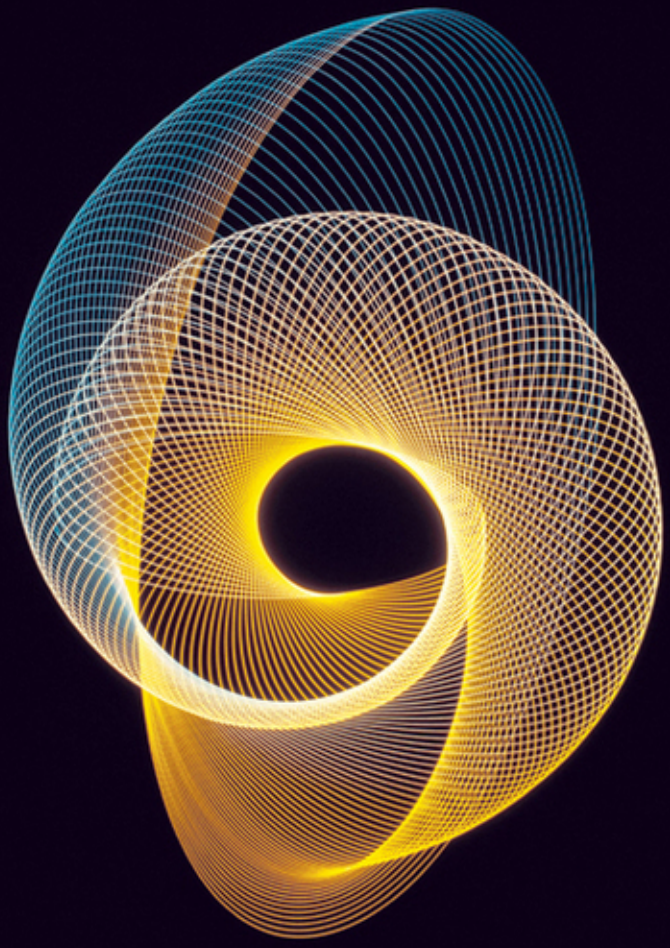
J. R. Parker

Algorithms for Image Processing and Computer Vision



SECOND EDITION

Algorithms for Image Processing and Computer Vision

Second Edition

Algorithms for Image Processing and Computer Vision

Second Edition

J.R. Parker

Wiley Publishing, Inc.

Algorithms for Image Processing and Computer Vision, Second Edition

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada

ISBN: 978-0-470-64385-3
ISBN: 978-1-118-02188-0 (ebk)
ISBN: 978-1-118-02189-7 (ebk)
ISBN: 978-1-118-01962-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2010939957

"Sin lies only in hurting other people unnecessarily.
All other 'sins' are invented nonsense.
(Hurting yourself is not a sin—just stupid.)"

—Robert A. Heinlein

Thanks, Bob.

Credits

Executive Editor
Carol Long

Project Editor
John Sleeva

Technical Editor
Kostas Terzidis

Production Editor
Daniel Scribner

Copy Editor
Christopher Jones

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Freelancer Editorial Manager
Rosemarie Graham

Marketing Manager
Ashley Zurcher

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Proofreaders
Nancy Hanger, Paul Sagan

Indexer
Ron Strauss

Cover Image
Ryan Sneed

Cover Designer
© GYRO PHOTOGRAPHY/
amanaimagesRB/Getty Images

About the Author

J.R. Parker is a computer expert and teacher, with special interests in image processing and vision, video game technologies, and computer simulations. With a Ph.D. in Informatics from the State University of Gent, Dr. Parker has taught computer science, art, and drama at the University of Calgary in Canada, where he is a full professor. He has more than 150 technical papers and four books to his credit, as well as video games such as the *Booze Cruise*, a simulation of impaired driving designed to demonstrate its folly, and a number of educational games. Jim lives on a small ranch near Cochrane, Alberta, Canada with family and a host of legged and winged creatures.

About the Technical Editor

Kostas Terzidis is an Associate Professor at the Harvard Graduate School of Design. He holds a Ph.D. in Architecture from the University of Michigan (1994), a Masters of Architecture from Ohio State University (1989), and a Diploma of Engineering from the Aristotle University of Thessaloniki (1986). His most recent work is in the development of theories and techniques for the use of algorithms in architecture. His book *Expressive Form: A Conceptual Approach to Computational Design*, published by London-based Spon Press (2003), offers a unique perspective on the use of computation as it relates to aesthetics, specifically in architecture and design. His book *Algorithmic Architecture* (Architectural Press/Elsevier, 2006) provides an ontological investigation into the terms, concepts, and processes of algorithmic architecture and provides a theoretical framework for design implementations. His latest book, *Algorithms for Visual Design* (Wiley, 2009), provides students, programmers, and researchers the technical, theoretical, and design means to develop computer code that will allow them to experiment with design problems.

Acknowledgments

Thanks this time to Sonny Chan, for the inspiration for the parallel computing chapter, to Jeff Boyd, for introducing me repeatedly to OpenCV, and to Ralph Huntsinger and Ghislain C. Vansteenkiste, for getting me into and successfully out of my Ph.D. program.

Almost all the images used in this book were created by me, using an IBM PC with a frame grabber and a Sony CCD camera, an HP scanner, and a Sony Eyetoy as a webcam. Credits for the few images that were not acquired in this way are as follows:

Corel Corporation made available the color image of the grasshopper on a leaf shown in Figure 3.33, and also was the origin of the example search images in Figure 10.5.

The sample images in Figure 10.1 were a part of the ALOI data set, use of which was allowed by J. M. Geusebroek.

Thanks to Big Hill Veterinary Clinic in Cochrane, Alberta, Canada, for the X-ray image shown in Figure 3.10e.

Finally, thanks to Dr. N. Wardlaw, of the University of Calgary Department of Geology, for the geological micropore image of Figure 3.16.

Most importantly, I need to thank my family: my wife, Katrin, and children, Bailey and Max. They sacrificed time and energy so that this work could be completed. I appreciate it and hope that the effort has been worthwhile.

Contents at a Glance

Contents

Preface

Humans still obtain the vast majority of their sensory input through their visual system, and an enormous effort has been made to artificially enhance this sense. Eyeglasses, binoculars, telescopes, radar, infrared sensors, and photomultipliers all function to improve our view of the world and the universe. We even have telescopes in orbit (eyes outside the atmosphere) and many of those ''see'' in other spectra: infrared, ultraviolet, X-rays. These give us views that we could not have imagined only a few years ago, and in colors that we'll never see with the naked eye. The computer has been essential for creating the incredible images we've all seen from these devices.

When the first edition of this book was written, the Hubble Space Telescope was in orbit and producing images at a great rate. It and the European Hipparcos telescope were the only optical instruments above the atmosphere. Now there is COROT, Kepler, MOST (Canada's space telescope), and Swift Gamma Ray Burst Explorer. In addition, there is the Spitzer (infrared), Chandra (X-ray), GALEX (ultraviolet), and a score of others. The first edition was written on a 450-Mhz Pentium III with 256 MB of memory. In 1999, the first major digital SLR camera was placed on the market: the Nikon D1. It had only 2.74 million pixels and cost just under $6,000. A typical PC disk drive held 100–200 MB. Webcams existed in 1997, but they were expensive and low-resolution. Persons using computer images needed to have a special image acquisition card and a relatively expensive camera to conduct their work, generally amounting to $1–2,000 worth of equipment. The technology of personal computers and image acquisition has changed a lot since then.

The 1997 first edition was inspired by my numerous scans though the Internet news groups related to image processing and computer vision. I noted that some requests appeared over and over again, sometimes answered and sometimes not, and wondered if it would be possible to answer the more

frequently asked questions in book form, which would allow the development of some of the background necessary for a complete explanation. However, since I had just completed a book (*Practical Computer Vision Using C*), I was in no mood to pursue the issue. I continued to collect information from the Net, hoping to one day collate it into a sensible form. I did that, and the first edition was very well received. (Thanks!)

Fifteen years later, given the changes in technology, I'm surprised at how little has changed in the field of vision and image processing, at least at the accessible level. Yes, the theory has become more sophisticated and three-dimensional vision methods have certainly improved. Some robot vision systems have accomplished rather interesting things, and face recognition has been taken to a new level. However, cheap character recognition is still, well, cheap, and is still not up to a level where it can be used reliably in most cases. Unlike other kinds of software, vision systems are not ubiquitous features of daily life. Why not? Possibly because the vision problem is really a hard one. Perhaps there is room for a revision of the original book?

My goal has changed somewhat. I am now also interested in ''democratization'' of this technology — that is, in allowing it to be used by anyone, at home, in their business, or at schools. Of course, you need to be able to program a computer, but that skill is more common than it was. All the software needed to build the programs in this edition is freely available on the Internet. I have used a free compiler (Microsoft Visual Studio Express), and OpenCV is also a free download. The only impediment to the development of your own image-analysis systems is your own programming ability.

Some of the original material has not changed very much. Edge detection, thinning, thresholding, and morphology have not been hot areas of research, and the chapters in this edition are quite similar to those in the original. The software has been updated to use Intel's OpenCV system, which makes image IO and display much easier for programmers. It is even a simple matter to capture images from a webcam in real time and use them as input to the programs. Chapter 1 contains a discussion of the basics of OpenCV use, and all software in this book uses OpenCV as a basis.

Much of the mathematics in this book is still necessary for the detailed understanding of the algorithms described. Advanced methods in image processing and vision require the motivation and justification that only mathematics can provide. In some cases, I have only scratched the surface, and have left a more detailed study for those willing to follow the references given at the ends of chapters. I have tried to select references that provide a range of approaches, from detailed and complex mathematical analyses to clear and concise exposition. However, in some cases there are very few clear descriptions in the literature, and none that do not require at least a university-level math course. Here I have attempted to describe the situation in an intuitive manner, sacrificing rigor (which can be found almost anywhere else) for as

clear a description as possible. The software that accompanies the descriptions is certainly an alternative to the math, and gives a step-by-step description of the algorithms.

I have deleted some material completely from the first edition. There is no longer a chapter on wavelets, nor is there a chapter on genetic algorithms. On the other hand, there is a new chapter on classifiers, which I think was an obvious omission in the first edition. A key inclusion here is the chapter on the use of parallel programming for solving image-processing problems, including the use of graphics cards (GPUs) to accelerate calculations by factors up to 200. There's also a completely new chapter on content-based searches, which is the use of image information to retrieve other images. It's like saying, ''Find me another image that looks like this.'' Content-based search will be an essential technology over the next two decades. It will enable the effective use of modern large-capacity disk drives; and with the proliferation of inexpensive high-resolution digital cameras, it makes sense that people will be searching through large numbers of big images (huge numbers of pixels) more and more often.

Most of the algorithms discussed in this edition can be found in source code form on the accompanying web page. The chapter on thresholding alone provides 17 programs, each implementing a different thresholding algorithm. Thinning programs, edge detection, and morphology are all now available on the Internet.

The chapter on image restoration is still one of the few sources of practical information on that subject. The symbol recognition chapter has been updated; however, as many methods are commercial, they cannot be described and software can't be provided due to patent and copyright concerns. Still, the basics are there, and have been connected with the material on classifiers.

The chapter on parallel programming for vision is, I think, a unique feature of this book. Again using downloadable tools, this chapter shows how to link all the computers on your network into a large image-processing cluster. Of couse, it also shows how to use all the CPUs on your multi-core and, most importantly, gives an introductory and very practical look at how to program the GPU to do image processing and vision tasks, rather than just graphics.

Finally, I have provided a chapter giving a selection of methods for use in searching through images. These methods have code showing their implementation and, combined with other code in the book, will allow for many hours of experimenting with your own ideas and algorithms for organizing and searching image data sets.

Readers can download all the source code and sample images mentioned in this book from the book's web page — `www.wiley.com/go/jrparker`. You can also link to my own page, through which I will add new code, new images, and perhaps even new written material to supplement and update the printed matter. Comments and mistakes (how likely is that?) can be communicated

through that web page, and errata will be posted, as will reader contributions to the software collection and new ideas for ways to use the code methods for compiling on other systems and with other compilers.

I invite you to make suggestions through the website for subjects for new chapters that you would like to read. It is my intention to select a popular request and to post a new chapter on that subject on the site at a future date. A book, even one primarily released on paper, need not be a completely static thing!

Jim Parker
Cochrane, Alberta, Canada
October 2010

CHAPTER

1

Practical Aspects of a Vision System—Image Display, Input/Output, and Library Calls

When experimenting with vision- and image-analysis systems or implementing one for a practical purpose, a basic software infrastructure is essential. Images consist of pixels, and in a typical image from a digital camera there will be 4–6 million pixels, each representing the color at a point in the image. This large amount of data is stored as a file in a format (such as GIF or JPEG) suitable for manipulation by commercial software packages, such as Photoshop and Paint. Developing new image-analysis software means first being able to read these files into an internal form that allows access to the pixel values. There is nothing exciting about code that does this, and it does not involve any actual image processing, but it is an essential first step. Similarly, image-analysis software will need to display images on the screen and save them in standard formats. It's probably useful to have a facility for image capture available, too. None of these operations modify an image but simply move it about in useful ways.

These bookkeeping tasks can require most of the code involved in an imaging program. The procedure for changing all red pixels to yellow, for example, can contain as few as 10 lines of code; yet, the program needed to read the image, display it, and output of the result may require an additional 2,000 lines of code, or even more.

Of course, this infrastructure code (which can be thought of as an *application programming interface*, or *API*) can be used for all applications; so, once it is developed, the API can be used without change until updates are required. Changes in the operating system, in underlying libraries, or in additional functionalities can require new versions of the API. If properly done, these

new versions will require little or no modification to the vision programs that depend on it. Such an API is the *OpenCV* system.

1.1 OpenCV

OpenCV was originally developed by Intel. At the time of this writing, version 2.0 is current and can be downloaded from `http://sourceforge.net/projects/opencvlibrary/`.

However, Version 2.0 is relatively new, yet it does not install and compile with all of the major systems and compilers. All the examples in this book use Version 1.1 from `http://sourceforge.net/projects/opencvlibrary/files/opencv-win/1.1pre1/OpenCV_1.1pre1a.exe/download`, and compile with the Microsoft Visual C++ 2008 Express Edition, which can be downloaded from `www.microsoft.com/express/Downloads/#2008-Visual-CPP`.

The *Algorithms for Image Processing and Computer Vision* website (`www.wiley.com/go/jrparker`) will maintain current links to new versions of these tools. The website shows how to install both the compiler and OpenCV. The advantage of using this combination of tools is that they are still pretty current, they work, and they are free.

1.2 The Basic OpenCV Code

OpenCV is a library of C functions that implement both infrastructure operations and image-processing and vision functions. Developers can, of course, add their own functions into the mix. Thus, any of the code described here can be invoked from a program that uses the OpenCV paradigm, meaning that the methods of this book are available in addition to those of OpenCV. One simply needs to know how to call the library, and what the basic data structures of open CV are.

OpenCV is a large and complex library. To assist everyone in starting to use it, the following is a basic program that can be modified to do almost anything that anyone would want:

```
// basic.c : A `wrapper´ for basic vision programs.
#include "stdafx.h"
#include "cv.h"
#include "highgui.h"
int main (int argc, char* argv[])
{
    IplImage *image = 0;
```

```
    image = cvLoadImage("C:\AIPCV\image1.jpg", 1 );
    if( image )
    {
        cvNamedWindow( "Input Image", 1 );
        cvShowImage( "Input Image", image );
        printf( "Press a key to exit\n");
        cvWaitKey(0);
        cvDestroyWindow("String");
    }
    else
        fprintf( stderr, "Error reading image\n" );
    return 0;
}
```

This is similar to many example programs on the Internet. It reads in an image (`C:\AIPCV\image1.jpg` is a string giving the path name of the image) and displays it in a window on the screen. When the user presses a key, the program terminates after destroying the display window.

Before anyone can modify this code in a knowledgeable way, the data structures and functions need to be explained.

1.2.1 The IplImage Data Structure

The `IplImage` structure is the in-memory data organization for an image. Images in `IplImage` form can be converted into arrays of pixels, but `IplImage` also contains a lot of structural information about the image data, which can have many forms. For example, an image read from a GIF file could be 256 grey levels with an 8-bit pixel size, or a JPEG file could be read into a 24-bit per pixel color image. Both files can be represented as an `IplImage`.

An `IplImage` is much like other internal image representations in its basic organization. The essential fields are as follows:

`width`	An integer holding the width of the image in pixels
`height`	An integer holding the height of the image in pixels
`imageData`	A pointer to an array of characters, each one an actual pixel or color value

If each pixel is one byte, this is really all we need. However, there are many data types for an image within OpenCV; they can be bytes, ints, floats, or doubles in type, for instance. They can be greys (1 byte) or 3-byte color (RGB), 4 bytes, and so on. Finally, some image formats may have the origin at the upper left (most do, in fact) and some use the lower left (only Microsoft).

Other useful fields to know about include the following:

`nChannels`	An integer specifying the number of colors per pixel (1–4).
`depth`	An integer specifying the number of bits per pixel.
`origin`	The origin of the coordinate system. An integer: 0=upper left, 1=lower left.
`widthStep`	An integer specifying, in bytes, the size of one row of the image.
`imageSize`	An integer specifying, in bytes, the size of the image (= widthStep * height).
`imageDataOrigin`	A pointer to the origin (root, base) of the image.
`roi`	A pointer to a structure that defines a region of interest within this image that is being processed.

When an image is created or read in from a file, an instance of an `IplImage` is created for it, and the appropriate fields are given values. Consider the following definition:

```
IplImage* img = 0;
```

As will be described later in more detail, an image can be read from a file by the following code:

```
img = cvLoadImage(filename);
```

where the variable `filename` is a string holding the name of the image file. If this succeeds, then

```
img->imageData
```

points to the block of memory where the pixels can be found. Figure 1.1 shows a JPEG image named `marchA062.jpg` that can be used as an example.

Reading this image creates a specific type of internal representation common to basic RGB images and will be the most likely variant of the `IplImage` structure to be encountered in real situations. This representation has each pixel represented as three bytes: one for red, one for green, and one for blue. They appear in the order b, g, r, starting at the first row of the image and stepping through columns, and then rows. Thus, the data pointed to by `img->imageData` is stored in the following order:

$b_{0,0}$ $g_{0,0}$ $r_{0,0}$ $b_{0,1}$ $g_{0,1}$ $r_{0,1}$ $b_{0,2}$ $g_{0,2}$ $r_{0,2}$ . . .

This means that the RGB values of the pixels in the first row (row 0) appear in reverse order (b, g, r) for all pixels in that row. Then comes the next row, starting over at column 0, and so on, until the final row.

Figure 1.1: Sample digital image for use in this chapter. It is an image of a tree in Chico, CA, and was acquired using an HP Photosmart M637 camera. This is typical of a modern, medium-quality camera.

How can an individual pixel be accessed? The field `widthStep` is the size of a row, so the start of image row `i` would be found at

```
img->imageData + i*img->widthStep
```

Column `j` is `j` pixels along from this location; if pixels are bytes, then that's

```
img->imageData + i*img->widthStep + j
```

If pixels are RGB values, as in the JPEG image read in above, then each pixel is 3 bytes long and pixel `j` starts at location

```
img->imageData + i*img->widthStep + j*3
```

The value of the field `nChannels` is essentially the number of bytes per pixel, so the pixel location can be generalized as:

```
img->imageData + i*img->widthStep))[j*img->nChannels]
```

Finally, the color components are in the order blue, green, and red. Thus, the blue value for pixel [i,j] is found at

```
(img->imageData + i*img->widthStep)[j*img->nChannels + 0]
```

and green and red at the following, respectively:

```
(img->imageData + i*img->widthStep)[j*img->nChannels + 1]
(img->imageData + i*img->widthStep)[j*img->nChannels + 2]
```

The data type for a pixel will be unsigned character (or `uchar`).

There is a generic way to access pixels in an image that automatically uses what is known about the image and its format and returns or modifies a specified pixel. This is quite handy, because pixels can be bytes, RGB, float, or

double in type. The function `cvGet2D` does this; getting the pixel value at `i,j` for the image above is simply

```
p = cvGet2D (img, i, j);
```

The variable `p` is of type `CvScalar`, which is

```
struct CvScalar
{
   double val[4];
}
```

If the pixel has only a single value (i.e., grey), then p.val[0] is that value. If it is RGB, then the color components of the pixel are as follows:

- Blue is `p.val[0]`
- Green is `p.val[1]`
- Red is `p.val[2]`

Modifying the pixel value is done as follows:

```
p.val[0] = 0;           // Blue
p.val[1] = 255;         // Green
p.val[2] = 255;         // Red
cvSet2D(img,i,j,p);     // Set the (i,j) pixel to yellow
```

This is referred to as *indirect access* in OpenCV documentation and is slower than other means of accessing pixels. It is, on the other hand, clean and clear.

1.2.2 Reading and Writing Images

The basic function for image input has already been seen; `cvLoadImage` reads an image from a file, given a path name to that file. It can read images in JPEG, BMP, PNM, PNG, and TIF formats, and does so automatically, without the need to specify the file type. This is determined from the data on the file itself. Once read, a pointer to an `IplImage` structure is returned that will by default be forced into a 3-channel RGB form, such as has been described previously. So, the call

```
img = cvLoadImage (filename);
```

returns an `IplImage*` value that is an RGB image, unless the file name indicated by the string variable `filename` can't be read, in which case the function returns `0` (null). A second parameter can be used to change the default return image. The call

```
img = cvLoadImage (filename, f);
```

returns a 1 channel (1 byte per pixel) grey-level image if f=0, and returns the actual image type that is found in the file if f<0.

