {
            if (g*MAG_SCALE <= 255)
                mag->data[i][j] = (unsigned char)(g*MAG_SCALE);
            else
                mag->data[i][j] = 255;
            ori->data[i][j] = atan2 (yC, xc) * ORI_SCALE;
        } else
        {
            mag->data[i][j] = 0;
        }

```

```
        ori->data[i][j] = 0;
    }
}
}

void estimate_thresh (IMAGE mag, int *hi, int *low)
{
    int i,j,k, hist[256], count;

    /* Build a histogram of the magnitude image. */
    for (k=0; k<256; k++) hist[k] = 0;

    for (i=WIDTH; i<mag->info->nr-WIDTH; i++)
        for (j=WIDTH; j<mag->info->nc-WIDTH; j++)
            hist[mag->data[i][j]]++;

    /* The high threshold should be > 80 or 90% of the pixels
    j = (int)(ratio*mag->info->nr*mag->info->nc);
    */
    j = mag->info->nr;
    if (j<mag->info->nc) j = mag->info->nc;
    j = (int)(0.9*j);
    k = 255;

    count = hist[255];
    while (count < j)
    {
        k--;
        if (k<0) break;
        count += hist[k];
    }
    *hi = k;

    i=0;
    while (hist[i]==0) i++;

    *low = (*hi+i)/2.0;
}
```

## 1.10 Source code for the Shen-Castan edge detector

```

#include <stdio.h>
#include <string.h>
#include <math.h>
#define MAX
#include "lib.h"

#define OUTLINE 25

/* Function prototypes */
void main( int argc, char **argv);
void shen(IMAGE im, IMAGE res);
void compute_ISEF (float **x, float **y, int nrows, int ncols);
float ** f2d (int nr, int nc);
void apply_ISEF_vertical (float **x, float **y, float **A, float **B,
                          int nrows, int ncols);
void apply_ISEF_horizontal (float **x, float **y, float **A, float **B,
                            int nrows, int ncols);
IMAGE compute_bli (float **buff1, float **buff2, int nrows, int ncols);
void locate_zero_crossings (float **orig, float **smoothed, IMAGE bli,
                            int nrows, int ncols);
void threshold_edges (float **in, IMAGE out, int nrows, int ncols);
int mark_connected (int i, int j, int level);
int is_candidate_edge (IMAGE buff, float **orig, int row, int col);
float compute_adaptive_gradient (IMAGE BLI_buffer, float **orig_buffer,
                                int row, int col);
void estimate_thresh (double *low, double *hi, int nr, int nc);
void debed (IMAGE im, int width);
void embed (IMAGE im, int width);

/* globals for shen operator*/
double b = 0.9; /* smoothing factor 0 < b < 1 */
double low_thresh=20, high_thresh=22; /* threshold for hysteresis */
double ratio = 0.99;
int window_size = 7;
int do_hysteresis = 1;
float **lap; /* keep track of laplacian of image */
int nr, nc; /* nrows, ncols */
IMAGE edges; /* keep track of edge points (thresholded) */
int thinFactor;

void main(int argc, char **argv)
{
    int i,j,n,m;
    IMAGE im, res;
    FILE *params;

    /* Command line args - file name, maybe sigma */
    if (argc < 2)
    {
        printf ("USAGE: shen <imagefile>\n");
        exit (1);
    }
    im = Input_PBM (argv[1]);
    if (im == 0)
    {
        printf ("Can't read input image from '%s'.\n", argv[1]);
        exit (2);
    }

```

```

    }

/* Look for parameter file */
params = fopen ("shen.par", "r");
if (params)
{
    fscanf (params, "%lf", &ratio);
    fscanf (params, "%lf", &b);
    if (b<0) b = 0;
    else if (b>1.0) b = 1.0;
    fscanf (params, "%d", &window_size);
    fscanf (params, "%d", &thinFactor);
    fscanf (params, "%d", &do_hysteresis);

    printf ("Parameters:\n");
    printf (" %% of pixels to be above HIGH threshold: %7.3f\n", ratio);
    printf (" Size of window for adaptive gradient : %3d\n",
            window_size);
    printf (" Thinning factor : %d\n", thinFactor);
    printf (" Smoothing factor : %7.4f\n", b);
    if (do_hysteresis) printf ("Hysteresis thresholding turned on.\n");
    else printf ("Hysteresis thresholding turned off.\n");
    fclose (params);
}
else printf ("Parameter file 'shen.par' does not exist.\n");

embed (im, OUTLINE);

res = newimage (im->info->nr, im->info->nc);
shen (im, res);

debed (res, OUTLINE);

Output_PBM (res, "shen.pgm");
printf ("Output file is 'shen.pgm'\n");
}

void shen (IMAGE im, IMAGE res)
{
    register int i,j;
    float **buffer;
    float **smoothed_buffer;
    IMAGE bli_buffer;

/* Convert the input image to floating point */
    buffer = f2d (im->info->nr, im->info->nc);
    for (i=0; i<im->info->nr; i++)
        for (j=0; j<im->info->nc; j++)
            buffer[i][j] = (float)(im->data[i][j]);

/* Smooth input image using recursively implemented ISEF filter */
    smoothed_buffer = f2d( im->info->nr, im->info->nc);
    compute_ISEF (buffer, smoothed_buffer, im->info->nr, im->info->nc);

/* Compute bli image band-limited laplacian image from smoothed image */
    bli_buffer = compute_bli(smoothed_buffer,
                             buffer,im->info->nr,im->info->nc);

/* Perform edge detection using bli and gradient thresholding */

```

```

locate_zero_crossings (buffer, smoothed_buffer, bli_buffer,
                      im->info->nr, im->info->nc);

free(smoothed_buffer[0]); free(smoothed_buffer);
freeimage (bli_buffer);

threshold_edges (buffer, res, im->info->nr, im->info->nc);
for (i=0; i<im->info->nr; i++)
    for (j=0; j<im->info->nc; j++)
        if (res->data[i][j] > 0) res->data[i][j] = 0;
        else res->data[i][j] = 255;

free(buffer[0]); free(buffer);
}

/* Recursive filter realization of the ISEF
   (Shen and Castan CVIGP March 1992) */
void compute_ISEF (float **x, float **y, int nrows, int ncols)
{
    float **A, **B;

    A = f2d(nrows, ncols); /* store causal component */
    B = f2d(nrows, ncols); /* store anti-causal component */

    /* first apply the filter in the vertical direction (to the rows) */
    apply_ISEF_vertical (x, y, A, B, nrows, ncols);

    /* now apply the filter in the horizontal direction (to the columns) and */
    /* apply this filter to the results of the previous one */
    apply_ISEF_horizontal (y, y, A, B, nrows, ncols);

    /* free up the memory */
    free (B[0]); free(B);
    free (A[0]); free(A);
}

void apply_ISEF_vertical (float **x, float **y, float **A, float **B,
                        int nrows, int ncols)
{
    register int row, col;
    float b1, b2;

    b1 = (1.0 - b)/(1.0 + b);
    b2 = b*b1;

    /* compute boundary conditions */
    for (col=0; col<ncols; col++)
    {
        /* boundary exists for 1st and last column */
        A[0][col] = b1 * x[0][col];
        B[nrows-1][col] = b2 * x[nrows-1][col];
    }

    /* compute causal component */
    for (row=1; row<nrows; row++)
        for (col=0; col<ncols; col++)
            A[row][col] = b1 * x[row][col] + b * A[row-1][col];

```



```

/* compute anti-causal component */
for (row=nrows-2; row>=0; row--)
    for (col=0; col<ncols; col++)
        B[row][col] = b2 * x[row][col] + b * B[row+1][col];

/* boundary case for computing output of first filter */
for (col=0; col<ncols-1; col++)
    y[nrows-1][col] = A[nrows-1][col];

/* now compute the output of the first filter and store in y */
/* this is the sum of the causal and anti-causal components */
for (row=0; row<nrows-2; row++)
    for (col=0; col<ncols-1; col++)
        y[row][col] = A[row][col] + B[row+1][col];
}

void apply_ISEF_horizontal (float **x, float **y, float **A, float **B,
                           int nrows, int ncols)
{
    register int row, col;
    float b1, b2;

    b1 = (1.0 - b)/(1.0 + b);
    b2 = b*b1;

    /* compute boundary conditions */
    for (row=0; row<nrows; row++)
    {
        A[row][0] = b1 * x[row][0];
        B[row][ncols-1] = b2 * x[row][ncols-1];
    }

    /* compute causal component */
    for (col=1; col<ncols; col++)
        for (row=0; row<nrows; row++)
            A[row][col] = b1 * x[row][col] + b * A[row][col-1];

    /* compute anti-causal component */
    for (col=ncols-2; col>=0; col--)
        for (row=0; row<nrows; row++)
            B[row][col] = b2 * x[row][col] + b * B[row][col+1];

    /* boundary case for computing output of first filter */
    for (row=0; row<nrows; row++)
        y[row][ncols-1] = A[row][ncols-1];