Writing an image to a file can be simple or complex, depending on what the user wants to accomplish. Writing grey-level or RGB color images is simple, using the code:

```
k = cvSaveImage( filename, img );
```

The `filename` is, as usual, a string indicating the name of the file to be saved, and the `img` variable is the image to be written to that file. The file type will correspond to the suffix on the file, so if the `filename` is `file.jpg`, then the file format will be JPEG. If the file cannot be written, then the function returns `0`.

1.2.3 Image Display

If the basic C/C++ compiler is used alone, then displaying an image is quite involved. One of the big advantages in using OpenCV is that it provides easy ways to call functions that open a window and display images within it. This does not require the use of other systems, such as Tcl/Tk or Java, and asks the programmer to have only a basic knowledge of the underlying system for managing windows on their computer.

The user interface functions of OpenCV are collected into a library named `highgui`, and are documented on the Internet and in books. The basics are as follows: a window is created using the `cvNamedWindow` function, which specifies a name for the window. All windows are referred to by their name and not through pointers. When created, the window can be given the `autosize` property or not. Following this, the function `cvShowImage` can be used to display an image (as specified by an `IplImage` pointer) in an existing window. For windows with the `autosize` property, the window will change size to fit the image; otherwise, the image will be scaled to fit the window.

Whenever `cvShowimage` is called, the image passed as a parameter is displayed in the given window. In this way, consecutive parts of the processing of an image can be displayed, and simple animations can be created and displayed. After a window has been created, it can be moved to any position on the screen using `cvMoveWindow (name, x, y)`. It can also be moved using the mouse, just like any other window.

1.2.4 An Example

It is now possible to write a simple OpenCV program that will read, process, and display an image. The input image will be that of Figure 1.1, and the goal will be to threshold it.

First, add the needed `include` files, declare an image, and read it from a file.

```
// Threshold a color image.

#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>


int main (int argc, char* argv[])
{
    IplImage *image = 0;
     int i,j,k;
     int mean=0, count=0;
     char c;

     image = cvLoadImage("C:/AIPCV/marchA062.jpg");
```

At this point, there should be image data pointed to by `image`. If so (if the image is not null), display it in a window, as before.

```
     if( image )
     {
        printf ("Height %d X with %d\n", image->height, image->width);
        cvNamedWindow( "mainWin", CV_WINDOW_AUTOSIZE);
        cvShowImage( "mainWin", image );
        printf ("Display of image is done.\n");
        cvWaitKey(0);          // wait for a key
```

Now perform the thresholding operation. But this is a color image, so convert it to grey first using the average of the three color components.

```
     for (i=0; i<image->height; i++)
       for (j=0; j<image->width; j++)
       {
         k=( (image->imageData+i*image->widthStep)[j*image->nChannels+0]
            +(image->imageData+i*image->widthStep)[j*image->nChannels+1]
            +(image->imageData+i*image->widthStep)[j*image->nChannels+2])/3;
        (image->imageData+i*image->widthStep)[j*image->nChannels+0]
             = (UCHAR) k;
        (image->imageData+i*image->widthStep)[j*image->nChannels+1]
             = (UCHAR) k;
        (image->imageData+i*image->widthStep)[j*image->nChannels+2]
             = (UCHAR) k;
```

At this point in the loop, count and sum the pixel values so that the mean can be determined later.

```
            mean += k;
            count++;
        }
```

Make a new window and display the grey image in it.

```
    cvNamedWindow( "grey", CV_WINDOW_AUTOSIZE);
    cvShowImage( "grey", image );
    cvWaitKey(0);            // wait for a key
```

Finally, compute the mean level for use as a threshold and pass through the image again, setting pixels less than the mean to 0 and those greater to 255;

```
    mean = mean/count;
    for (i=0; i<image->height; i++)
      for (j=0; j<image->width; j++)
      {
        k=(image->imageData+i*image->widthStep)
                              [j * image->nChannels + 0];
        if (k < mean) k = 0;
         else k = 255;

        (image->imageData+i*image->widthStep)[j*image->nChannels+0]
          = (UCHAR) k;
        (image->imageData+i*image->widthStep)[j*image->nChannels+1]
          = (UCHAR) k;
        (image->imageData+i*image->widthStep)[j*image->nChannels+2]
          = (UCHAR) k;
      }
```

One final window is created, and the final thresholded image is displayed and saved.

```
    cvNamedWindow( "thresh");
    cvShowImage( "thresh", image );
    cvSaveImage( "thresholded.jpg", image );
```

Wait for the user to type a key before destroying all the windows and exiting.

```
    cvWaitKey(0);          // wait for a key

    cvDestroyWindow("mainWin");
    cvDestroyWindow("grey");
    cvDestroyWindow("thresh");
}
```

```
    else
      fprintf( stderr, "Error reading image\n" );
    return 0;
}
```

Figure 1.2 shows a screen shot of this program.

Figure 1.2: The three image windows created by the thresholding program.

1.3 Image Capture

The processing of still photos or scientific images can be done quite effectively using scanned image or data from digital cameras. The availability of digital image data has increased many-fold over the past decade, and it is no longer unusual to find a digital camera, a scanner, and a video camera in a typical household or small college laboratory. Other kinds of data and other devices can be quite valuable sources of images for a vision system, key among these the *webcam*. These are digital cameras, almost always USB powered, having image sizes of 640x480 or larger. They acquire color images at video rates, making such cameras ideal for certain vision applications: surveillance,

robotics, games, biometrics, and places where computers are easily available and very high quality is not essential.

There are a great many types of webcam, and the details of how they work are not relevant to this discussion. If a webcam is properly installed, then OpenCV should be able to detect it, and the capture functions should be able to acquire images from it. The scheme used by OpenCV is to first declare and initialize a camera, using a handle created by the system. Assuming that this is successful, images can be captured through the handle.

Initializing a camera uses the `cvCaptureFromCAM` function:

```
CvCapture  *camera = 0;
camera = cvCaptureFromCAM( CV_CAP_ANY );
if( !camera )           error ...
```

The type `CvCapture` is internal, and represents the handle used to capture images. The function `cvCaptureFromCam` initializes capturing a video from a camera, which is specified using the single parameter. `CV_CAP_ANY` will allow any connected camera to be used, but the system will choose which one. If 0 is returned, then no camera was seen, and image capture is not possible; otherwise, the camera's handle is returned and is needed to grab images.

A frame (image) can be captured using the `cvQueryFrame` function:

```
IplImage   *frame = 0;
frame = cvQueryFrame( camera );
```

The image returned is an `IplImage` pointer, which can be used immediately.

When the program is complete, it is always a good idea to free any resources allocated. In this case, that means releasing the camera, as follows:

```
cvReleaseCapture( &camera );
```

It is now possible to write a program that drives the webcam. Let's have the images displayed in a window so that the live video can be seen. When a key is pressed, the program will save the current image in a JPEG file named `VideoFrame`*xx*`.jpg`, where *xx* is a number that increases each time.

```
// Capture.c - image capture from a webcam
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
#include "cv.h"
#include "highgui.h"

int main(int argc, char ** argv)
{
  CvCapture  *camera = 0;
```

```
IplImage   *frame = 0;
int         i, n=0;
char        filename[256];
char c;
```

Initialize the camera and check to make sure that it is working.

```
camera = cvCaptureFromCAM( CV_CAP_ANY );
if( !camera )                          // Get a camera?
{
  fprintf(stderr, "Can't initialize camera\n");
  return -1;
}
```

Open a window for image display.

```
cvNamedWindow("video", CV_WINDOW_AUTOSIZE);
cvMoveWindow ("video", 150, 200);
```

This program will capture 600 frames. At video rates of 30 FPS, this would be 20 seconds, although cameras do vary on this.

```
for(i=0; i<600; i++)
{
  frame = cvQueryFrame( camera );     // Get one frame.
  if( !frame )
  {
    fprintf(stderr, "Capture failed.\n");
  }
```

The following creates a short pause between frames. Without it, the images come in too fast, and in many cases nothing is displayed. `cvWaitKey` waits for a key press or for the time specified — in this case, 100 milliseconds.

```
  c = cvWaitKey(100);
```

Display the image we just captured in the window.

```
// Display the current frame.
   cvShowImage("video", frame);
```

If `cvWaitKey` actually caught a key press, this means that the image is to be saved. If so, the character returned will be >0. Save it as a file in the `AIPCV` directory.

```
   if (c>0)
   {
      sprintf(filename, "C:/AIPCV/VideoFrame%d.jpg", n++);
      if( !cvSaveImage(filename, frame) )
```

```
        {
          fprintf(stderr, "Failed to save frame as '%s'\n", filename);
        } else
          fprintf (stderr, "Saved frame as 'VideoFrame%d.jpg'\n", n-1);
    }
}
```

Free the camera to avoid possible problems later.

```
 cvReleaseCapture( &camera );

// Wait for terminating keypress.
 cvWaitKey(0);
 return 0;
}
```

The data from the camera will be displayed at a rate of 10 frames/second, because the delay between frames (as specified by `cvWaitKey` is 100 milliseconds, or 100/1000 = 0.1 seconds. This means that the frame rate can be altered by changing this parameter, without exceeding the camera's natural maximum. Increasing this parameter decreases the frame rate. An example of how this program appears on the screen while running is given as Figure 1.3.

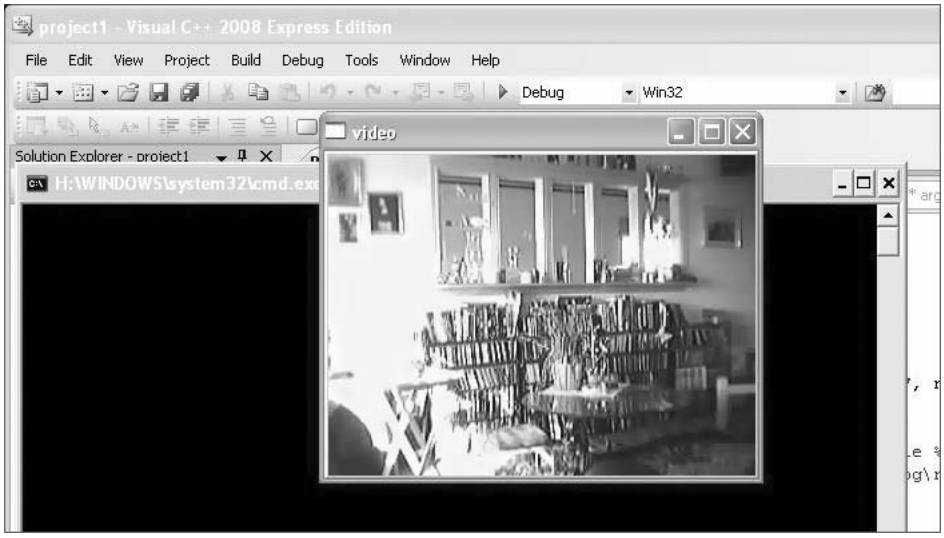


Figure 1.3: How the camera capture program looks on the screen. The image seems static, but it is really live video.

1.4 Interfacing with the AIPCV Library

This book discusses many algorithms, almost all of which are provided in source code form at the book's corresponding website. To access the examples and images on a PC, copy the directory `AIPCV` to the `C:` directory. Within that directory are many C source files that implement the methods discussed here. These programs are intended to be explanatory rather than efficient, and represent another way, a very precise way, to explain an algorithm. These programs comprise a library that uses a specific internal form for storing image data that was intended for use with grey-level images. It is not directly compatible with OpenCV, and so a conversion tool is needed.

OpenCV is not only exceptionally valuable for providing infrastructure to a vision system, but it also provides a variety of image-processing and computer vision functions. Many of these will be discussed in upcoming chapters (Canny and Sobel edge detection, for example), but many of the algorithms described here and provided in code form in the AIPCV library do not come with OpenCV. How can the two systems be used together?

The key detail when using OpenCV is knowledge of how the image structure is implemented. Thus, connecting OpenCV with the AIPCV library is largely a matter of providing a way to convert between the image structures of the two systems. This turns out to be quite simple for grey-level, one-channel images, and more complex for color images.

The basic image structure in the AIPCV library consists of two structures: a header and an image. The image structure, named simply `image`, consists of two pointers: one to a header and one to an array of pixel data:

```
struct image
{
  struct header *info;     // Pointer to header
  unsigned char **data;    // Pointer tp pixels
};
```

The pixel data is stored in the same way as for single-channel byte images in OpenCV: as a block of bytes addressed in row major order. It is set up to be indexed as a 2D array, however, so `data` is an array of pointers to rows. The variable `data[0]` is a pointer to the beginning of the entire array, and so is equivalent to `IplImage.imageData`.

The header is quite simple:

```
struct header
{
      int nr, nc;
      int oi, oj;
};
```

The field `nr` is the number of rows in the image, and `nc` is the number of columns. These are equivalent to `IplImage.height` and `IplImage.width`, respectively. The `oi` and `oj` fields specify the origin of the image, and are used only for a very few cases (e.g., restoration). There are no corresponding fields in OpenCV.

The way to convert an AIPCV image into an OpenCV image is now clear, and is needed so that images can be displayed in windows and saved in JPEG and other formats.

```
IplImage *toOpenCV (IMAGE x)
{
    IplImage *img;
    int i=0, j=0;
    CvScalar s;

    img=cvCreateImage(cvSize(x->info->nc,x->info->nr),8, 1);
    for (i=0; i<x->info->nr; i++)
    {
        for (j=0; j<x->info->nc; j++)
        {
            s.val[0] = x->data[i][j];
            cvSet2D (img, i,j,s);
        }
    }
    return img;
}
```

This function copies the pixel values into a new `IplImage`. It is also possible to use the original data array in the `IplImage` directly. There is some danger in this, in that OpenCV may decide to free the storage, for instance, making both versions inaccessible.

Converting from `IplImage` to AIPCV is more complicated, because OpenCV images might be in color. If so, how is it converted into grey? We'll not dwell on this except to say that one color image can be converted into three monochrome images (one each for red, green, and blue), or a color map could be constructed using a one-byte index that could be used as the pixel value. The solution presented here is to convert a 3-channel color image into grey by averaging the RGB values, leaving the other solutions for future consideration.

```
IMAGE fromOpenCV (IplImage *x)
{
    IMAGE img;
    int color=0, i=0;
    int k=0, j=0;
    CvScalar s;

    if ((x->depth==IPL_DEPTH_8U) &&(x->nChannels==1)) // Grey image
```

```
    img = newimage (x->height, x->width);
    else if ((x->depth==8) && (x->nChannels==3)) //Color
    {
        color = 1;
        img = newimage (x->height, x->width);
    }
    else return 0;

    for (i=0; i<x->height; i++)
    {
        for (j=0; j<x->width; j++)
        {
            s = cvGet2D (x, i, j);
            if (color)
               k= (unsigned char)  ((s.val[0]+s.val[1]+s.val[2])/3);
            else k = (unsigned char)(s.val[0]);
               img->data[i][j] = k;
         }
     }
     return img;
}
```

The two functions `toOpenCV` and `fromOpenCV` do the job of allowing the image-processing routines developed here to be used with OpenCV. As a demonstration, here is the main routine only for a program that thresholds an image using the method of grey-level histograms devised by Otsu and presented in Chapter 4. It is very much like the program for thresholding written earlier in Section 1.2.4, but instead uses the AIPCV library function `thr_glh` to find the threshold and apply it.

```
int main(int argc, char *argv[])
{
  IplImage* img=0;
  IplImage* img2=0;
  IMAGE x;
  int height,width,step,channels;
  uchar *data;
  int mean=0,count=0;

  if(argc<1){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
  }

  // load an image
  img=cvLoadImage("H:/AIPCV/marchA062.jpg");
```

```
  if(!img)
  {
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
  }

  // get the image data
  height    = img->height;
  width     = img->width;
  step      = img->widthStep;
  channels  = img->nChannels;
  data      = (uchar *)img->imageData;
  printf("Processing a %dx%d image with %dchannels\n",
         height,width,channels);

  // create a window
  cvNamedWindow("win1", CV_WINDOW_AUTOSIZE);
  cvMoveWindow("win1", 100, 100);

  // show the image
  cvShowImage("win1", img );

  // Convert to AIPCV IMAGE type
  x = fromOpenCV (img);
  if (x)
  {
    thr_glh (x);
    img2 = toOpenCV (x); // Convert to OpenCV to display
    cvNamedWindow( "thresh");
    cvShowImage( "thresh", img2 );
    cvSaveImage( "thresholded.jpg", img2 );
  }


  // wait for a key
  cvWaitKey(0);

  // release the image
  cvReleaseImage(&img);
  return 0;
}
```

In the remainder of this book, we will assume that OpenCV can be used for image display and I/O and that the native processing functions of OpenCV can be added to what has already been presented.

For convenience, the AIPCV library contains the following X functions for IO and display of its images directly to OpenCV:

`display_image (IMAGE x)`	Displays the specified image on the screen
`save_image (IMAGE x, char *name)`	Saves the image in a file with the given name
`IMAGE get_image (char *name)`	Reads the image in the named file and return a pointer to it
`IMAGE grab_image ()`	Captures an image from an attached webcam and return a pointer to it

1.5 Website Files

The website associated with this book contains code and data associated with each chapter, in addition to new information, errata, and other comments. Readers should create a directory for this information on their PC called `C:\AIPCV`. Within that, directories for each chapter can be named `CH1`, `CH2`, and so on.

The following material created for this chapter will appear in `C:\AIPCV\CH1`:

`capture.c`	Gets an image from a webcam
`lib0.c`	A collection of OpenCV input/output/display functions
`thr_glh.c`	Thresholds an image

1.6 References

Agam, Gady. ''Introduction to Programming With OpenCV,'' `www.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html` (accessed January 27, 2006).

Bradsky, Gary and Kaehler, Adrian. *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol: O'Reilly Media Inc, 2008.

''CV Reference Manual,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cv.htm` (accessed March 16, 2010).

''cvCam Reference Manual,'' `www.cognotics.com/opencv/docs/1.0/cvcam.pdf` (accessed March 16, 2010).

''CXCORE Reference Manual,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cxcore.htm` (accessed March 16, 2010).

''Experimental and Obsolete Functionality Reference,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cvaux.htm` (accessed March 16, 2010).

''HighGUI Reference Manual,'' `cognotics.com/opencv/docs/1.0/ref/opencvref_highgui.htm` (accessed March 16, 2010).

''OpenCV Wiki-Pages,'' `http://opencv.willowgarage.com/wiki`.

Otsu, N, ''A Threshold Selection Method from Grey-Level Histograms,'' *SMC* 9, no. 1 (1979): 62–66.

Parker, J. R. *Practical Computer Vision Using C*. New York: John Wiley & Sons, Inc., 1994.

CHAPTER 2

Edge-Detection Techniques

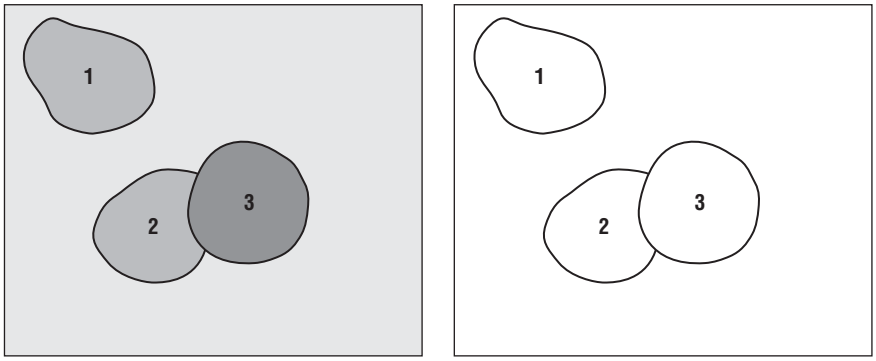
2.1 The Purpose of Edge Detection

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, in the generic sense. Objects are subjects of interest in image analysis and vision systems. 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 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.

Figure 2.1 illustrates a straightforward example of edge detection. There are two overlapping objects in the original picture: (a), which has a uniform grey background; and (b), the edge-enhanced version of the same image 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 2.1 have been isolated, and further processing may determine what kind of object each

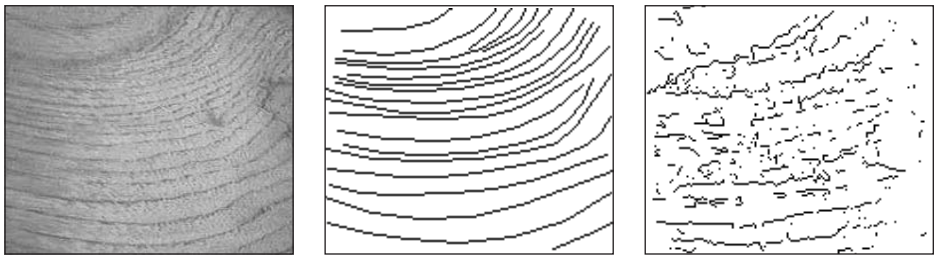
region represents. While in this example edge detection is merely a step in the segmentation process, it is sometimes all that is needed, especially when the objects in an image are lines.



(a) (b)

Figure 2.1: Example of edge detection. (a) Synthetic image with blobs on a grey background. (b) Edge-enhanced image showing only the outlines of the objects.

Consider the image in Figure 2.2, 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 2.2b, is all that is needed to segment the image into foreground (objects = rings) and background (everything else).



(a) (b) (c)

Figure 2.2: The 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.

Technically, *edge detection* is the process of locating the edge pixels, and *edge enhancement* is the process of increasing the contrast between the edges and the background so that the edges become more visible. In practice, however, 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 specific algorithm for edge tracing. The result is a non-raster representation of the objects that can be used to compute shape measures or otherwise identify or classify the object.