    /* now compute the output of the second filter and store in y */
    /* this is the sum of the causal and anti-causal components */
    for (row=0; row<nrows; row++)
        for (col=0; col<ncols-1; col++)
            y[row][col] = A[row][col] + B[row][col+1];
}

/* compute the band-limited laplacian of the input image */
IMAGE compute_bli (float **buff1, float **buff2, int nrows, int ncols)
{
    register int row, col;
    IMAGE bli_buffer;

```

```

    bli_buffer = newimage(nrows, ncols);
    for (row=0; row<nrows; row++)
        for (col=0; col<ncols; col++)
            bli_buffer->data[row][col] = 0;

/* The BLI is computed by taking the difference between the smoothed image */
/* and the original image. In Shen and Castan's paper this is shown to */
/* approximate the band-limited laplacian of the image. The bli is then */
/* made by setting all values in the bli to 1 where the laplacian is */
/* positive and 0 otherwise. */
    for (row=0; row<nrows; row++)
        for (col=0; col<ncols; col++)
        {
            if (row<OUTLINE || row >= nrows-OUTLINE ||
                col<OUTLINE || col >= ncols-OUTLINE) continue;
            bli_buffer->data[row][col] =
                ((buff1[row][col] - buff2[row][col]) > 0.0);
        }
    return bli_buffer;
}

void locate_zero_crossings (float **orig, float **smoothed, IMAGE bli,
                           int nrows, int ncols)
{
    register int row, col;

    for (row=0; row<nrows; row++)
    {
        for (col=0; col<ncols; col++)
        {
            /* ignore pixels around the boundary of the image */
            if (row<OUTLINE || row >= nrows-OUTLINE ||
                col<OUTLINE || col >= ncols-OUTLINE)
            {
                orig[row][col] = 0.0;
            }

            /* next check if pixel is a zero-crossing of the laplacian */
            else if (is_candidate_edge (bli, smoothed, row, col))
            {
                /* now do gradient thresholding */
                float grad = compute_adaptive_gradient (bli, smoothed, row, col);
                orig[row][col] = grad;
            }
            else orig[row][col] = 0.0;
        }
    }
}

void threshold_edges (float **in, IMAGE out, int nrows, int ncols)
{
    register int i, j;

    lap = in;
    edges = out;
    nr = nrows;

```

```

nc = ncols;

estimate_thresh (&low_thresh, &high_thresh, nr, nc);
if (!do_hysteresis)
    low_thresh = high_thresh;

for (i=0; i<nrows; i++)
    for (j=0; j<ncols; j++)
        edges->data[i][j] = 0;

for (i=0; i<nrows; i++)
    for (j=0; j<ncols; j++)
    {
        if (i<OUTLINE || i >= nrows-OUTLINE ||
            j<OUTLINE || j >= ncols-OUTLINE) continue;

/* only check a contour if it is above high_thresh */
        if ((lap[i][j]) > high_thresh)

/* mark all connected points above low thresh */
            mark_connected (i,j,0);
    }

    for (i=0; i<nrows; i++)/* erase all points which were 255 */
        for (j=0; j<ncols; j++)
            if (edges->data[i][j] == 255) edges->data[i][j] = 0;
}

/* return true if it marked something */
int mark_connected (int i, int j, int level)
{
    int notChainEnd;

/* stop if you go off the edge of the image */
    if (i >= nr || i < 0 || j >= nc || j < 0) return 0;

/* stop if the point has already been visited */
    if (edges->data[i][j] != 0) return 0;

/* stop when you hit an image boundary */
    if (lap[i][j] == 0.0) return 0;

    if ((lap[i][j]) > low_thresh)
    {
        edges->data[i][j] = 1;
    }
    else
    {
        edges->data[i][j] = 255;
    }

    notChainEnd = 0;

    notChainEnd = mark_connected(i, j+1, level+1);
    notChainEnd = mark_connected(i, j-1, level+1);
    notChainEnd = mark_connected(i+1, j+1, level+1);
    notChainEnd = mark_connected(i+1, j, level+1);
    notChainEnd = mark_connected(i+1, j-1, level+1);
    notChainEnd = mark_connected(i-1, j-1, level+1);

```