The remainder of this chapter discusses the theory of edge detection, including a collection of traditional methods. This includes the Canny edge detector and the Shen-Castan, or ISEF, edge detector. 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.

2.2 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, and then using this definition to suggest methods of enhancement and identification.

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 2.3.

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 (although 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 the edges happen to correspond exactly with a pixel boundary. Indeed, the change in level may extend across some number of pixels (Figures 2.3b–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 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 can be characterized 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.

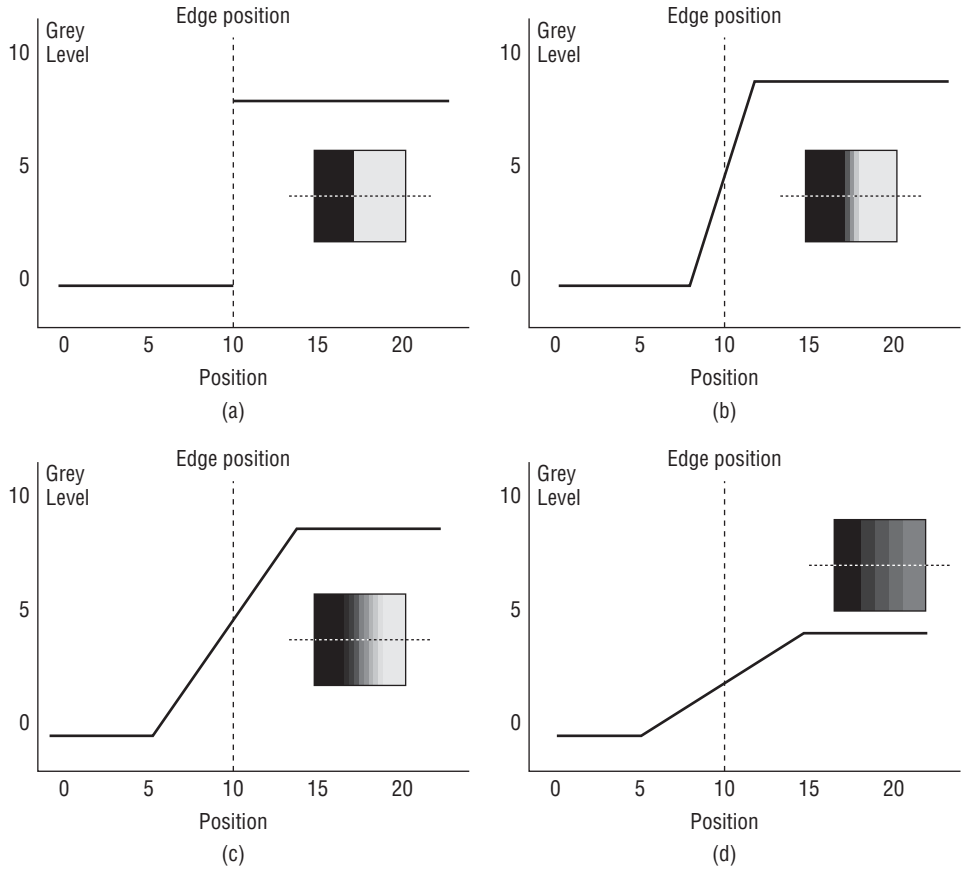


Figure 2.3: Step edges. (a) The change in level occurs exactly at pixel 10. (b) The same level change as before, but over 4 pixels centered at pixel 10. This is a *ramp* edge. (c) Same level change but over 10 pixels, centered at 10. (d) A smaller change over 10 pixels. The insert shows the way the image would appear, and the dotted line shows where the image was sliced to give the illustrated cross-section.

2.2.1 Models of Edges

The step edge of Figure 2.3a 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 2.4a 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 2.4c 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:

$$v = \frac{(v_w a_w + v_b a_b)}{a_w + a_b} \qquad \text{(EQ 2.1)}$$

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, as shown in Figure 2.4d.

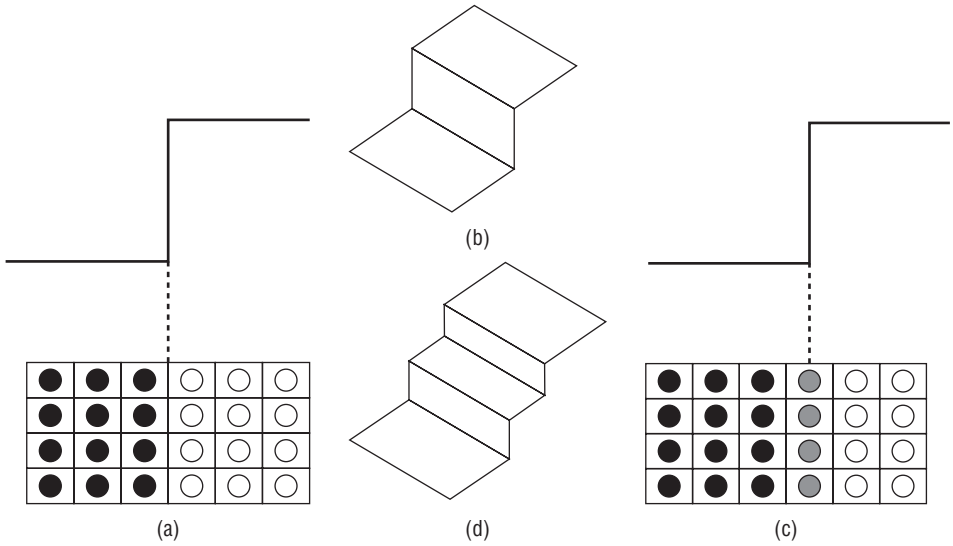


Figure 2.4: The effect of sampling on a step edge. (a) An ideal step edge. (b) Three-dimensional view of the step edge. (c) Step edge sampled at the center of a pixel, instead of on a margin. (d) The result, in three dimensions, has the appearance of a staircase.

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.

2.2.2 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.

Two types of noise are of specific interest in image analysis:

- Signal-independent noise
- Signal-dependent noise

Signal-independent noise is a random set of grey levels, statistically independent of the image data, 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:

$$B = A + N \qquad \text{(EQ 2.2)}$$

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 (Gaussian) 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 2.5 shows an image of a chessboard that has been subjected to various degrees of artificial noise. For a normal distribution with a mean of zero, 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 chessboard. If the noise is consistent over the entire image, 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, illumination being constant, any variation can be assumed to be caused by the noise alone. In this 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).

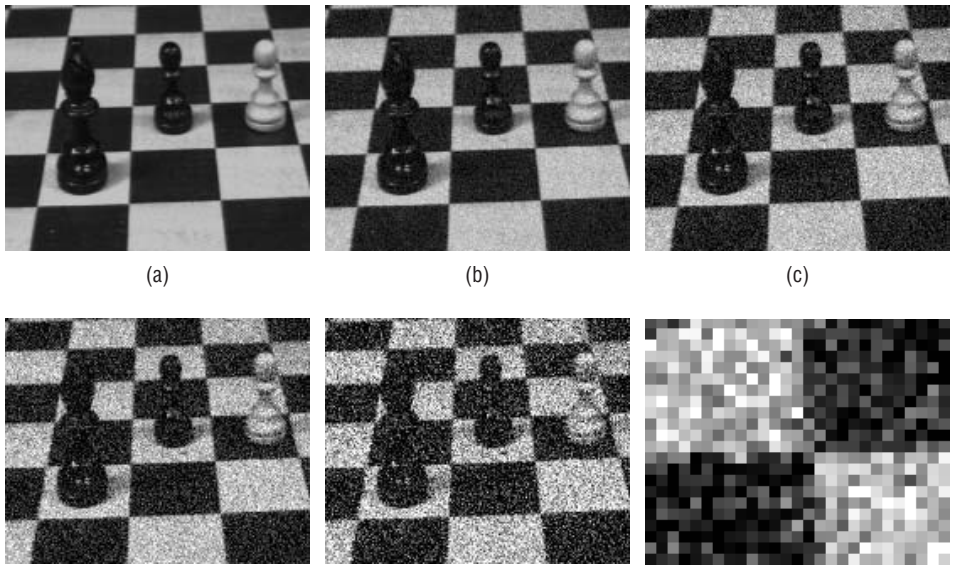


Figure 2.5: Normally distributed noise and its effect on an image. (a) Original image. (b) Noise having s = 10. (c) Noise having s = 20. (d) Noise having s = 30. (e) Noise having s = 50. (f) Expanded view of an intersection of four regions in the s = 50 image.

A program that does this appears in Figure 2.6.