```

notChainEnd |= mark_connected(i-1,j , level+1);
notChainEnd |= mark_connected(i-1,j+1, level+1);

if (notChainEnd && ( level > 0 ) )
{
/* do some contour thinning */
if ( thinFactor > 0 )
if ( (level%thinFactor) != 0 )
{
/* delete this point */
edges->data[i][j] = 255;
}
}

return 1;
}

/* finds zero-crossings in laplacian (buff) orig is the smoothed image */
int is_candidate_edge (IMAGE buff, float **orig, int row, int col)
{
/* A positive z-c must have a positive 1st derivative, where positive
z-c means the second derivative goes from + to - as we cross the edge */

if (buff->data[row][col] == 1 && buff->data[row+1][col] == 0) /* positive
z-c */
{
if (orig[row+1][col] - orig[row-1][col] > 0) return 1;
else return 0;
}
else if (buff->data[row][col] == 1 && buff->data[row][col+1] == 0) /*
positive z-c */
{
if (orig[row][col+1] - orig[row][col-1] > 0) return 1;
else return 0;
}
else if ( buff->data[row][col] == 1 && buff->data[row-1][col] == 0) /*
negative z-c */
{
if (orig[row+1][col] - orig[row-1][col] < 0) return 1;
else return 0;
}
else if (buff->data[row][col] == 1 && buff->data[row][col-1] == 0) /*
negative z-c */
{
if (orig[row][col+1] - orig[row][col-1] < 0) return 1;
else return 0;
}
else
/* not a z-c */
return 0;
}

float compute_adaptive_gradient (IMAGE BLI_buffer, float **orig_buffer,
int row, int col)
{
register int i, j;
float sum_on, sum_off;
float avg_on, avg_off;
int num_on, num_off;

```

```

sum_on = sum_off = 0.0;
num_on = num_off = 0;

for (i= (-window_size/2); i<=(window_size/2); i++)
{
    for (j=(-window_size/2); j<=(window_size/2); j++)
    {
        if (BLI_buffer->data[row+i][col+j])
        {
            sum_on += orig_buffer[row+i][col+j];
            num_on++;
        }
        else
        {
            sum_off += orig_buffer[row+i][col+j];
            num_off++;
        }
    }
}

if (sum_off) avg_off = sum_off / num_off;
else avg_off = 0;

if (sum_on) avg_on = sum_on / num_on;
else avg_on = 0;

return (avg_off - avg_on);
}

void estimate_thresh (double *low, double *hi, int nr, int nc)
{
    float vmax, vmin, scale, x;
    int i,j,k, hist[256], count;

    /* Build a histogram of the Laplacian image. */
    vmin = vmax = fabs((float)(lap[20][20]));
    for (i=0; i<nr; i++)
        for (j=0; j<nc; j++)
        {
            if (i<OUTLINE || i >= nr-OUTLINE ||
                j<OUTLINE || j >= nc-OUTLINE) continue;
            x = lap[i][j];
            if (vmin > x) vmin = x;
            if (vmax < x) vmax = x;
        }
    for (k=0; k<256; k++) hist[k] = 0;

    scale = 256.0/(vmax-vmin + 1);

    for (i=0; i<nr; i++)
        for (j=0; j<nc; j++)
        {
            if (i<OUTLINE || i >= nr-OUTLINE ||
                j<OUTLINE || j >= nc-OUTLINE) continue;
            x = lap[i][j];
            k = (int)((x - vmin)*scale);
            hist[k] += 1;
        }
}

```

```

/* The high threshold should be > 80 or 90% of the pixels */
k = 255;
j = (int)(ratio*nr*nc);
count = hist[255];
while (count < j)
{
    k--;
    if (k<0) break;
    count += hist[k];
}
*hi = (double)k/scale + vmin ;
*low = (*hi)/2;
}

void embed (IMAGE im, int width)
{
    int i,j,I,J;
    IMAGE new;

    width += 2;
    new = newimage (im->info->nr+width+width, im->info->nc+width+width);
    for (i=0; i<new->info->nr; i++)
        for (j=0; j<new->info->nc; j++)
        {
            I = (i-width+im->info->nr)%im->info->nr;
            J = (j-width+im->info->nc)%im->info->nc;
            new->data[i][j] = im->data[I][J];
        }

    free (im->info);
    free(im->data[0]); free(im->data);
    im->info = new->info;
    im->data = new->data;
}

void debed (IMAGE im, int width)
{
    int i,j;
    IMAGE old;

    width +=2;
    old = newimage (im->info->nr-width-width, im->info->nc-width-width);
    for (i=0; i<old->info->nr-1; i++)
    {
        for (j=1; j<old->info->nc; j++)
        {
            old->data[i][j] = im->data[i+width][j+width];
            old->data[old->info->nr-1][j] = 255;
        }
        old->data[i][0] = 255;
    }

    free (im->info);
    free(im->data[0]); free(im->data);
    im->info = old->info;
    im->data = old->data;
}

```