As a simple test, a black square and a white square were isolated from the image in Figure 2.5c 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

```
/* 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 2.6: 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.

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-dependent 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 2.7 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 the random variations, but a good edge detector should be able to determine the edge position even in this situation.

Figure 2.7: (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

Returning, with less confidence, to the case of the ideal step edge, the question of how to identify the location of the edge still remains. An edge, based on the previous discussion, is defined by a grey level (or color) contour. If this contour is crossed, 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.

Both edge pixels and noise pixels are characterized by a significant change in grey level as compared to their surroundings. The fact that edge pixels connect to each other to form a contour allows a distinction to be made between the two.

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 type 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 type, but examples of the first two types will be explored first.

2.2.3 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 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(x,y)` then the gradient is defined as:

$$\nabla A(x,y) = \left(\frac{\partial A}{\partial x}, \frac{\partial A}{\partial y}\right) \qquad \text{(EQ 2.3)}$$

which is a two-dimensional vector.

Of course, an image is not a function and cannot 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 Δ_1:

$$\begin{aligned} \nabla_{x1} A(x,y) &= A(x,y) - A(x-1,y) \\ \nabla_{y1} A(x,y) &= A(x,y) - A(x,y-1) \end{aligned} \qquad \text{(EQ 2.4)}$$

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 *(x,y)*, but at $(x - \frac{1}{2}, y - \frac{1}{2})$. The edge locations would therefore be shifted by one half of a pixel in the $-x$ and $-y$ directions. A better choice for an approximation might be Δ_2:

$$\nabla_{x2}A = A(x+1, y) - A(x-1, y)$$
$$\nabla_{y2}A = A(x, y+1) - A(x, y-1) \qquad \text{(EQ 2.5)}$$

This operator is symmetrical with respect to the pixel *(x,y)*, although it does not consider the value of the pixel at *(x,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 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:

$$G_{mag} = \sqrt{\left(\frac{\partial A}{\partial x}\right)^2 + \left(\frac{\partial A}{\partial y}\right)^2} \qquad \text{(EQ 2.6)}$$

and the direction of the edge is approximately:

$$G_{dir} = \text{atan}\left[\frac{\frac{\partial A}{\partial y}}{\frac{\partial A}{\partial x}}\right] \qquad \text{(EQ 2.7)}$$

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, whereas 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 the methods to be presented, a standard set of test images is suggested. The basic set appears in Figure 2.8, and noisy versions off these will also be used. Noise will be normally distributed and have standard deviations of 3, 9, and 18. For an edge gradient of 18 grey levels, these correspond to signal-to-noise ratios of 6, 2, and 1, respectively. The appearance of the edge-enhanced test images will give a rough cue about how successful the edge-detection algorithm is.

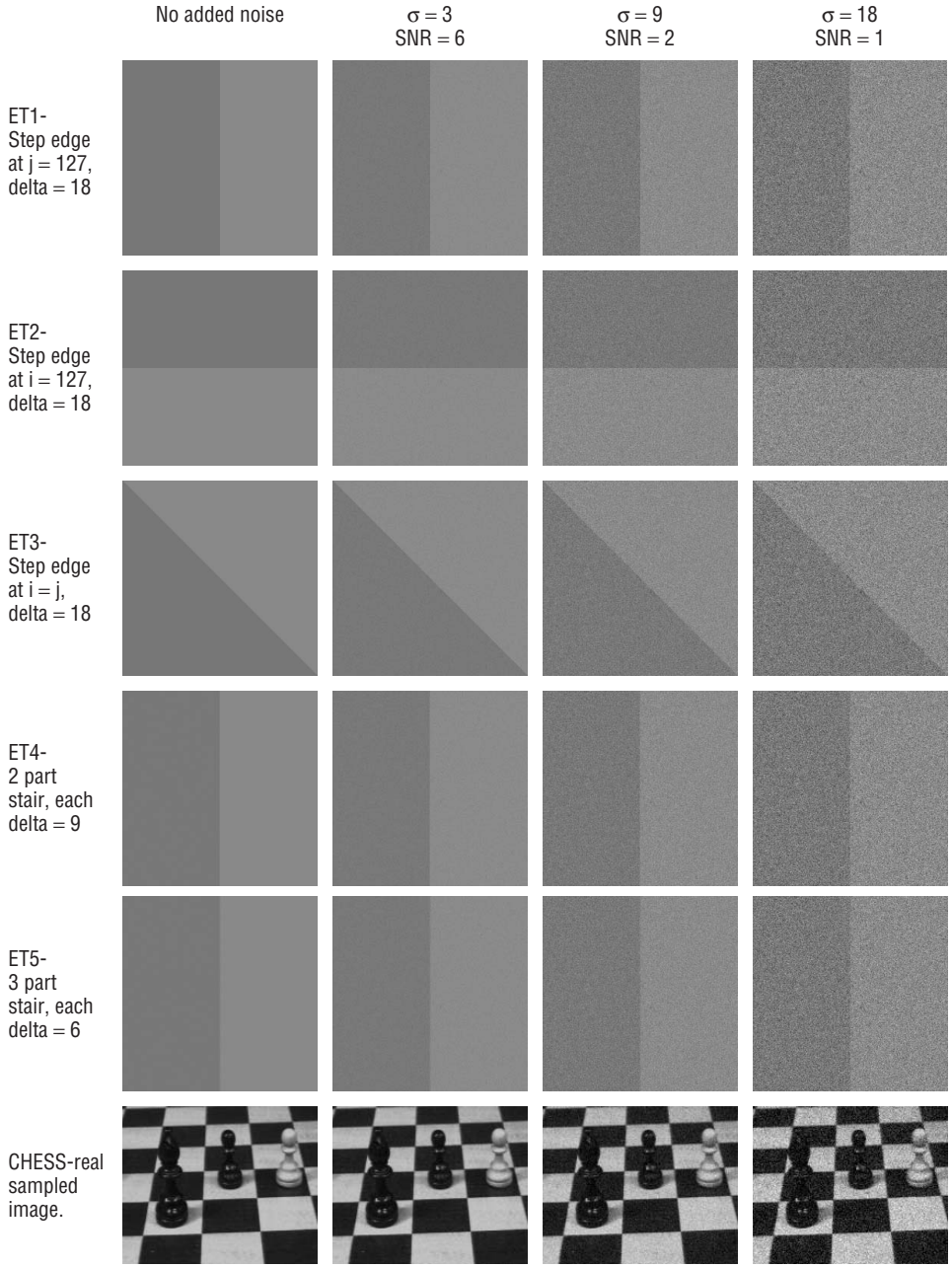


Figure 2.8: Standard test images for edge-detector evaluation. There are three step edges and two stairs, plus a real sampled image; all have been subjected to normally distributed zero mean noise with known standard deviations of 3, 9, and 18.

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, thresholding, or simply poor design, 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 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:

$$E_1 = \frac{\sum_{i=1}^{I_A} \left(\frac{1}{1 + \alpha d(i)^2} \right)}{\max(I_A, I_I)} \qquad \text{(EQ 2.8)}$$

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 a 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 used in Pratt's work. This metric is, as discussed previously, a function of the distance between correct and measured edge positions, but it 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 it 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:

$$L(k) = \begin{cases} a(d, d_k) a\left(\frac{k\pi}{4}, d + \frac{\pi}{2}\right) & \text{if neighbor k is an edge pixel} \\ 0 & \text{Otherwise} \end{cases} \qquad \text{(EQ 2.9)}$$

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:

$$a(\alpha, \beta) = \frac{\pi - |\alpha - \beta|}{\pi} \quad \text{(EQ 2.10)}$$

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

$$R(k) = \begin{cases} a(d, d_k)a\left(\frac{k\pi}{4}, d - \frac{\pi}{2}\right) & \text{if neighbor k is an edge pixel} \\ 0 & \text{Otherwise} \end{cases} \quad \text{(EQ 2.11)}$$

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 3x3 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:

$$E_2 = \gamma^C + (1 - \gamma)T \quad \text{(EQ 2.12)}$$

where γ 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. Table 2.1 gives all the evaluations for Δ_1.

Table 2.1: Evaluation of the Δ_1 operator

IMAGE	EVALUATOR	NO NOISE	SNR = 6	SNR = 2	SNR = 1
ET1	EVAL1	0.965	0.9741	0.0510	0.0402
	EVAL2	1.000	0.6031	0.3503	0.3494
ET2	EVAL1	0.965	0.6714	0.0484	0.3493
	EVAL2	1.000	0.6644	0.3491	0.3493
ET3	EVAL1	0.9726	0.7380	0.0818	0.0564
	EVAL2	0.9325	0.6743	0.3532	0.3493
ET4	EVAL1	0.4947	0.0839	0.0375	0.0354
	EVAL2	0.8992	0.3338	0.3473	0.3489
ET5	EVAL1	0.9772	0.0611	0.0365	0.0354
	EVAL2	0.7328	0.7328	0.3461	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.

Table 2.2 gives the evaluations for Δ_2.

Table 2.2: Evaluation of the Δ_2 operator

IMAGE	EVALUATOR	NO NOISE	SNR = 6	SNR = 2	SNR = 1
ET1	EVAL1	0.9727	0.8743	0.0622	0.0421
	EVAL2	0.8992	0.6931	0.4167	0.4049
ET2	EVAL1	0.9726	0.9454	0.0612	0.0400
	EVAL2	0.8992	0.6696	0.4032	0.4049
ET3	EVAL1	0.9726	0.9707	0.1000	0.0623
	EVAL2	0.9325	0.9099	0.4134	0.4058
ET4	EVAL1	0.5158	0.4243	0.0406	0.0320
	EVAL2	1.0000	0.5937	0.4158	0.4043
ET5	EVAL1	0.5062	0.1963	0.0350	0.0316
	EVAL2	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 the edge-enhanced images, since in this case alone there are 48 of them. Figure 2.9 shows a selection of the results from both operators, and from these images, and from the evaluations, it can be concluded that Δ_2 is slightly superior, especially where the noise is higher.

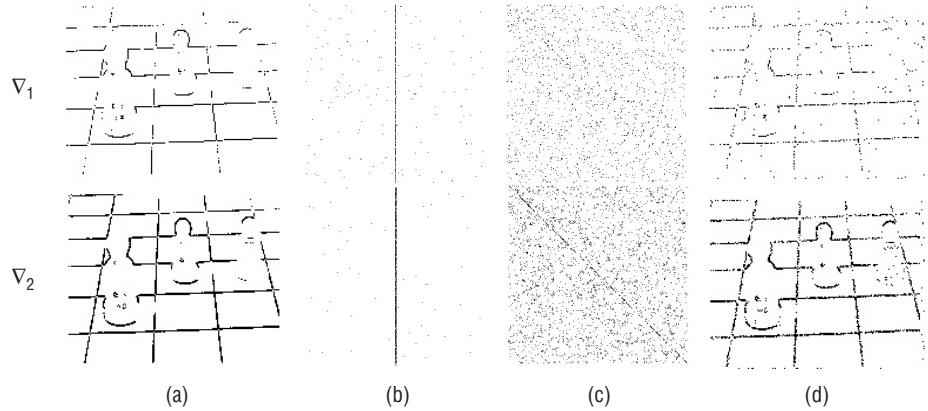


Figure 2.9: Sample results from the gradient edge detectors. (a) Chess image ($\sigma = 3$). (b) ET1 image (SNR = 6). (c) ET3 image (SNR = 2). (d) Chess image ($\sigma = 18$).

2.2.4 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.

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:

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

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 are 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 2.6 and 2.7.

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

$$\begin{aligned} S_x &= I[i-1][j+1] + 2I[i][j+1] + I[i+1][j+1] \\ &\quad - (I[i-1][j-1] + 2I[i][j-1] + I[i+1][j-1]) \\ S_y &= I[i+1][j+1] + 2I[i+1][j] + I[i+1][j-1] \\ &\quad - (I[i-1][j+1] + 2I[i-1][j] + I[i-1][j-1]) \end{aligned}$$

which is equivalent to applying the operator ∇_1 to each 2x2 portion of the 3x3 region, and then averaging the results. 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 2.6, 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 2.4 and 2.5 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 as follows:

$$K0 = \begin{matrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{matrix} \quad K1 = \begin{matrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{matrix} \quad K2 = \begin{matrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{matrix} \quad K3 = \begin{matrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{matrix}$$

$$K4 = \begin{matrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{matrix} \quad K5 = \begin{matrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{matrix} \quad K6 = \begin{matrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{matrix} \quad K7 = \begin{matrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{matrix}$$

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 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 2.8. Table 2.3 shows the results.

Table 2.3: Evaluation of the Sobel edge detector

IMAGE	EVALUATOR	NO NOISE	SNR = 6	SNR = 2	SNR = 1
ET1	EVAL1	0.9727	0.9690	0.1173	0.0617
	EVAL2	0.8992	0.8934	0.4474	0.4263
ET2	EVAL1	0.9726	0.9706	0.1609	0.0526
	EVAL2	0.8992	0.8978	0.4215	0.4255
ET3	EVAL1	0.9726	0.9697	0.1632	0.0733
	EVAL2	0.9325	0.9186	0.4349	0.4240
ET4	EVAL1	0.4860	0.4786	0.0595	0.0373
	EVAL2	0.7328	0.6972	0.4426	0.4266
ET5	EVAL1	0.4627	0.3553	0.0480	0.0355
	EVAL2	0.7496	0.6293	0.4406	0.4250

Table 2.4 shows the results for the Kirsch operator.

Table 2.4: Evaluation of the Kirsch edge detector

IMAGE	EVALUATOR	NO NOISE	SNR = 6	SNR = 2	SNR = 1
ET1	EVAL1	0.9727	0.9727	0.1197	0.0490
	EVAL2	0.8992	0.8992	0.4646	0.4922
ET2	EVAL1	0.9726	0.9726	0.1517	0.0471
	EVAL2	0.8992	0.8992	0.4528	0.4911
ET3	EVAL1	0.9726	0.9715	0.1458	0.0684
	EVAL2	0.9325	0.9200	0.4708	0.4907
ET4	EVAL1	0.4860	0.4732	0.0511	0.0344
	EVAL2	0.7328	0.7145	0.4819	0.4907
ET5	EVAL1	0.4627	0.3559	0.0412	0.0339
	EVAL2	0.7496	0.6315	0.5020	0.4894

Figure 2.10 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 superior to the simple derivative operators, especially as the noise increases.

Note that in all the cases studied so far, unspecified aspects to the edge-detection methods will have an impact on their efficacy. Principal among these is the thresholding method used, but 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.

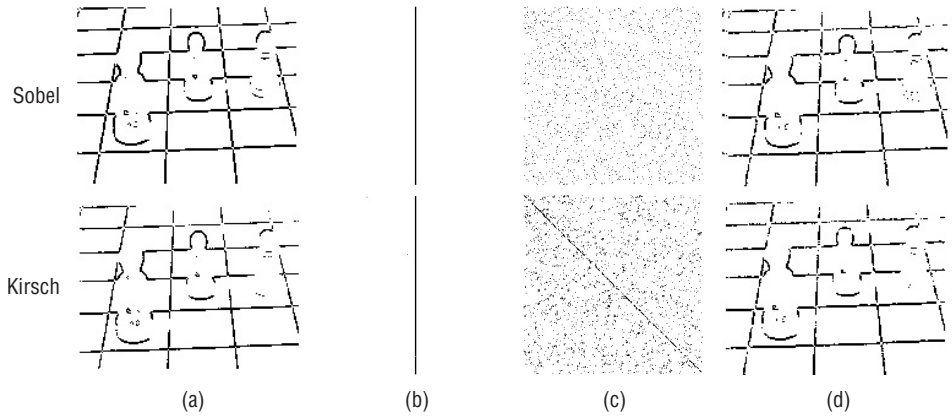


Figure 2.10: Sample results from the template-based edge detectors. (a) Chess image, noise s = 3. (b) ET1, SNR = 6. (c) ET3, SNR = 2. (d) Chess image, noise s = 18.

2.3 Edge Models: The Marr-Hildreth Edge Detector

In the late 1970s, 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*’’ [Marr 1980]. He called the lowest level description 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 preceding five points, 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. Find the edge pixels — 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:

$$I * G(i,j) = \sum_n \sum_m I(n,m)G(i-n, j-m) \qquad \text{(EQ 2.13)}$$

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

$$G_\sigma(x,y) = \sigma^2 e^{\frac{-(x^2+y^2)}{\sigma^2}} \qquad \text{(EQ 2.14)}$$

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

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \qquad \text{(EQ 2.15)}$$

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

$$\nabla^2 G_\sigma = \left(\frac{r^2 - 2\sigma^2}{\sigma^4}\right) e^{\left(\frac{-r}{2\sigma^2}\right)} \qquad \text{(EQ 2.16)}$$

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 LoG (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 LoG 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 `marr.c` reads the image input file name from the standard input, and also accepts a standard deviation value s as a parameter. 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 result is displayed and is stored in a file named `marr.jpg`.

Figure 2.11 illustrates the steps in this process, using the chess image (no noise) as an example. Figures 2.11a and b shows the original image after being convolved with the LoGs, having σ values of 1.2 and 2.8, respectively. Figures 2.11c and 2.11d are the responses from these two different values of σ, and Figure 2.11e shows the result of merging the edge pixels in these two images.

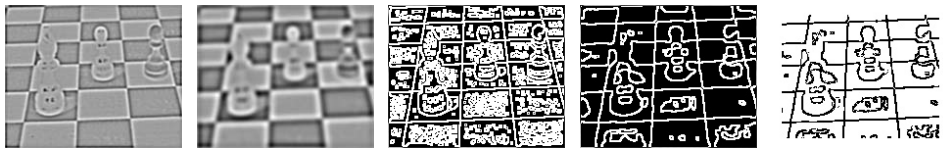


Figure 2.11: Steps in the computation of the Marr-Hildreth edge detector. (a) Convolution of the original image with the LoG having $\sigma = 1.2$. (b) Convolution of the image with the LoG having $\sigma = 2.8$. (c) Zero crossings found in (a). (d) Zero crossings found in (b). (e) Result, found by using zero crossings common to both.

Figure 2.12 shows the result of the Marr-Hildreth edge detector applied to the all the test images of Figure 2.8.

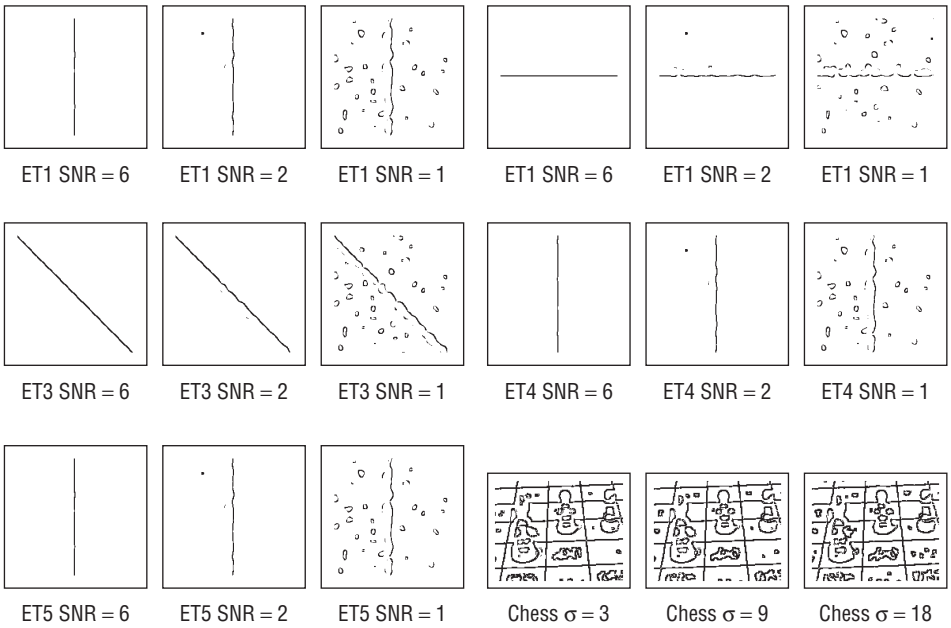


Figure 2.12: Edges from the test images as found by the Marr-Hildreth algorithm, using two resolution values.

In addition, the evaluation of this operator is given in Table 2.5.

Table 2.5: Evaluation of the Marr-Hildreth edge detector

IMAGE	EVALUATOR	NO NOISE	SNR = 6	SNR = 2	SNR = 1
ET1	EVAL1	0.8968	0.7140	0.7154	0.2195
	EVAL2	0.9966	0.7832	0.6988	0.7140
ET2	EVAL1	0.6948	0.6948	0.6404	0.1956
	EVAL2	0.9966	0.7801	0.7013	0.7121
ET3	EVAL1	0.7362	0.7319	0.7315	0.2671
	EVAL2	0.9133	0.7766	0.7052	0.7128
ET4	EVAL1	0.4194	0.4117	0.3818	0.1301
	EVAL2	0.8961	0.7703	0.6981	0.7141
ET5	EVAL1	0.3694	0.3822	0.3890	0.1290
	EVAL2	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 4σ from the boundary of the image are not processed; 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, on the other hand, are 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.

2.4 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:

- **Error rate** — The edge detector should respond only to edges, and should find all of them; no edges should be missed.
- **Localization** — The distance between the edge pixels as found by the edge detector and the actual edge should be as small as possible.
- **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 glace.

Canny assumed a step edge subject to white Gaussian noise. The edge detector was assumed to be a convolution filter 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 f to an edge G is given by a convolution integral:

$$H = \int_{-W}^{W} G(-x)f(x)dx \qquad \text{(EQ 2.17)}$$

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

$$SNR = \frac{A\left|\int_{-W}^{0} f(x)dx\right|}{n_0\sqrt{\int_{-W}^{w} f^2(x)dx}} \qquad \text{(EQ 2.18)}$$

$$Localization = \frac{A|f(0)|}{n_0\sqrt{\int_{-W}^{W} f^2dx}} \qquad \text{(EQ 2.19)}$$

$$x_{zc} = \pi\left(\frac{\int_{-\infty}^{\infty} f^2(x)dx}{\int_{-\infty}^{\infty} f^2(x)dx}\right)^{\frac{1}{2}} \qquad \text{(EQ 2.20)}$$

The value of *SNR* is the output signal-to-noise ratio (error rate), and should be as large as possible: we need a lot 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 x_{zc} is a constraint; it represents the mean distance between zero crossings of f, and is essentially a statement that the edge detector f will not have too many responses to the same edge in a small region.

Canny attempts to find the filter f that maximizes the product SNR $*$ localization subject to the multiple response constraint. Although 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:

$$G(x) = e^{-\frac{x^2}{2\sigma^2}} \qquad \text{(EQ 2.21)}$$

The derivative with respect to x is therefore

$$G'(x) = \left(-\frac{x}{\sigma^2}\right) e^{-\left(\frac{x^2}{2\sigma^2}\right)} \quad \text{(EQ 2.22)}$$

In two dimensions, a Gaussian is given by

$$G(x,y) = \sigma^2 e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \quad \text{(EQ 2.23)}$$

and G has derivatives in both the x and y directions. The approximation to Canny's optimal filter for edge detection is G', and so by convolving the input image with G', we obtain an image 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 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, I.
2. Create a one-dimensional Gaussian mask G to convolve with I. The standard deviation(s) of this Gaussian is a parameter to the edge detector.
3. Create a one-dimensional mask for the first derivative of the Gaussian in the x and y directions; call these G_x and G_y. The same s value is used as in step 2.
4. Convolve the image I with G along the rows to give the x component image I_x, and down the columns to give the y component image I_y.
5. Convolve I_x with G_x to give I_x', the x component of I convolved with the derivative of the Gaussian, and convolve I_y with G_y to give I_y'.
6. Compute the magnitude of the edge response (i.e., if you want to view the result at this point) by combining the x and y components. The magnitude of the result can be 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 `s`. It then calls the function `canny`, which does most of the actual work. First, `canny` computes 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 `s`; for small `s` 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 `dxy_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 2.13 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 though 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.

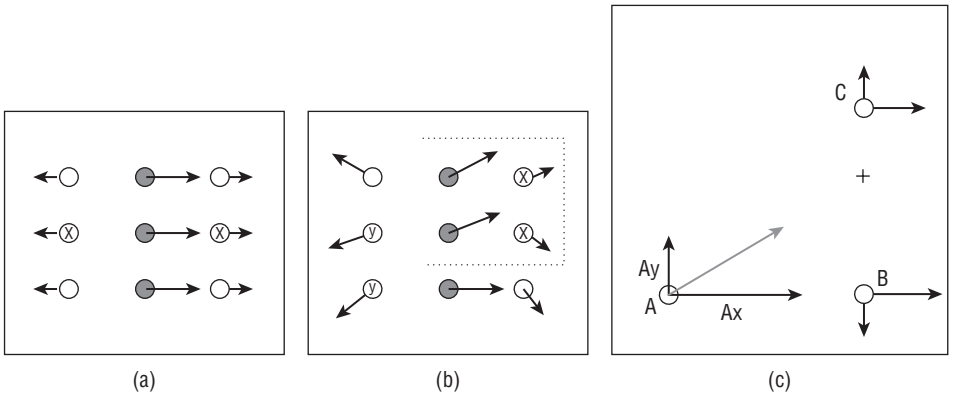


Figure 2.13: Non-maximum suppression. (a) Simple case, where the gradient direction is horizontal. (b) Most cases have gradient directions that are not horizontal or vertical, so there is no exact gradient at the desired point. (c) Gradients at pixels neighboring A are used to estimate the gradient at the location marked with "+."

A more general case is shown in Figure 2.13b. 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 2.13c, since the other case is really the same. The pixel named A is the one 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 2.13 the crossing point is marked with a ''+'', and is in between B and C. The gradient magnitude at this point is estimated as

$$G = (P_y - C_y)Norm(C) + (B_y - P_y)Norm(B) \quad \text{(EQ 2.24)}$$

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 2.14 shows the various stages in processing the chessboard test image of Figure 2.8 (no added noise).

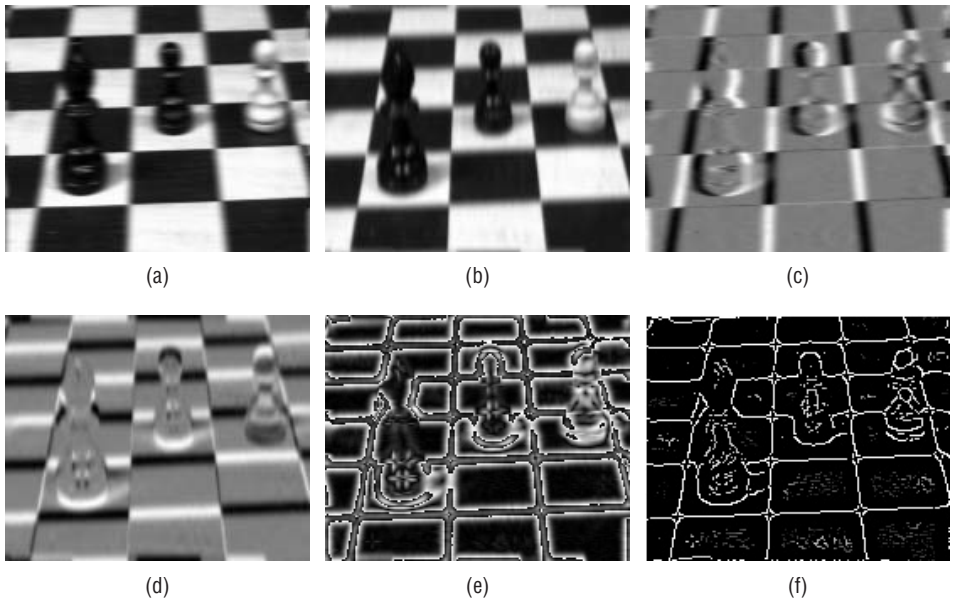


Figure 2.14: Intermediate results from the Canny edge detector. (a) *X* component of the convolution with a Gaussian. (b) *Y* component of the convolution with a Gaussian. (c) *X* component of the image convolved with the derivative of a Gaussian. (d) *Y* component of the image convolved with the derivative of a Gaussian. (e) Resulting magnitude image. (f) After non-maximum suppression.

The stages are: computing the result of convolving with a Gaussian in the x and y directions (Figures 2.14a and b); computing the derivatives in the x and

y directions (Figure 2.14c and d); computing the magnitude of the gradient before non-maximal suppression (Figure 2.14e) and again after non-maximal suppression (Figure 2.14f). 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_l 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 2.15 shows the result of adding hysteresis thresholding after non-maximum suppression. 2.15a is an expanded piece of Figure 2.14f, 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 (2.15b) and a high threshold (2.15c) have been globally applied to the magnitude image, and the result of hysteresis thresholding is given in Figure 2.15d.

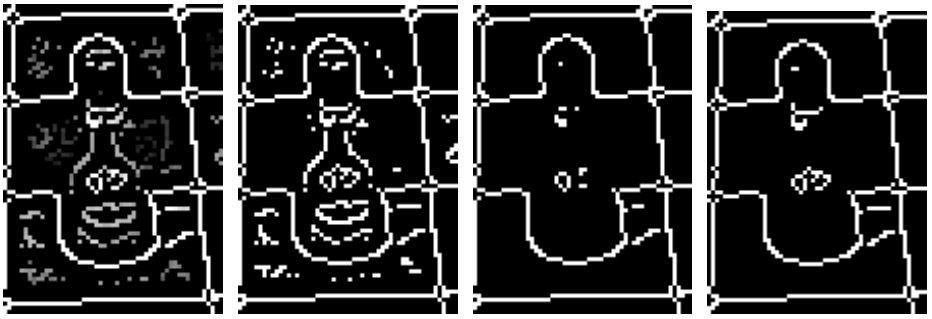


Figure 2.15: Hysteresis thresholding. (a) Enlarged portion of Figure 2.14f. (b) This portion after thresholding with a single low threshold. (c) After thresholding with a single high threshold. (d) After hysteresis thresholding.

Examples of results from this edge detector will be seen in Section 2.6.

2.5 The Shen-Castan (ISEF) Edge Detector

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

reason to think they are the only ones possible. 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 though 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):

$$C_N^2 = \frac{4 \int_0^\infty f^2(x)dx \cdot \int_0^\infty f'^2(x)dx}{f^4(0)} \qquad \text{(EQ 2.25)}$$

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):

$$f(x) = \frac{p}{2} e^{-p|x|} \qquad \text{(EQ 2.26)}$$

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 it 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:

$$f(x, y) = a \cdot e^{-p(|x|+|y|)} \qquad \text{(EQ 2.27)}$$

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*. Although 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:

$$f[i, j] = \frac{(1 - b)b^{|x|+|y|}}{1 + b} \qquad \text{(EQ 2.28)}$$

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]`:

$$y_1[i,j] = \frac{1-b}{1+b} I[i,j] + b y_1[i,j-1], j = 1 \ldots N, i = 1 \ldots M$$
$$y_2[i,j] = b\frac{1-b}{1+b} I[i,j] + b y_1[i,j+1], j = N \ldots 1, i = 1 \ldots M$$
$$r[i,j] = y_1[i,j] + y_2[i,j+1] \quad \text{(EQ 2.29)}$$

with the boundary conditions:

$$I[i,0] = 0$$
$$y_1[i,0] = 0$$
$$y_2[i,M+1] = 0 \quad \text{(EQ 2.30)}$$

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

$$y_1[i,j] = \frac{1-b}{1+b} I[i,j] + b y_1[i-1,j], i = 1 \ldots M, j = 1 \ldots N$$
$$y_2[i,j] = b\frac{1-b}{1+b} I[i,j] + b y_1[i+1,j], i = N \ldots 1, j = 1 \ldots N$$
$$y[i,j] = y_1[i,j] + y_2[i+1,j] \quad \text{(EQ 2.31)}$$

with the boundary conditions:

$$I[0,j] = 0$$
$$y_1[0,j] = 0$$
$$y_2[N+1,j] = 0 \quad \text{(EQ 2.32)}$$

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 2.29) and `ISEF_horiz` to filter the columns (Equation 2.31). The value of b is a parameter to the filter, and is specified by the user.

All 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:

$$S[i,j] - I[i,j] \approx \frac{1}{4a^2} I[i,j] * \nabla^2 f(i,j) \quad \text{(EQ 2.33)}$$

The resulting image $B = S\text{-}I$ is the *band-limited Laplacian* of the image. From this the *binary Laplacian image* (BLI) is obtained by setting all the positive valued pixels in 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 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.

2.6 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, and then performs non-maximum suppression and hysteresis thresholding. The Shen-Castan algorithm convolves the image with the Infinite Symmetric Exponential Filter, computes the BLI,

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:

Canny	Shen-Castan (ISEF)
Sigma (standard deviation)	0<=b<=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 2.16 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.

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 becomes greater, at least for the E1 metric, as shown in Table 2.6.

Canny has the advantage using the E2 metric, as shown in Table 2.7.

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, Δ_2 and Δ_1 in that order. The comparison 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.

Table 2.6: Evaluation of Canny VS ISEF: E1

IMAGE	EVALUATOR	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 ISEF	0.9650	0.9155	0.6991	0.2530
	Canny ISEF	0.9650	0.9338	0.8269	0.7170
ET3	Canny ISEF	0.8776	0.9015	0.7347	0.5238
	Canny ISEF	0.8776	0.9015	0.7347	0.5238
ET4	Canny ISEF	0.5157	0.5092	0.3201	0.1103
	Canny ISEF	0.4686	0.4787	0.4599	0.4227
ET5	Canny ISEF	0.5024	0.4738	0.3008	0.0955
	Canny ISEF	0.4957	0.4831	0.4671	0.4074

Table 2.7: Evaluation Canny VS ISEF: E2

IMAGE	EVALUATOR	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 ISEF	1.0000	0.6039	0.5518	0.5726
	Canny ISEF	1.0000	0.9462	0.6018	0.5209
ET3	Canny ISEF	0.9291	0.7541	0.6032	0.5899
	Canny 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 ISEF	1.0000	0.5319	0.5269	0.5706
	Canny ISEF	0.9900	0.6162	0.5243	0.5123

2.7 Color Edges

So far, only edges created by a change in brightness, as indicated by grey level value, have been examined. This involves probably 90% of edges of interest in real problems, but not all of them. It turns out that changes in color, or *hue*, are not always detected by the edge detectors described so far. If edges are the boundaries of objects, then boundaries that are marked by color alone should be detectable, and because most images involve color it is important to consider it when looking for edges.

There are two main ways that color edges are located. One method is to apply one of the edge detectors already discussed to each of the color channels — red, green, and blue — and then to merge the three results into a single result. The other method involves multi-dimensional gradients, or partial derivatives. The former scheme is pretty obvious and easy to implement. The code already exists to apply any of the methods already discussed to any grey level image, and a red, green, or blue component image is effectively grey, consisting as it does of 8-bit pixels.

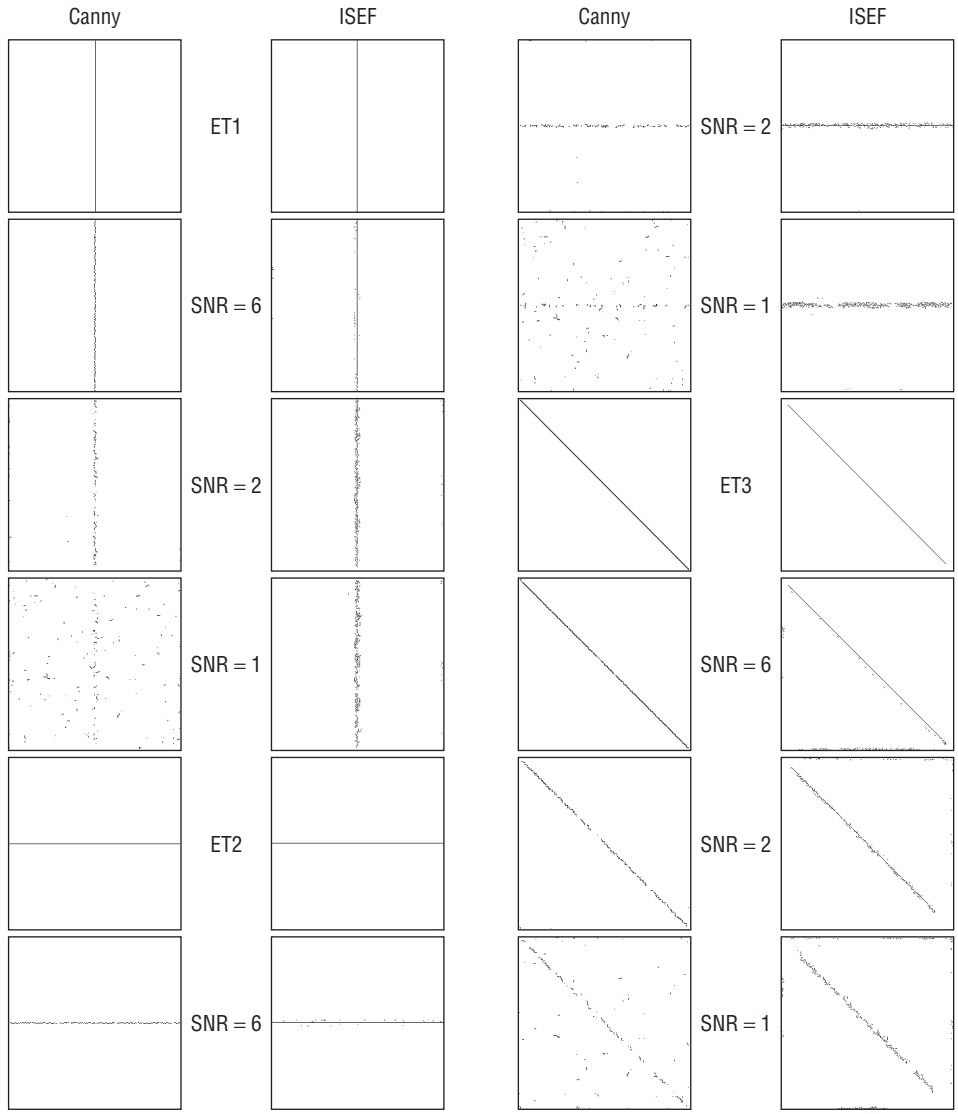


Figure 2.16: Side-by-side comparison of the output of the Canny and Shen-Castan (ISEF) edge detectors. All of the test images from figure 2.8 have been processed by both algorithms, and the output appears here and on the next page.

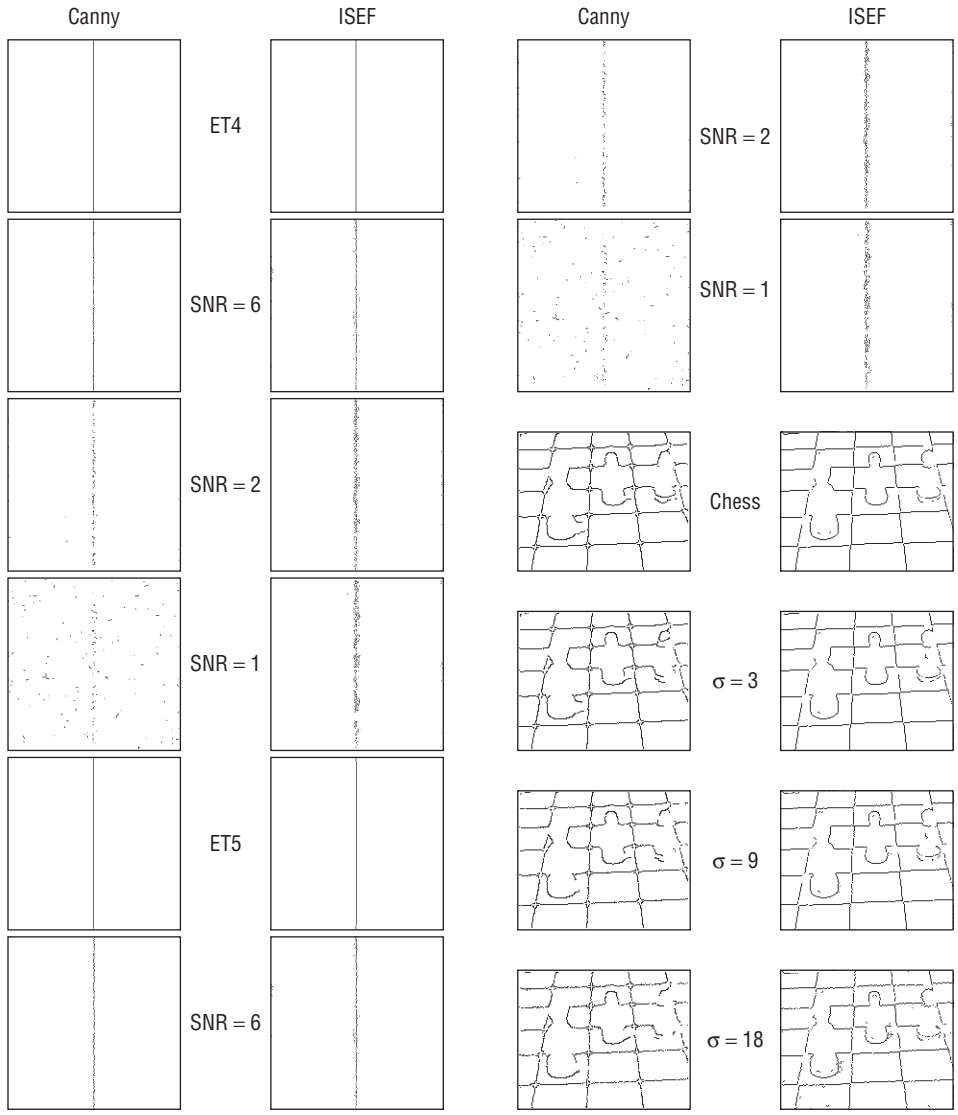


Figure 2.16: (*Continued*)

The following is the basic code for applying the Sobel algorithm to each color channel:

```
if(get_RGB(&x, &y, &z, image_name))
{
    sobel (x);
    sobel (y);
    sobel (z);
    for (i=0; i<x->info->nr; i++)
      for (j=0; j<x->info->nc; j++)
        x->data[i][j] =  255 - (x->data[i][j]+y->data[i][j] +
                          z->data[i][j])/3;
    save_iamge (x, out_name);
}
```

This precise scheme gives the image results shown in Figure 2.17.

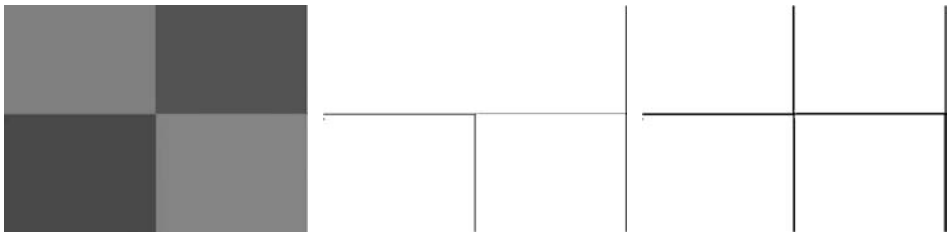


Figure 2.17: (a) The original color image to be processed. (b) The result of converting the image to grey and finding images. (c) The result of applying the Sobel operator to each color channel and then merging the results.

The target image consists of four distinctly colored squares (Figure 2.17a). If the image is converted into grey levels, or pure intensity, the Sobel edges detected are those seen in Figure 2.17b; two are clearly missing. Figure 2.17c is the results of thresholding the image created by averaging the values found but applying the Sobel operator to each of the color components.

A reason that RGB values are not as good as some other color coding schemes for many vision tasks is that they include a significant proportion of intensity. Each color component is the intensity of that color within the whole pixel, and intensity is what is recognized by other edge detectors. A more pure representation of color would be desirable for finding color edges using a differential operator.

In one of the first publications on the subject, Nevatia [1977] suggests using new values T_1 and T_2 instead of RGB:

$$T_1 = \frac{R}{R+G+B}$$
$$T_2 = \frac{G}{R+G+B} \qquad \text{(EQ 2.34)}$$

T_1 and T_2 are variables only involving color. Any local change in T_1 or T_2 can be indicative of a color edge. Results of applying a Sobel operator to T_1 and T_2 can be seen in Figure 2.18.

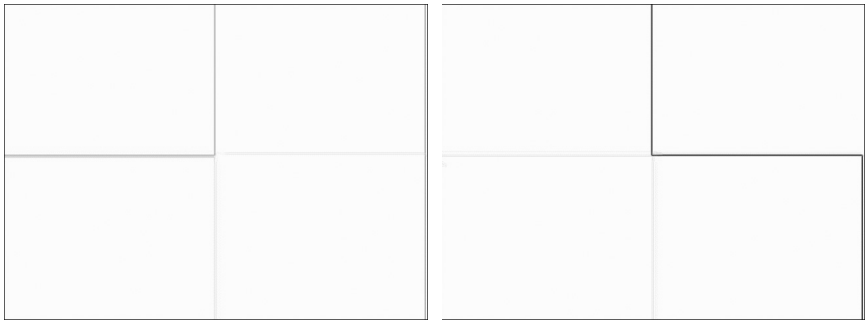


Figure 2.18: Sobel edges found using (a) *T1* color metric and (b) *T2* color metric.

A more traditional (and difficult) calculation is to find the *hue*, the value of the color portion of a pixel. Hue is the ''H'' part of the HSV color system, which can be found described in dozens of places. The basic idea is that a color pixel consists of a *value* (*V*), which is its intensity, a *saturation* (*S*), which is the amount of color, and the *hue* (*H*), which is the nature of the color.

As shown in Figure 2.19a, the value is height along a vertical axis, and saturation is a distance along a radius of the color cone shown.

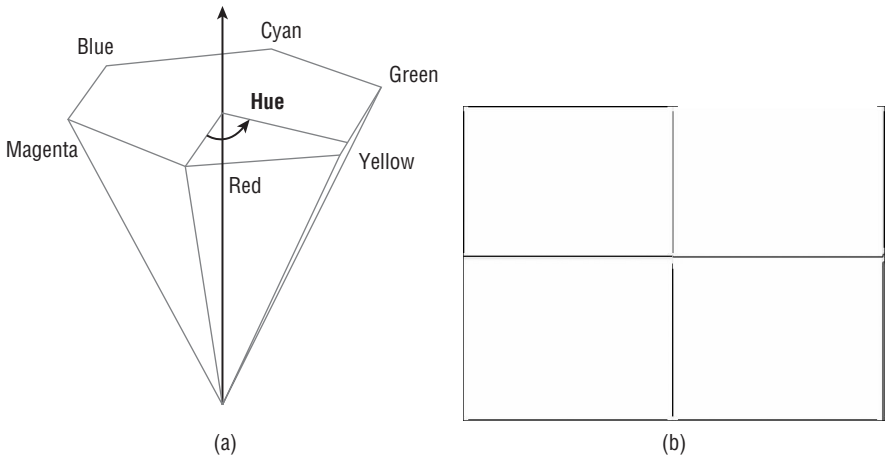


Figure 2.19: (a) The HSV color space is a cone, and the hue value is actually an angle. (b) The result of applying the Sobel operator to the hue component of the pixels of Figure 2.17a.

The hue is actually an angle from 0 (red) to the color being specified. This makes hue a pure color coordinate, and could be used to detect edges. Figure 2.19b is the simple color image of 2.17a converted into hue and then run through the Sobel edge detector. The edges have been thresholded, partly because the horizontal ones correspond to transitions between similar colors, and so are weaker than the vertical edges.

2.8 Source Code for the Marr-Hildreth Edge Detector

```
/* Marr/Hildreth edge detection */

#include "stdio.h"
#include "cv.h"
#include "highgui.h"
#include <math.h>
#include "lib.h"

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;
}
```

```
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];
          }
}

int main ()
{
        int i,j,n;
        float s=1.0;
        FILE *params;
        IMAGE im1, im2;
        char name[128];

        // Try to read an image
        printf ("Enter path to the image file to be processed: ");
        scanf ("%s", name);
        printf ("Opening file '%s'\n", name);
        im1 = get_image(name);
        printf ("Enter standard deviation: ");
        scanf ("%f", &s);

        display_image (im1);

/* Look for parameter file */
        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;

        display_image (im1);
        save_image (im1, "marr.jpg");

        return 0;
}
```

2.9 Source Code for the Canny Edge Detector

```
/* Canny edge detection */
#include "stdio.h"
#include "cv.h"
#include "highgui.h"

/* Scale floating point magnitudes and angles to 8 bits */
#define ORI_SCALE 40.0
#define MAG_SCALE 20.0
#define PI 3.1415926535

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

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

int range (IMAGE x, int i, int j)
{
       if ( (i>=0) && (i<x->info->nr) && (j>=0) && (j<x->info->nc) )
              return 1;
       else return 0;
}

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,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)
{
    float xx;

    if (sigma == 0) return 0.0;
    xx = (float)exp((double) ((-x*x)/(2*sigma*sigma)));
    return xx;
```

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

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

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

/*  HYSTERESIS thersholding 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 threhsold.       */

void hysteresis (int high, int low, IMAGE im, IMAGE mag, IMAGE oriim)
{
       int i,j;

       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;
         }
}

void nonmax_suppress (float **dx, float **dy, int nr, int nc,
              IMAGE mag, IMAGE ori)
{
       int i,j;
       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] = (unsigned char) (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)/2.0f;
}
```

```
int main ()
{
       int i,j;
       float s=1.0;
       int low= 0,high=-1;
       FILE *params;
       IMAGE im, magim, oriim;
       char name[128];

       // Try to read an image
       printf ("Enter path to the image file to be processed: ");
       scanf ("%s", name);
       printf ("Opening file '%s'\n", name);

/* 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, "%f", &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");

       im = get_image(name);
       display_image (im);

       /* 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);

/* 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;



        display_image (im);
        save_image (im, "canny.jpg");

        return 0;
}
```

2.10 Source Code for the Shen-Castan Edge Detector

```
/* ISEF edge detector */
#include "stdio.h"
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <string.h>
#include <math.h>

#define OUTLINE 25

/* 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 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 direcion (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)
  {
/* test for zero-crossings of laplacian then make sure that zero-crossing */
  /* sign correspondence principle is satisfied.  i.e. a positive z-c must */
  /* have a positive 1st derivative where positive z-c means the 2nd deriv */
  /* goes from positive to negative as we pass through the step 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;
}

int main ()
{
        IMAGE im, res;
        FILE *params;
        char name[128];

        // Try to read an image
        printf ("Enter path to the image file to be processed: ");
        scanf ("%s", name);
        printf ("Opening file '%s'\n", name);
```

```
        im = get_image(name);
        display_image (im);

/* 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);

        display_image (res);
        save_image (res, "shen.jpg");

        return 0;
}
```

2.11 Website Files

canny.c	Canny edge detector
colorEdge1.c	Color edge detector

colorEdge2.c	Color edge detector
colorEdge3.c	Color edge detector
eval1.c	Pratt edge evaluation
eval2.c	Rosenfeld edge evaluation
gnoise.c	Adds Gaussian noise to an image
grad1.c	Simple gradient edge detector
grad2.c	Simple gradient edge detector
kirsch.c	Kirsch edge detector
lib.c	Basic code library
lib.h	Library include file
maketmpl.c	Builds edge evaluation images
marr.c	Marr-Hildreth edge detector
measure.c	Measure noise in an image
shen.c	ISEF edge detector
sobel.c	Sobel edge detector
canny.par	Parameter file for Canny edge detector
shen.par	Parameter file for ISEF edge detector
chess.jpg	Test image, chessboard; JPEG version
chess.pgm	Test image, chessboard; PGM version
chess_18.pgm	Chess image, with noise mean = 18
chess_3.pgm	Chess image, with noise mean = 3
chess_9.pgm	Chess image, with noise mean = 9
et1.pgm	Edge test image, vertical boundary
et1_18.pgm	ET1 with noise mean = 18
et1_3.pgm	ET1 with noise mean = 3
et1_9.pgm	ET1 with noise mean = 9
et2.pgm	Edge test image, horizontal boundary
et2_18.pgm	ET2 with noise mean = 18
et2_3.pgm	ET2 with noise mean = 3

et2_9.pgm	ET2 with noise mean = 9
et3.pgm	Edge test image, upper-left to lower-right boundary
et3_18.pgm	ET3 with noise mean = 18
et3_3.pgm	ET3 with noise mean = 3
et3_9.pgm	ET4 with noise mean = 9
et4.pgm	Edge test image, EF1.line at boundary
et4_18.pgm	ET4 with noise mean = 18
et4_3.pgm	ET4 with noise mean = 3
et4_9.pgm	ET4 with noise mean = 9
et5.pgm	Edge test image, ET1 2-pixel line at boundary
et5_18.pgm	ET5 with noise mean = 18
et5_3.pgm	ET5 with noise mean = 3
et5_9.pgm	ET5 with noise mean = 9
n20b.pgm	Noise in a black region
n20w.pgm	Noise in a white region
wood.pgm	Teak wood grain image (2.2a)
et1.edg	Ground truth for ET1.PGM
et2.edg	Ground truth for ET2.PGM
et3.edg	Ground truth for ET3.PGM
et4.edg	Ground truth for ET4.PGM
et5.edg	Ground truth for ET5.PGM

2.12 References

Abdou, I. E. and Pratt, W. K. ''Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors.'' *Proceedings of the IEEE* 67 (1979): 753–763.

Baker, S. and Nayar, S. K. ''Global Measures of Coherence for Edge Detector Evaluation.'' *Proceedings of CVPR99* (2002): 373–379.

Basu, M. ''Gaussian-Based Edge-Detection Methods: A Survey.'' *SMC-C* 32 (August 2002): 252–260.

Bruni, C., de Santis, A., Iacoviello, D. and G. Koch. ''Modeling for Edge Detection Problems in Blurred Noisy Images.'' *IP* 10 (October 2001): 1447–1453.

Canny, J. ''A Computational Approach to Edge Detection.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8* 6 (November 1986): 679–698.

Cumani, A. ''Edge Detection in Multispectral Images.'' *GMIP* 53, no. 1 (1991): 40–51.

Cumani, A., Grattoni, P. and A. Guiducci. ''An Edge-Based Description of Color Images.'' *GMIP* 53, no. 4 (1991): 313–323.

Deutsch, E. S. and Fram, J. R. ''A Quantitative Study of Orientation Bias of Some Edge Detector Schemes.'' *IEEE Transactions on Computers* C-27, no. 3 (March 1978): 205–213.

Dutta, Soumya and Bidyut B. Chaudhuri. ''A Color Edge Detection Algorithm in RGB Color Space, Advances in Recent Technologies in Communication and Computing.'' *International Conference on Advances in Recent Technologies in Communication and Computing* (2009): 337–340.

Evans, A. N. and X. U. Liu. ''A Morphological Gradient Approach to Color Edge Detection.'' *IP* 15, no. 6 (June 2006): 1454–1463.

Fan, J. P., Aref, W. G., Hacid, M. S. and A. K. Elmagarmid. ''An Improved Automatic Isotropic Color Edge Detection Technique.'' *PRL* 22, no. 13 (November 2001): 1419–1429.

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

Huntsberger, T. L. and M. F. Descalzi. ''Color Edge Detection.'' *PRL* 3 (1985): 205–209.

Kaplan, W. *Advanced Calculus*, 2nd ed. Reading: Addison Wesley, 1973.

Kirsch, R. A. ''Computer Determination of the Constituent Structure of Biological Images.'' *Computers and Biomedical Research* 4 (1971): 315–328.

Kitchen, L. and Rosenfeld, A. ''Edge Evaluation Using Local Edge Coherence.'' *IEEE Transactions on Systems, Man, and Cybernetics SMC-11* 9 (September 1981): 597–605.

Liu, G. and R. M. Haralick. ''Assignment Problem in Edge Detection Performance Evaluation.'' *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1 (June 2000): 26–31.

Mendonça, P. R. S., Padfield, D., Miller, J. and M. Turek, ''Bias in the Localization of Curved Edges.'' *ECCV*, 2 (May 2004): 554–565.

Marr, D. and Hildreth, E. ''Theory of Edge Detection.'' *Proceedings of the Royal Society of London Series B* 207 (1980): 187–217.

Medina-Carnicer, R., Carmona-Poyato, Á., Muñoz-Salinas, R. and Madrid-Cuevas. ''Determining Hysteresis Thresholds for Edge Detection by Combining the Advantages and Disadvantages of Thresholding Methods.'' *IP* 19, no. 1 (January 2010): 165–173.

Naik, S. K. and C. A. Murthy. ''Standardization of Edge Magnitude in Color Images.'' *IP* 15, no. 9 (August 2006): 2588–2595.

Nalwa, V. S. and Binford, T. O. ''On Detecting Edges.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8* 6 (November 1986): 699–714.

Nevatia, R. ''A Color Edge Detector and Its Use in Scene Segmentation.'' *SMC* 7, no. 11 (November 1977): 820–826.

Peli, E. ''Feature Detection Algorithm Based on a Visual System Model.'' *PIEEE* 90, no. 1 (January 2002): 78–93.

Prager, J. M. ''Extracting and Labeling Boundary Segments in Natural Scenes.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-2* 1 (January 1980): 16–27.

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

Saber, E., Tekalp, A.M. and G. Bozdagi. ''Fusion of Color and Edge Information for Improved Segmentation and Edge Linking.'' *IVC* 15, no. 10 (October 1997): 769–780.

Scharcanski, J. and A.N. Venetsanopoulos. ''Edge-Detection of Color Images Using Directional Operators.'' *CirSysVideo* 7, no. 2 (April 1997): 397–401.

Shah, M., Sood, A. and R. Jain. ''Pulse and Staircase Edge Models.'' *Computer Vision, Graphics, and Image Processing* 34 (1986): 321–343.

Shen, J. and S. Castan. ''An Optimal Linear Operator for Step Edge Detection.'' *Computer Vision, Graphics, and Image Processing: Graphical Models and Understanding* 54, no. 2 (March 1992): 112–133.

Song, J. Q. A., Cai, M. and M. R. Lyu. ''Edge Color Distribution Transform: An Efficient Tool for Object Detection in Images.'' *ICPR02* I, (2002): 608–611.

Theoharatos, C., Economou, G. and S. Fotopoulos. ''Color Edge Detection Using the Minimal Spanning Tree.'' *PR* 38, no. 4 (April 2005): 603–606.

Toivanen, P. J., Ansamäki, J., Parkkinen, J. P. S. and J. Mielikäinen. ''Edge Detection in Multispectral Images Using the Self-Organizing Map.'' *PRL* 24, no. 16 (December 2003): 2987–2994.

Torre, V. and T. A. Poggio. ''On Edge Detection.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8* 2 (March 1986): 147–163.

Trahanias, P.E. and A.N. Venetsanopoulos. ''Color Edge Detection Using Vector Order Statistics.'' *IP* 2, no. 2 (April 1993): 259–264.

Tremblais, Benoit and Bertrand Augereau. ''A Fast Multi-Scale Edge Detection Algorithm.'' *Pattern Recognition Letters* 25, no. 6 (April 2004): 603–618

Tsai, P., Chang, C. C. and Y.C. Hu. ''An Adaptive Two-Stage Edge Detection Scheme for Digital Color Images.'' *RealTimeImg* 8, no. 4 (August 2002): 329–343.

Yang, C. K., W. H. Tsai. ''Reduction of Color Space Dimensionality by Moment-Preserving Thresholding and Its Application for Edge-Detection in Color Images.'' *PRL* 17, no. 5 (May 1, 1996): 481–490.

CHAPTER 3

Digital Morphology

3.1 Morphology Defined

''Morphology'' means the form and structure of an object, or the arrangements and interrelationships between the parts of an object. Morphology is related to shape, and digital morphology is a way to describe or analyze the shape of a digital, most often raster, object.

The oldest uses of the word relate to language and to biology. In linguistics morphology is the study of the structure of words, and this has been an area of study for a great many years. In biology, morphology relates more directly to the shape of an organism — the shape of a leaf can be used to identify a plant, and the shape of a colony of bacteria can be used to identify its variety. In each case there is an intricate scheme for classification, based on overall shape (elliptical, circular, etc.), type and degree of irregularities (convex, rough or smooth outline, etc.), internal structures (holes, linear or curved features) that has been accumulated over many years of observation. In all cases, shape is a key concept.

The science of digital or mathematical morphology is relatively new, since it is only recently that digital computers have made it practical. On the other hand, the mathematics behind it is simply set theory, which is a well-studied area. The idea underlying digital morphology is that images consist of a set of picture elements (pixels) that collect into groups having a two-dimensional structure (shape). Certain mathematical operations on the set of pixels can be used to enhance specific aspects of the shapes so that they might be, for example, counted or recognized. Basic operations are *erosion*, in which pixels

matching a given pattern are deleted from the image, and *dilation*, in which a small area about a pixel is set to a given pattern. However, depending on the type of image being processed (bi-level, grey-level, or color), the definition of these operations changes, so each must be considered separately.

3.2 Connectedness

Underlying most of digital morphology are the concepts of *connectedness* and *connected regions* among sets of pixels. These are defined on bi-level images, which is the usual domain of morphology. On a standard raster grid, each pixel has a set of neighbors, or pixels that are thought to be ''touching'' it or ''next to'' it. Given that the coordinates of a pixel P are (i, j), the candidates for its neighbors are:

$$\begin{array}{lll} (i-1, j-1) & (i-1, j) & (i-1, j+1) \\ (i, j-1) & (i, j) & (i, j+1) \\ (i+1, j-1) & (i+1, j) & (i+1, j+1) \end{array}$$

The pixels that are different from (i, j) by one in either index would appear to certainly be neighbors. They are the nearest pixels, being a distance of one unit from P either horizontally or vertically. There are four of these, and so they can be called *4-neighbors*. The pixels diagonally next to P, those that differ by one in both coordinates, can also be considered to be neighbors. Treating a raster grid as a chessboard, these pixels are also one unit (square) away from P, and there seems to be a natural neighbor relation between them. There are eight of these pixels, and so they are called *8-neighbors* of P.

A 4-connected region (or *4-region*) is a set of pixels that are 4-connected to each other. It consists of all pixels that are 4-connected, not just a subset, and any pixel in that region is 4-connected to all the rest. Similarly, an 8-connected region (or *8-region*) is a set of pixels in which all pixels are 8-connected to each other. Such regions within an image tend to represent objects that have been scanned by a camera and processed into bi-level areas for processing — for example, text on a page. Thus, it is important to find these, count them, smooth them, and otherwise process them.

Figure 3.1 shows the difference between 4-connected and 8-connected regions. Note that in some of these examples changing a single pixel from black to white changes the number of regions in the image. This illustrates some of the power of digital morphology, and foreshadows the discussion that will take place. Regions can be linked by small numbers of pixels that, when removed according to a set or carefully defined rules, yields a different set of regions that have a clearer meaning or a different meaning altogether.

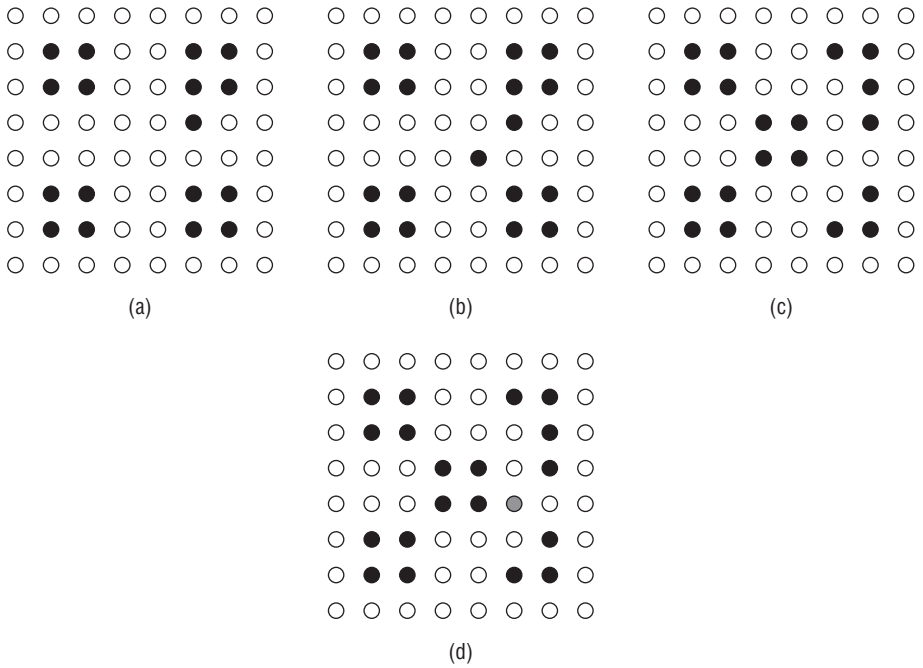


Figure 3.1: (a) A collection of pixels consisting of four 4-connected regions. (b) Adding a pixel seems to connect the regions, but does not; now there are five 4-connected regions. (c) Three 8-connected regions, showing the diagonal connections allowed for 8-regions. There are five 4-regions in this image. (d) Setting one pixel to black (the grey one) connects all regions together.

Digital morphology uses the geometry of small connected sets of pixels to accomplish tasks that are useful in processing regions within images. Morphology can count and mark connected regions in images, can fill in small holes, and can smooth boundaries. These are important functions for some kinds of vision processing, and some form of morphological processing is nearly always one step in the process of locating and recognizing objects in images.

3.3 Elements of Digital Morphology—Binary Operations

As has been explained, most morphological operations are defined on bi-level images — that is, images that consist of either black or white pixels only. These will be called *binary* morphological operators to distinguish them from the less common grey-level morphological operations described in Section 3.2. For the purpose of beginning the discussion, consider the image seen in Figure 3.2a. The set of black pixels form a square object.

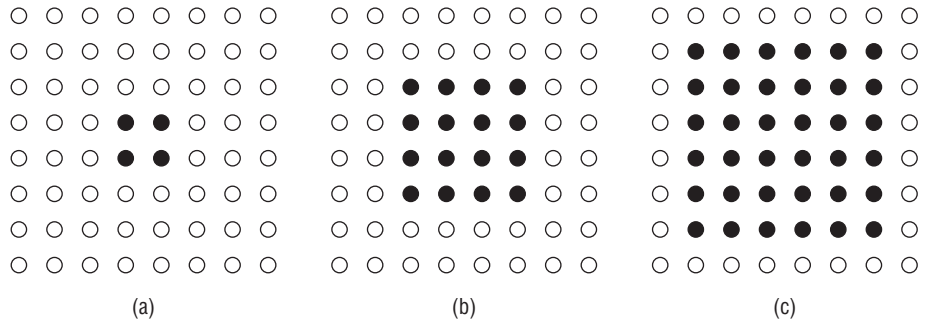


Figure 3.2: The effects of a simple binary dilation one a small object. (a) Original image. (b) Dilation of the original by one pixel. (c) Dilation of the original by two pixels (dilation of (b) by one pixel).

The object in Figure 3.2b is also square, but is one pixel larger in all directions. It was obtained from the previous square by simply setting all white neighbors of any black pixel to black. This amounts to a simple *binary dilation*, so named because it causes the original object to grow larger. Figure 3.2c shows the result of dilating Figure 3.2b by one pixel, which is the same as dilating Figure 3.2a by two pixels; this process could be continued until the entire image consisted entirely of black pixels, at which point the image would stop showing any change.

This is a very basic example of digital morphology, one that can be implemented directly by first *marking* all white pixels having at least one black neighbor, and then setting all the marked pixels to black. This is not, however, how morphological operators are usually implemented. In general, the object is considered to be a mathematical set of black pixels; because each pixel is identified by its row and column indices, a pixel is said to be a point in two-dimensional space (E^2). The set of pixels comprising the object in Figure 3.2a can now be written as $\{(3,3)(3,4)(4,3)(4,4)\}$ if the upper-left pixel in the image has the index (0,0). This set is too awkward to write out in full all the time, so it will simply be called A. The operation shown in Figure 3.2 will be called a ''simple'' binary dilation, because it is the most basic of what will soon be seen to be a large set of possible dilations.

3.3.1 Binary Dilation

Now some definitions of simple set operations can be stated, with the goal being to define dilation in a more general fashion in terms of sets. The *translation* of the set A by the point x is defined, in set notation, as:

$$(A)_x = \{c|c = a + x, a \in A\} \qquad \text{(EQ 3.1)}$$

For example, if x were at (1,2), then the first (upper left) pixel in A_x would be $(3,3) + (1,2) = (4,5)$; all the pixels in A shift down by one row and right by

two columns in this case. This is a translation in the same sense that is seen in computer graphics — a change in position by a specified amount.

The *reflection* of the set A is defined as:

$$\hat{A} = \{c | c = -a, a \in A\} \quad \text{(EQ 3.2)}$$

This is really a rotation of the object A by 180 degrees about the origin. The *complement* of the set A is the set of pixels not belonging to A This would correspond to the white pixels in the figure, or in the language of set theory:

$$A^c = \{c | c \notin A\} \quad \text{(EQ 3.3)}$$

The *intersection* of the two sets A and B is the set of elements (pixels) belonging to both A and B:

$$A \cap B = \{c | ((c \in A) \wedge (c \in B))\} \quad \text{(EQ 3.4)}$$

The *union* of the two sets A and B is the set of pixels that belong to either A or B, or to both:

$$A \cup B = \{c | (c \in A) \vee (c \in B)\} \quad \text{(EQ 3.5)}$$

Finally, completing this collection of basic definitions, the *difference* between the set A and the set B is:

$$A - B = \{c | (c \in A) \wedge (c \notin B)\} \quad \text{(EQ 3.6)}$$

which is the set of pixels that belong to A but not also to B. This is really just the intersection of A with the complement of B.

It is now possible to define more formally what is meant by a dilation. A dilation of the set A by the set B is:

$$A \oplus B = \{c | c = a + b, a \in A, b \in B\} \quad \text{(EQ 3.7)}$$

where A represents the image being operated on, and B is a second set of pixels, a shape that operates on the pixels of A to produce the result; the set B is called a *structuring element*, and its composition defines the nature of the specific dilation. To explore this idea, let A be the set of Figure 3.2a, and let B be the set $\{(0,0)(0,1)\}$. The pixels in the set $C = A + B$ are computed using Equation 3.7, which can be re-written in this case as:

$$A \oplus B = (A + \{(0,0)\}) \cup (A + \{(0,1)\}) \quad \text{(EQ 3.8)}$$

There are four pixels in the set A, and because any pixel translated by (0,0) does not change, those four will also be in the resulting set C after computing $C = A + \{(0,0)\}$:

$$(3,3) + (0,0) = (3,3) \quad (3,4) + (0,0) = (3,4)$$
$$(4,3) + (0,0) = (4,3) \quad (4,4) + (0,0) = (4,3)$$

The result of $A + \{(0,1)\}$ is:

$$(3,3) + (0,1) = (3,4) \quad (3,4) + (0,1) = (3,5)$$
$$(4,3) + (0,1) = (4,4) \quad (4,4) + (0,1) = (4,5)$$

The set C is the result of the dilation of A using structuring element B, and consists of all the pixels listed above (some of which are duplicates). Figure 3.3 illustrates this operation, showing graphically the effect of the dilation. The pixels marked with an X, either white or black, represent the origin of each image. The location of the origin is really quite important. In the example above, if the origin of B were the rightmost of the two pixels the effect of the dilation would be to add pixels to the left of A, rather than to the right. The set B in this case would be $\{(0,-1)(0,0)\}$.

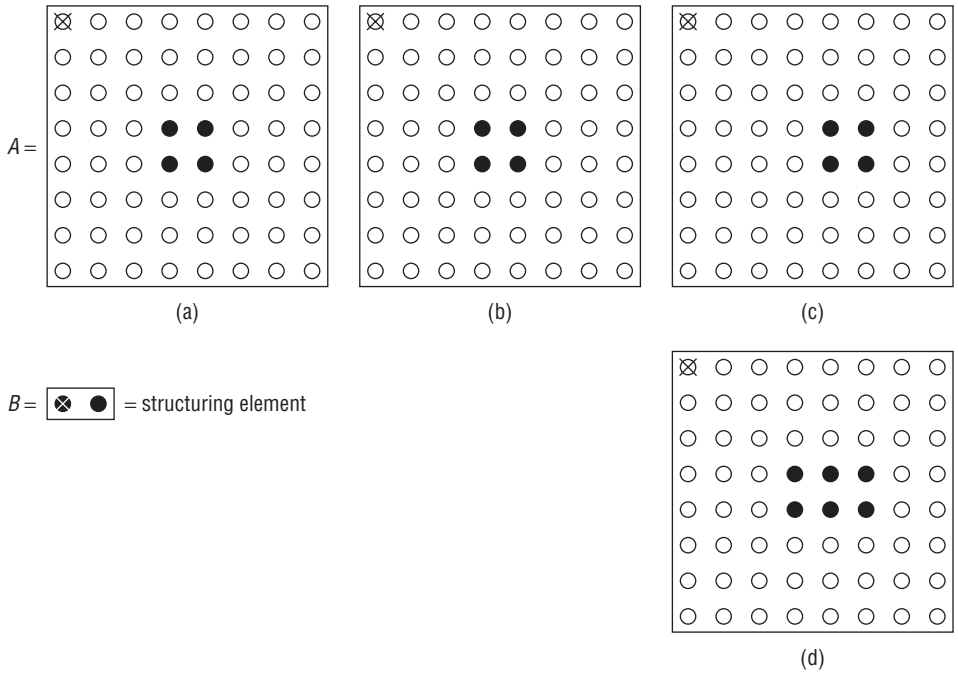


Figure 3.3: Dilation of the set A (Figure 3.2a) by the set B. (a) The two sets. (b) The set obtained by adding (0,0) to all elements of A. (c) The set obtained by adding (0,1) to all elements of A. (d) The union of the two sets is the result of the dilation.

Moving back to the simple binary dilation that was performed in Figure 3.2, one question that remains is: What was the structuring element that was used? Note that the object increases in size in all directions, and by a single pixel. From the example just completed, it was observed that if the structuring element has a set pixel to the right of the origin, then a dilation that uses that structuring element ''grows'' a layer of pixels on the right of the object.

To grow a layer of pixels in all directions, it seems to make sense to use a structuring element having one pixel on every side of the origin — that is, a 3x3 square with the origin at the center. This structuring element will be named ''simple'' in the ensuing discussion and is correct in this instance, although it is not always easy to determine the shape of the structuring element needed to accomplish a specific task.

As a further example, consider the object and structuring element shown in Figure 3.4. In this case, the origin of the structuring element B_1 contains a white pixel, implying that *the origin is not included* in the set B_1. There is no rule against this, but it is more difficult to see what will happen, so the example will be done in detail.

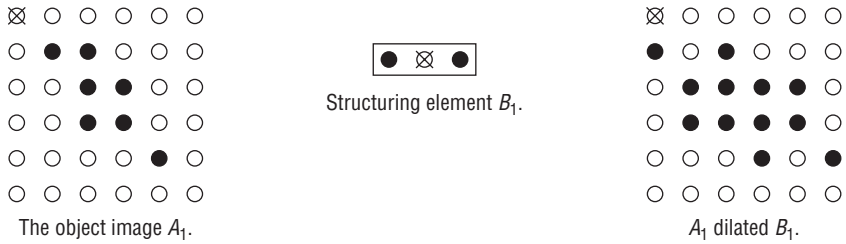


Figure 3.4: Dilation by a structuring element that does not include the origin. Some pixels that are set in the original image are not set in the dilated image.

The image to be dilated, A_1, has the following set representation:

$$A_1 = \{(1,1)\ (2,2)\ (2,3)\ (3,2)\ (3,3)\ (4,4)\}$$

The structuring element B_1 is:

$$B_1 = \{(0,-1)\ (0,1)\}$$

The translation of A_1 by (0, −1) yields

$$(A_1)_{(0,-1)} = \{(1,0)\ (2,1)\ (2,2)\ (3,1)\ (3,2)\ (4,3)\}$$

And the translation of A_1 by (0,1) yields:

$$(A_1)_{(0,1)} = \{(1,2)\ (2,3)\ (2,4)\ (3,3)\ (3,4)\ (4,5)\}$$

The dilation of A_1 by B_1 is the union of $(A_1)_{(0,-1)}$ with $(A_1)_{(0,1)}$, and is shown in Figure 3.11. Notice that the original object pixels, those belonging to A_1, are not necessarily set in the result; (1,1) and (4,4), for example, are set in A_1 but not in $A_1 + B_1$. This is the effect of the origin not being a part of B_1.

The manner in which the dilation is calculated above presumes that a dilation can be considered to be the union of all the translations specified by the structuring element — that is, as

$$A \oplus B = \bigcup_{b \in B} (A)_b \qquad \text{(EQ 3.9)}$$

Not only is this true, but because dilation is commutative, a dilation can also be considered to be the union of all translations of the structuring element by all pixels in the image:

$$A \oplus B = \bigcup_{a \in A} (B)_a \qquad \text{(EQ 3.10)}$$

This gives a clue concerning a possible implementation for the dilation operator. Think of the structuring element as a template, and move it over the image. When the origin of the structuring element aligns with a black pixel in the image all the image pixels that correspond to black pixels in the structuring element are marked, and will later be changed to black. After the entire image has been swept by the structuring element, the dilation calculation is complete. Normally the dilation is not computed in place — that is, where the result is copied over top of the original image. A third image, initially all white, is used to store the dilation while it is being computed.

3.3.2 Implementing Binary Dilation

The general implementation of dilation for bi-level images consists of two parts: a *program* that creates a dilated image given an input image and a structuring element, and a *function* that will do the same but that can be called from another function, and allow dilation to be incorporated into a larger imaging program. The program, which will be called `BinDil`, reads the names of three files from standard input:

```
BinDil
Enter input image filename:                squares.pbm
Enter structuring element filename:        simple.pbm
Enter output filename:                     xx.pbm
```

whereupon the following output is created:

```
PBM file class 1 size 3 columns x 3 rows Max=1
BinDil: Perform a binary dilation on image `squares.pbm'
Structuring element is:
=============================================
Structuring element 3x3 origin at (1,1)
                 1     1     1
                 1     1     1
                 1     1     1
=============================================
PBM file class 1 size 10 columns x 10 rows Max=1
```

All three arguments are filenames:

- `squares.pbm` is the name of the image file that contains the image to be dilated.

- `simple.pbm` contains the data for the structuring element.
- `xx.pbm` is the name of the file that will be created to hold the resulting dilated image.

Both the input file and the output file will be in PBM image file format. The structuring element file has its own special format, because the location of the origin is needed. Because dilation is commutative, though, the structuring element should also be a PBM image. It was decided to add a small ''feature'' to the definition of a PBM file — if a comment begins with the string `#origin`, then the coordinates of the origin of the image will follow, first the column and then the row. Such a file is still a PBM image file, and can still be read in and displayed as such because the new specification occurs in a comment. If no origin is explicitly given, it is assumed to be at (0,0).

Thus, the PBM file for the structuring element B_1 of Figure 3.4 would be:

```
P1
#origin 1 0
3 1
1 0 1
```

This file will be called `B1.pbm`. To perform the dilation seen in Figure 3.4, the call to `BinDil` would be:

```
BinDil
Enter input image filename:                A1.pbm
Enter structuring element filename:        B1.pbm
Enter output filename:                     A1dil.pbm
```

where the file `A1.pbm` contains the image A_1, and `A1dil.pbm` will be the file into which the dilated image will be written.

The program `BinDil` really does not do very much. It merely reads in the images and passes them to the function that does the work: the C function `bin_dilate`, defined as

```
int bin_dilate (IMAGE im, SE p);
```

This function implements a dilation by the structuring element pointed to by `p` by moving the origin of `p` to each of the black pixel positions in the image `im`, and then copying the black pixels from `p` to the corresponding positions in the output image. Figure 3.5 shows this process, which is basically that specified in Equation 3.9, and which has a strong resemblance to a convolution.

As shown in Figure 3.6, the function `bin_dilate` looks through the data image for black pixels, calling `dil_apply` when it finds one; this function performs the actual copy from the current position of the structuring element to the output (dilated) image. A temporary image is used for the result, which is copied over the input image after the dilation is complete.

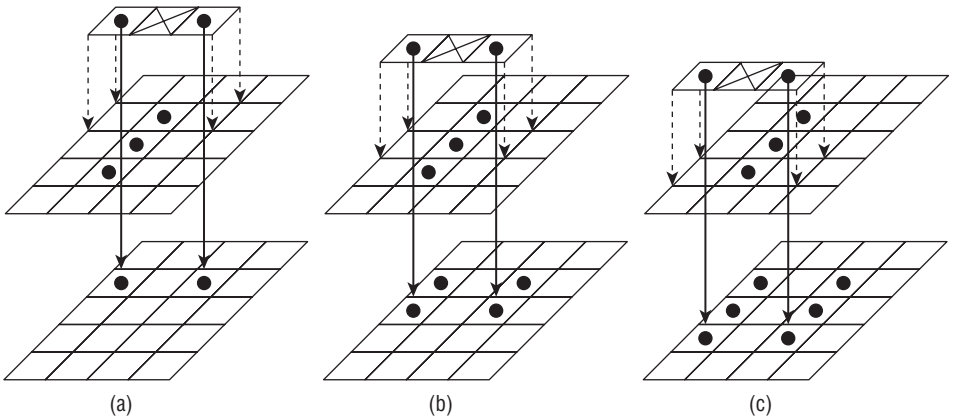


Figure 3.5: Dilating an image using a structuring element. (a) The origin of the structuring element is placed over the first black pixel in the image, and the pixels in the structuring element are copied into their corresponding positions in the result image. (b) Then the structuring element is placed over the next black pixel in the image and the process is repeated. (c) This is done for every black pixel in the image.

3.3.3 Binary Erosion

If dilation can be said to add pixels to an object, or to make it bigger, erosion will make an image smaller. In the simplest case, a binary erosion will remove the outer layer of pixels from an object. For example, Figure 3.2b is the result of such a simple erosion process applied to Figure 3.2c. This can be implemented by marking all black pixels having at least one white neighbor and then setting to white all the marked pixels. The structuring element implicit in this implementation is the same 3x3 array of black pixels that defined the simple binary dilation.

In general, the erosion of image A by structuring element B can be defined as

$$A \ominus B = \{c|(B)_c \subseteq A\} \qquad \text{(EQ 3.11)}$$

In other words, it is the set of all pixels c such that the structuring element B translated by c corresponds to a set of black pixels in A. That the result of an erosion is a subset of the original image seems clear enough; any pixels that do not match the pattern defined by the black pixels in the structuring element will not belong to the result. However, the manner in which the erosion removes pixels is not clear, at least at first, so a few examples are in order, and the statement above that the eroded image is a subset of the original is not necessarily true if the structuring element does not contain the origin.

```
void dil_apply (IMAGE im, SE p, int ii, int jj, IMAGE res)
{
        int i,j, is,js, ie, je, k;

/* Find start and end pixel in IM */
       is = ii - p->oi; js = jj - p->oj;   ie = is + p->nr; je = js + p->nc;

/* Place SE over the image from (is,js) to (ie,je). Set pixels
in RES if the corresponding SE pixel is 1; else do nothing. */
        for (i=is; i<ie; i++)
                for (j=js; j<je; j++)
                {
                        if (range(im,i,j))
                        {
                                k = (p->data[i-is][j-js] == 1);
                                if (k>=0) res->data[i][j] |= k;
}               }        }

int bin_dilate (IMAGE im, SE p)
{
        IMAGE tmp;
        int i,j;

/* Source image empty? */
        if (im==0)
        {
                printf (˜Bad image in BIN_DILATE\n˝);
                return 0;
        }

 /* Create a result image */
        tmp = newimage (im->info->nr, im->info->nc);
        if (tmp == 0)
        {
                printf (˜Out of memory in BIN_DILATE.\n˝);
                return 0;
        }
        for (i=0; i<tmp->info->nr; i++)
                for (j=0; j<tmp->info->nc; j++)
                        tmp->data[i][j] = 0;
 /* Apply the SE to each black pixel of the input */
        for (i=0; i<im->info->nr; i++)
                for (j=0; j<im->info->nc; j++)
                        if (im->data[i][j] == WHITE)
        dil_apply (im, p, i, j, tmp);
 /* Copy result over the input */
        for (i=0; i<im->info->nr; i++)
                for (j=0; j<im->info->nc; j++)
                        im->data[i][j] = tmp->data[i][j];

        /* Free the result image - it was a temp */
        freeimage (tmp);
        return 1;
        }
```

Figure 3.6: C source code for the dilation of a binary image by a binary structuring element.

First, a simple example. Consider the structuring element

$$B = \{(0,0)\ (1,0)\}$$

and the object image

$$A = \{(3,3)\ (3,4)\ (4,3)\ (4,4)\}$$

The set $A \ominus B$ is the set of translations of B that align B over a set of black pixels in A. This means that not all translations need to be considered, but only those that initially place the origin of B at one of the members of A. There are four such translations:

$$B_{(3,3)} = \{(3,3)\ (4,3)\}$$
$$B_{(3,4)} = \{(3,4)\ (4,4)\}$$
$$B_{(4,3)} = \{(4,3)\ (5,3)\}$$
$$B_{(4,4)} = \{(4,4)\ (5,4)\}$$

In two cases, $B_{(3,3)}$ and $B_{(3,4)}$, the resulting (translated) set consists of pixels that are all members of A, and so those pixels will appear in the erosion of A by B. This example is illustrated in Figure 3.7.

Now consider the structuring element $B_2 = \{(1,0)\}$; in this case the origin is not a member of B_2. The erosion $A \ominus B$ can be computed as before, except that now the origin of the structuring element need not correspond to a black pixel in the image. There are quite a few legal positions, but the only ones that result in a match are:

$$B_{(2,3)} = \{(3,3)\}$$
$$B_{(2,4)} = \{(3,4)\}$$
$$B_{(3,3)} = \{(4,3)\}$$
$$B_{(3,4)} = \{(4,4)\}$$

This means that the result of the erosion is

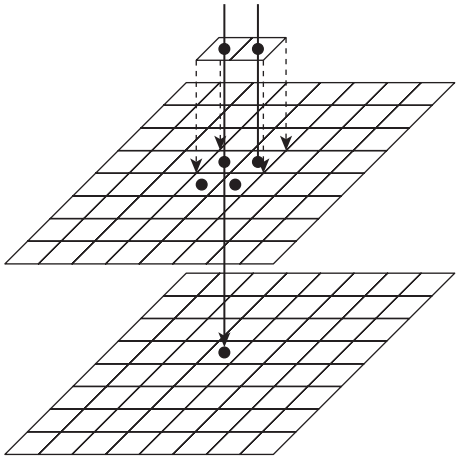
$$\{(2,3)\ (2,4)\ (3,3)\ (3,4)\}$$

which is *not* a subset of the original.

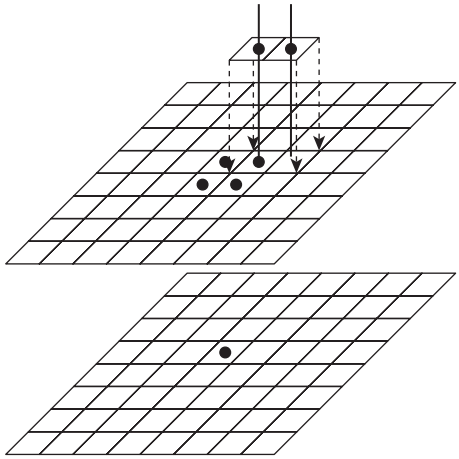
It is important to realize that erosion and dilation are not inverse operations. Although there are some situations where an erosion will undo the effect of a dilation exactly, this is not true in general. Indeed, as will be observed later, this fact can be used to perform useful operations on images. However, erosion and dilation are *duals* of each other in the following sense:

$$(A \ominus B)^c = A^c \oplus \hat{B} \qquad \text{(EQ 3.12)}$$

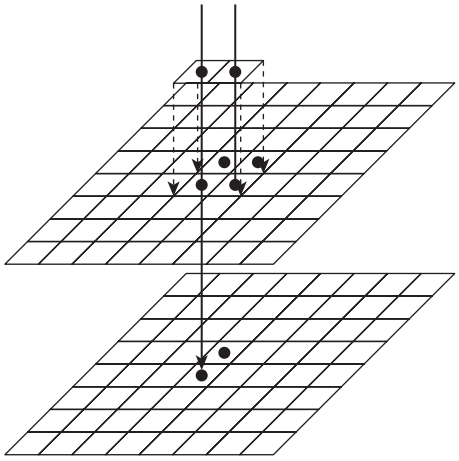
In more or less plain English, this says that the complement of an erosion is the same as a dilation of the complement image by the reflected structuring element. If the structuring element is symmetrical, reflecting it does not change it, and the implication of Equation 3.12 is that the complement of an erosion of an image is the dilation of the background, in the case where simple is the structuring element.



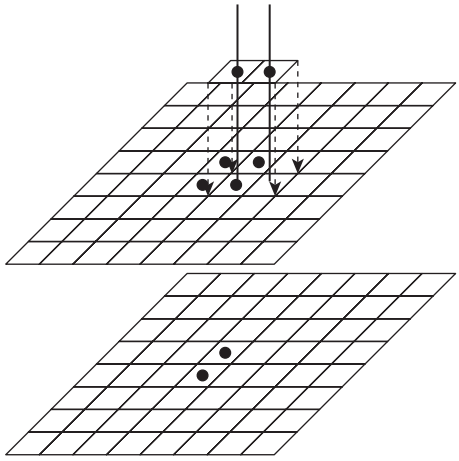
The structuring element is translated to the position of a black pixel in the image. In this case all members of the structuring element correspond to black image pixels, so the result is a black pixel.



Now the structuring element is translated to the next black pixel in the image, and there is one pixel that does not match. The result is a white pixel.



At the next translation there is another match so, again, the pixel in the output image that corresponds to the translated origin of the structuring element is set to black.



The final translation is not a match, and the result is a white pixel. The remaining image pixels are white and could not match the origin of the structuring element; they need not be considered.

Figure 3.7: Binary erosion using a simple structuring element.

The proof of the erosion-dilation duality is fairly simple and may yield some insights into how morphological expressions are manipulated and validated. The definition of erosion is:

$$A \ominus B = \{z|(B)_z \subseteq A\} \quad \text{(EQ 3.13)}$$

So, the complement of the erosion is:

$$(A \ominus B)^c = \{z|(B)_z \subseteq A\}^c \quad \text{(EQ 3.14)}$$

If $(B)_z$ is a subset of A, then the intersection of $(B)_z$ with A is not empty:

$$(A \ominus B)^c = \{z|((B)_z \cap A) \neq \varnothing\}^c \quad \text{(EQ 3.15)}$$

but the intersection with A^c will be empty:

$$\left\{z|(B)_z \cap A^c = \varnothing\right\}^c \quad \text{(EQ 3.16)}$$

and the set of pixels not having this property is the complement of the set that does:

$$\left\{z|((B)_z \cap A^c) \neq \varnothing\right\} \quad \text{(EQ 3.17)}$$

By the definition of translation in Equation 3.1, if $(B)_z$ intersects A^c, then

$$\{z|b + z \in A^c, b \in B\} \quad \text{(EQ 3.18)}$$

which is the same thing as

$$\{z|b + z = a, a \in A^c, b \in B\} \quad \text{(EQ 3.19)}$$

Now if $a = b + z$, then $z = a - b$:

$$\{z|z = a - b, a \in A^c, b \in B\} \quad \text{(EQ 3.20)}$$

Finally, using the definition of reflection, if b is a member of B, then $-b$ is a member of the reflection of B:

$$\{z|z = a + b, a \in A^c, b \in \hat{B}\} \quad \text{(EQ 3.21)}$$

which is the definition of $A^c \oplus \hat{B}$.

The erosion operation also brings up an issue that was not a concern about dilation: the idea of a ''don't care'' state in the structuring element. When using a strictly binary structuring element to perform an erosion, the member black pixels must correspond to black pixels in the image in order to set the pixel in the result, but the same is not true for a white (0) pixel in the structuring element. We don't care what the corresponding pixel in the image might be when the structuring element pixel is white.

Figure 3.8 gives some examples of erosions of a simple image by a collection of different structuring elements. The basic shape of the structuring element

is, in each case, identified if it appears in the data image. The intent of SE2, for example, is to identify ''T'' intersections of a vertical line with a horizontal line on the left, and SE3 and SE4 attempt to isolate corners. SE6 has three white pixels spacing apart two black ones; at first glance it might be used to locate horizontal lines spaced three pixels apart, but it will also respond to vertical line segments. This sort of unexpected (to the beginner) behavior leads to difficulties in designing structuring elements for specific tasks.

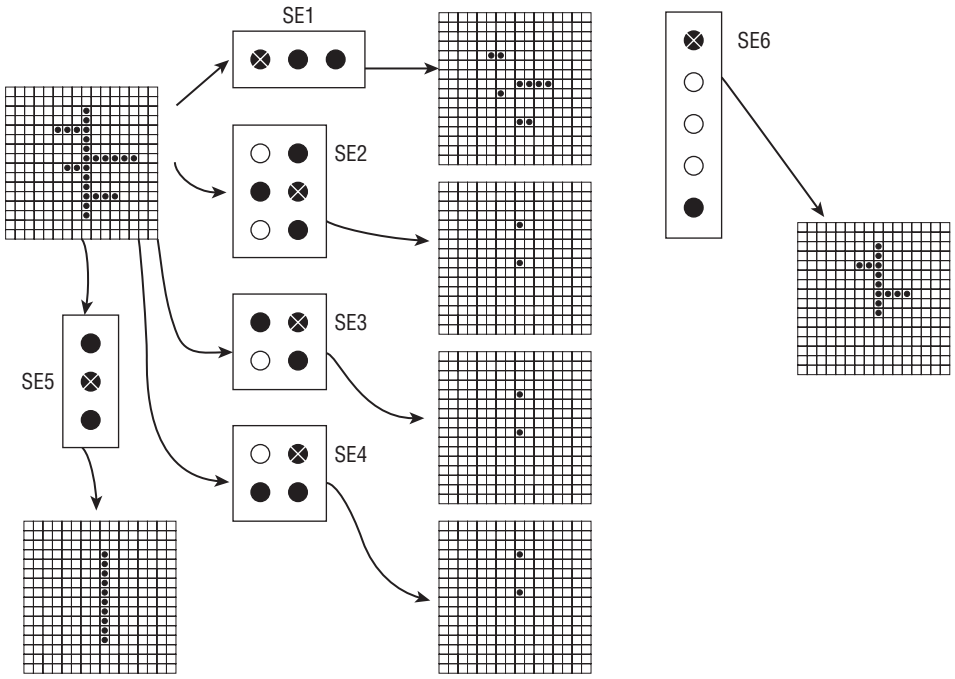


Figure 3.8: Examples of erosions by various structuring elements. (left and above) simple binary structuring elements.

Figure 3.9 is a better illustration of the use of erosion elements in a practical sense. The problem is to design a structuring element that will locate the staff lines in a raster image of printed music. The basic problem is to isolate the symbols, so once identified the staff lines will be removed. The structuring element consists of five horizontal straight line segments separated by ''don't care'' pixels — the latter corresponds to whatever occupies the space between the staff lines: note heads, sharps, etc. In effect, these elements act as spacers, permitting the combination of five distinct structuring elements into one.

After an erosion by the structuring element, each short section of staff lines has been reduced to a single pixel. The staff lines can be regenerated by a dilation of the eroded image by the same structuring element (Figure 3.9d). If it is necessary to remove the staff lines, subtract this image from the original

(Figure 3.9e). There are now gaps in the image where the lines used to be, but otherwise the music symbols are free of the lines. A further morphological step can fill in some of the gaps (Figure 3.9f).

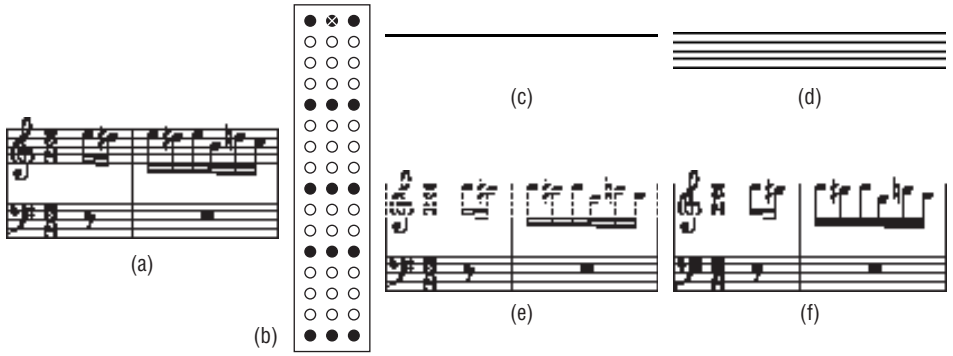


Figure 3.9: Morphological removal of staff lines from printed music. (a) The original image. (b) The structuring element. (c) Result of the erosion of (a) by (b). (d) Result of dilating again by the same structuring element. (e) Subtract (d) from (a). (f) Use a simple morphological operator to fill in the gaps.

3.3.4 Implementation of Binary Erosion

As was done previously in the case of dilation, the implementation will consist of a program that creates an eroded image given an input image and a structuring element, and a function that does the actual work. The program is in the same style as `BinDil`, and is called `BinErode`:

```
BinErode
Enter input image file name:                squares.pbm
Enter structuring element file name:        simple.pbm
Enter output filename:                      xx.pbm
```

The PBM file for the structuring element of Figure 3.9b would be:

```
P1
#origin 1 0
3 16
1 1 1
0 0 0
0 0 0
0 0 0
1 1 1
0 0 0
0 0 0
0 0 0
1 1 1
0 0 0
```

```
0 0 0
1 1 1
0 0 0
0 0 0
0 0 0
1 1 1
```

This file will be called `elise_se.pbm`. To perform the dilation seen in Figure 3.9, the call to `BinErode` would be:

```
BinErode
Enter input image file name:                elise.pbm
Enter structuring element file name:        elise_SE.pbm
Enter output filename:                      out1.pbm
```

where the `elise.pbm` file contains the image 3.8a, and `out1.pbm` will be the file into which the dilated image will be written (Figure 3.9c). The C function `bin_erode` is implemented in a very similar manner to `bin_dilate`, and appears in Figure 3.10.

3.3.5 Opening and Closing

The application of an erosion immediately followed by a dilation using the same structuring element is referred to as an *opening* operation. The name opening is a descriptive one, describing the observation that the operation tends to ''open'' small gaps or spaces between touching objects in an image. This effect is most easily observed when using the simple structuring element. Figure 3.11 shows an image having a collection of small objects, some of them touching each other. After an opening using simple, the objects are better isolated and might now be counted or classified.

Figure 3.11 also illustrates another, and quite common, use of opening: the removal of noise. When a noisy grey-level image is thresholded some of the noise pixels are above the threshold, and result in isolated pixels in random locations. The erosion step in an opening will remove isolated pixels as well as boundaries of objects, and the dilation step will restore most of the boundary pixels without restoring the noise. This process seems to be successful at removing spurious black pixels, but does not remove the white ones.

The example in Figure 3.9 is actually an example of opening, albeit with a more complex structuring element. The image was eroded, leaving only a horizontal line, and then dilated by the same structuring element, which is certainly an opening. What is being eroded in this case is all portions of the image that are not staff lines, which the dilation subsequently restores. The same description of the process applies to Figure 3.11; what is being eroded is all parts of the image that are not small black squares, which are restored by the dilation thus removing everything except that in which we are interested.

```
/* Apply a erosion step on one pixel of IM, result to RES */

 void erode_apply (IMAGE im, SE p, int ii, int jj, IMAGE res)
 {
        int i,j, is,js, ie, je, k, r;

        /* Find start and end pixel in IM */
        is = ii - p->oi; js = jj - p->oj;
        ie = is + p->nr; je = js + p->nc;

        /* Place SE over the image from (is,js) to (ie,je). Set
        pixels in RES
        if the corresponding pixels in the image agree. */
        r = 1;
        for (i=is; i<ie; i++)
                for (j=js; j<je; j++)
                {
                        if (range(im,i,j))
                        {
                                k = p->data[i-is][j-js];
                                if (k == 1 && im->data[i][j]==0) r = 0;
                        } else if (p->data[i-is][j-js] != 0) r = 0;
                }
                res->data[ii][jj] = r;
 }       }

int bin_erode (IMAGE im, SE p)
{
        IMAGE tmp;
        int i,j;

/* Create a result image */
        tmp = newimage (im->info->nr, im->info->nc);
        for (i=0; i<tmp->info->nr; i++)
                for (j=0; j<tmp->info->nc; j++)
                        tmp->data[i][j] = 0;
/* Apply the SE to each black pixel of the input */
        for (i=0; i<im->info->nr; i++)
                for (j=0; j<im->info->nc; j++)
                        erode_apply (im, p, i, j, tmp);

/* Copy result over the input */
        for (i=0; i<im->info->nr; i++)
                for (j=0; j<im->info->nc; j++)
                        im->data[i][j] = tmp->data[i][j];

/* Free the result image - it was a temp */
        freeimage (tmp);
        return 1;
}
```

Figure 3.10: Slightly shortened version of the C source code for bin_erode. (See the website for the complete version.) Some error checking has been removed for brevity.

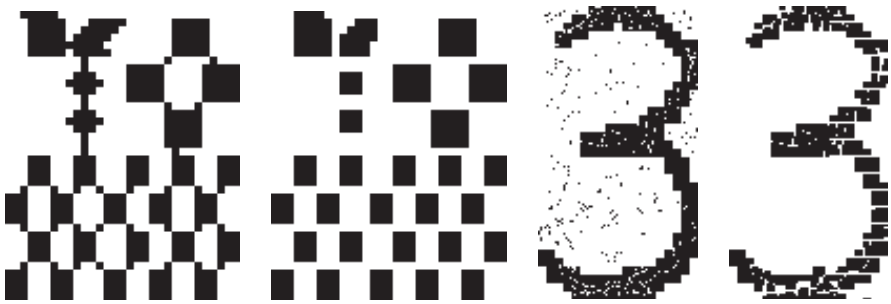


Figure 3.11: The use of opening. (a) An image having many connected objects. (b) Objects can be isolated by opening using the simple structuring element. (c) An image that has been subjected to noise. (d) The noisy image after opening, showing that the black noise pixels have been removed.

A *closing* is similar to an opening, except that the dilation is performed first, followed by an erosion using the same structuring element. If an opening creates small gaps in the image, a closing will fill them, or ''close'' the gaps. Figure 3.12a shows a closing applied to the image of Figure 3.11d, which you may remember was opened in an attempt to remove noise. The closing removes much of the white pixel noise, giving a fairly clean image.

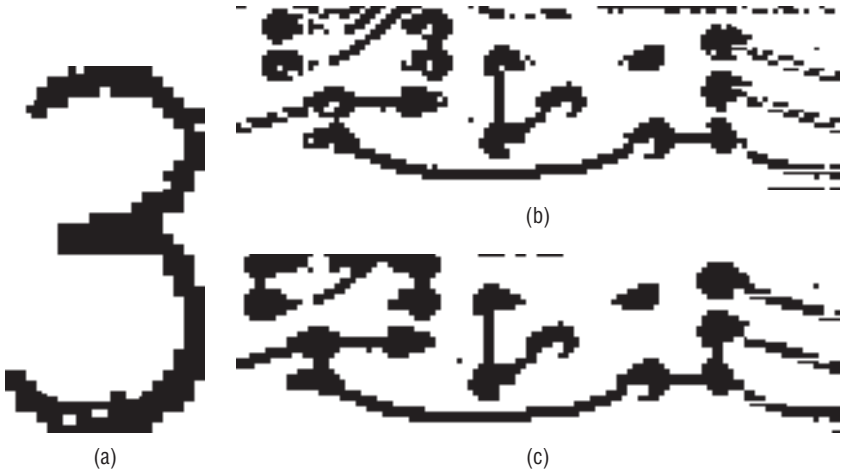


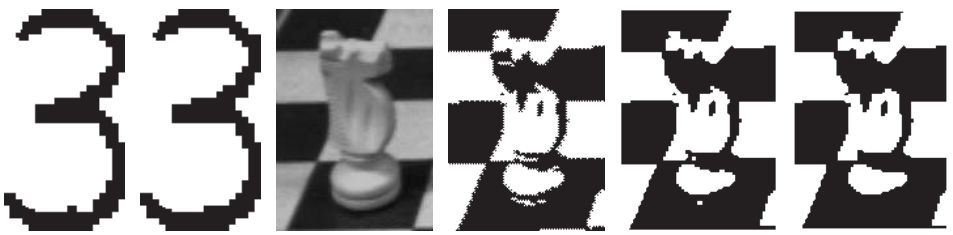
Figure 3.12: The closing operation. (a) The result of closing Figure 3.11d using the simple structuring element. (b) A thresholded image of a circuit board, showing broken traces. (c) The same image after closing, showing that most of the breaks have been repaired.

The same figure shows an application of a closing to reconnect broken features. Figure 3.12b is a section of a printed circuit board that has been thresholded. Noise somewhere in the process has resulted in the traces

connecting the components being broken in a number of places. Closing this image (Figure 3.12c) fixes many of these breaks, but not all of them. It is important to realize that when using real images it is rare for any single technique to provide a complete and perfect solution to an image processing or vision problem. A more complete method for fixing the circuit board may use four or five more structuring elements and two or three other techniques outside of morphology.

Closing can also be used for smoothing the outline of objects in an image. Sometimes digitization followed by thresholding can give a jagged appearance to boundaries; in other cases the objects are naturally rough, and it may be necessary to determine how rough the outline is. In either case, closing can be used. However, more than one structuring element may be needed, because the simple structuring element is only useful for removing or smoothing single pixel irregularities. Another possibility is repeated application of dilation followed by the same number of erosions; N dilation/erosion applications should result in the smoothing of irregularities of N pixels in size.

First consider the smoothing application, and for this purpose Figure 3.12a will be used as an example. This image has been both opened and closed already, and another closing will not have any effect. However, the outline is still jagged, and there are still white holes in the body of the object. An opening of depth 2 — that is, two dilations followed by two erosions — gives Figure 3.13a.



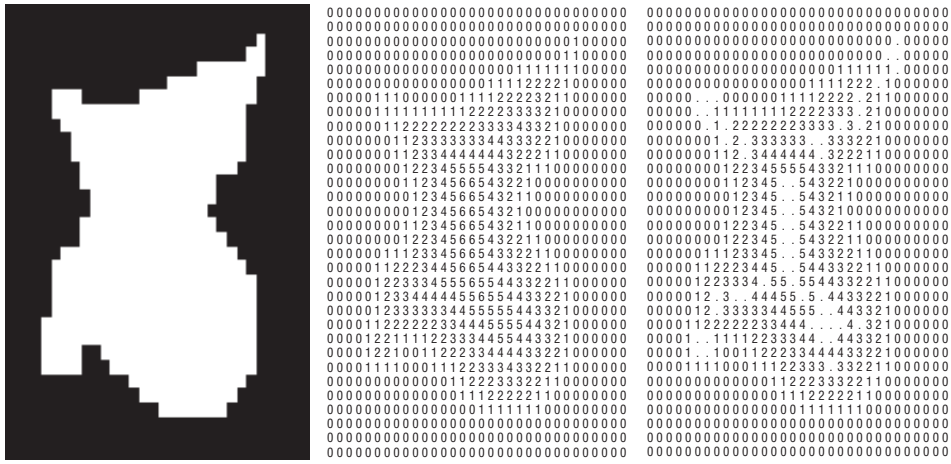
(a) (b) (c) (d) (e) (f)

Figure 3.13: Multiple closings for outline smoothing. (a) Glyph from Figure 3.12a after a depth 2 closing. (b) After a depth 3 closing. (c) A chess piece. (d) Thresholded chess piece showing irregularities in the outline and some holes. (e) Chess piece after closing. (f) Chess piece after a depth 2 closing.

Note that the holes have been closed and that most of the outline irregularities are gone. On opening of depth 3, very little change is seen (one outline pixel is deleted), and no further improvement can be hoped for. The example of the chess piece in the same figure shows more specifically the kind of irregularities introduced sometimes by thresholding, and illustrates the effect that closings can have in this case.

Most opening and closings use the simple structuring element in practice. The traditional approach to computing an opening of depth N is to perform N consecutive binary erosions followed by N binary dilations. This means that computing all the openings of an image up to depth ten requires that 110 erosions or dilations be performed. If erosion and dilation are implemented in a naive fashion, this will require 220 passes through the image. The alternative is to save each of the ten erosions of the original image; each of these is then dilated by the proper number of iterations to give the ten opened images. The amount of storage required for this latter option can be prohibitive, and if file storage is used the I/O time can be large also.

A fast erosion method is based on the *distance map* of each object, where the numerical value of each pixel is replaced by a new value representing the distance of that pixel from the nearest background pixel. Pixels on a boundary would have a value of 1, being that they are 1 pixel width from a background pixel; pixels that are 2 widths from the background would be given a value of 2, and so on. The result has the appearance of a contour map, where the contours represent the distance from the boundary. For example, the object shown in Figure 3.14a has the distance map shown in Figure 3.14b.



(a) (b) (c)

Figure 3.14: Erosion using a distance map. (a) A blob as an example of an image to be eroded. (b) The distance map of the blob image. (c) Nodal pixels in this image are shown as periods (.).

The distance map contains enough information to perform an erosion by any number of pixels in just one pass through the image; in other words, all erosions have been encoded into one image. This *globally eroded* image can be produced in just two passes through the original image, and a simple thresholding operation will give any desired erosion.

There is also a way, similar to that of global erosion, to encode all possible openings as one grey-level image, and all possible closings can be computed at the same time. First, as in global erosion, the distance map of the image is found. Then all pixels that do *not* have at least one neighbor nearer to the background and one neighbor more distant are located and marked: these will be called nodal pixels. Figure 3.14c shows the nodal pixels associated with the object of Figure 3.14a. If the distance map is thought of as a three-dimensional surface where the distance from the background is represented as height, every pixel can be thought of as being the peak of a pyramid having a standardized slope. Peaks not included in any other pyramid are the nodal pixels. One way to locate nodal pixels is to scan the distance map, looking at all object pixels; find the minimum and maximum value of all neighbors of the target pixel, and compute MAX-MIN. If this value is less than the maximum possible, which is 2 when using 8-distance, the pixel is nodal.

To encode all openings of the object, a digital disk is drawn centered at each nodal point. The pixel values and the extent of the disk are equal to the value of the nodal pixel. If a pixel has already been drawn, it will take on the larger of its current value or the new one being painted. The resulting object has the same outline as the original binary image, so the object can be re-created from the nodal pixels alone. In addition, the grey levels of this globally opened image represent an encoding of all possible openings. As an example, consider the disk shaped object in Figure 3.15a and the corresponding distance map of Figure 3.15b.

There are nine nodal points: four have the value 3, and the remainder have the value 5. Thresholding the encoded image yields an opening having depth equal to the threshold.

All possible closings can be encoded along with the openings if the distance map is changed to include the distance of background pixels from an object. Closings are coded as values less than some arbitrary central value (say, 128) and openings are coded as values greater than this central value.

As a practical case, consider an example from geology. To a geologist, the pores that exist in oil-bearing (reservoir) rock are of substantial interest; oil resides in these pores. Porosity of reservoir rock can be measured by slicing the rock into thin sections after filling the pores with a colored resin. The slices show microscopic features of grain and pore space, and one method of characterizing the shapes of the pores is to examine the differences between openings of increasing depth. Openings of higher orders are smoother than those of lower orders, and the difference between the order N opening and the order N + 1 opening is referred to as the *roughness* of order N. The histogram of the pixels in each opened pore image by order of roughness yields a *roughness spectrum*, which has been extensively applied to the classification of pore shape.

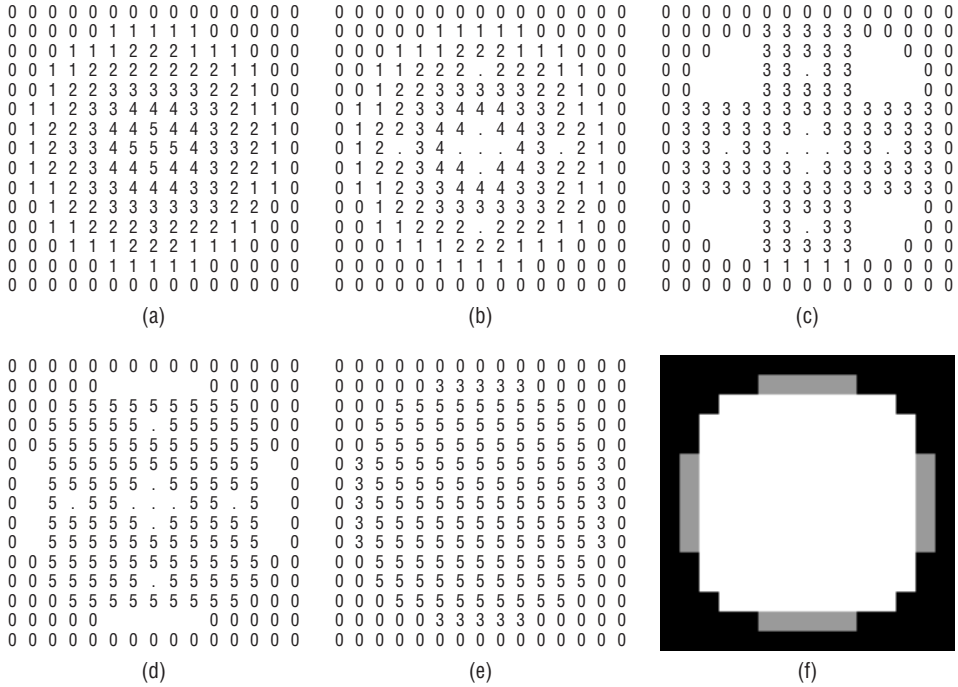


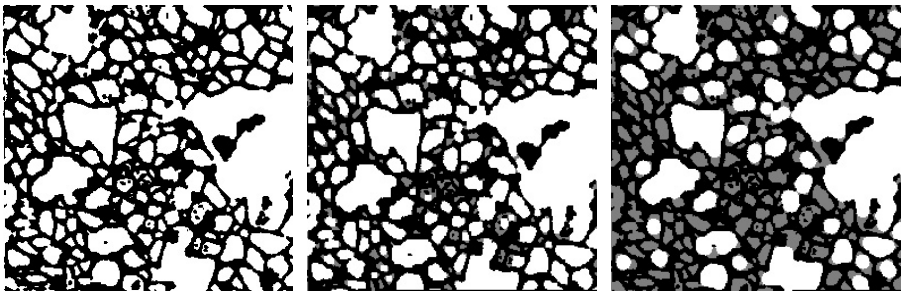
Figure 3.15: Multiple closings for outline smoothing. (a) Glyph from Figure 3.12a after a depth 2 closing. (b) After a depth 3 closing. (c) A chess piece. (d) Thresholded chess piece showing irregularities in the outline and some holes. (e) Chess piece after closing. (f) Chess piece after a depth 2 closing.

Figure 3.16 is a sample reservoir rock pore image having a size of 300x300 pixels. The results of opening using the distance encoded image are identical with the results from the traditional erode-dilate method, although all openings were found in only three passes through the image. Two sample openings are shown, with the deleted pixels shown in grey. The roughness spectrum can be calculated by repeatedly thresholding the image and counting the pixels remaining after each step, but there is an easier way. The roughness spectrum can now be found by simply computing the grey-level histogram of the globally opened image.

3.3.6 MAX—A High-Level Programming Language for Morphology

Many of the tools needed for experimenting with morphology are provided in this chapter and on the accompanying website in the form of C functions and procedures. However, their use requires a certain fluency in the C language, and any given experiment may involve a sequence of compilation/test/debug steps that can be time consuming. To avoid this, and to

encourage experimentation with morphological techniques, a programming language named *MAX* (Morphology And eXperimentation) has been devised. MAX is a very simple language in the style of Pascal and Modula, the sole purpose of which is to evaluate morphological expressions.



(a) (b) (c)

Figure 3.16: Computing the roughness of reservoir rock pores. (a) A pore image. (b) Opening of depth 3. Grey areas are pixels that have been changed. (c) Opening of depth 6. Used by permission of Dr. N. Wardlaw, Department of Geology and Geophysics, University of Calgary.

To help explain the use of MAX, here is a simple program that reads in two images and copies them to new image files:

```
// Test of input and output in MAX.
image a, b;
begin

// Read in two images
       do a << `a″;
       do b << `b″;

// Copy them to new files.
       do b>>″copyb″;
       do a>>″copya″;
end;
```

The `//` characters begin a comment, which extends to the end of the line. The MAX compiler ignores any characters within a comment. The first statements in this program are declarations; all variables must be declared between the beginning of the file and the first `begin` statement. MAX recognizes only three types:

- `IMAGE` is used for both data and structuring elements interchangeably.
- `INT` is a traditional integer type.
- `PIXEL` is a pair of integers that represent the row and column indices of a pixel in an image.

Variables may be declared to be of any of these three types by stating the type name followed by a list of variables having that type. `STRING` constants are allowed in some cases, but there are no string variables.

The executable part of a MAX program is a sequence of statements enclosed by a `begin` statement at the beginning and an `end` statement at the end. A semicolon (`;`) separates each statement from the next in a statement sequence, except before an `end` statement. The preceding program has only one kind of statement: a `DO` statement. This is simply the word ''do'' followed by any legal expression, and permits the expression to be evaluated without assigning the value to anything.

The only operators seen above are `<<` (input) and `>>` (output), in this case applied to images. The expression `a << "a"` reads an image in PBM file format from the file named `"a"` into the image variable a; if the string constant is the empty string `""`, then standard input is used as the input file, and if the string is `"$1"`, then the first command line argument is copied into the string, allowing a program to open dynamically specified file. In the preceding program, two images are read into variables. These are immediately written again under different names: the output operator `>>` works in the same way as the input operator, creating a PBM image file from the specified image.

MAX has six different types of statements, designed to allow a great deal of flexibility in what kinds of morphological operations can be easily implemented. In summary, the legal statements are as follows:

```
if ( expression ) then statement
```

If the `expression` evaluates to a non-zero integer (TRUE), the statement that follows will be executed. The statement can be a sequence of statements enclosed by begin–end.

```
if ( expression ) then statement1 else statement2
```

This is another form of the `if` statement, but if the `expression` is `0` (FALSE), `statement2` is executed.

```
loop ... end
```

Repeat a sequence of code. When the `end` is reached, execution resumes from the statement following the `loop` statement. Statements within the `loop` must be separated by semicolons.

```
exit N when expression
```

Exit from a loop if the expression evaluates to a non-zero integer. If *N* is omitted, the exit branches to the statement following the end of the nearest enclosing loop. If `N=2`, we escape from the nearest 2 nested loops, and so on.

```
do expression
```

Evaluate the `expression`. This is mainly useful for input and output.

`message` *`expression`*

Print a message to standard output. If the `expression` is a string constant, that string is printed on the screen. Integers and pixels can also be printed, and images will be printed as a two-dimensional array of integers as if they were structuring elements.

`assignment`

The assignment operator is `:=`. The type of the variable on the left of the assignment operator must agree with the expression on the right of it. If the expression is an image, the result of the assignment is a copy of that image.

For a small language, MAX has quite an array of operators, many of which can operate on all three possible data types. The following table is a convenient summary of all the legal MAX operators. Note that LEFT is the name of the structuring element specific to the left of the operator, and RIGHT is the structuring element to the right.

OPERATOR	LEFT	RIGHT	RESULT	DESCRIPTION
++	image	image	image	Dilate LEFT by RIGHT (for example, A++B).
--	image	image	image	Erode LEFT by RIGHT (for example, EG A--B).
<=	image	image	int	Subset: Is LEFT a subset of RIGHT?
<=	int	int	int	Less than or equal to (integer).
>=	image	image	int	Subset: Is RIGHT a subset of LEFT?
>=	int	int	int	Greater or equal (integer).
>	image	image	int	Proper subset.
>	int	int	int	Greater than.
<	image	image	int	Proper subset.
<	int	int	int	Less than.
<>	image	image	int	Images not the same.
<>	int	int	int	Not equal.

OPERATOR	LEFT	RIGHT	RESULT	DESCRIPTION
<>	pixel	pixel	int	Not equal, pixels.
=	image	image	int	Are LEFT and RIGHT equal?
=	int	int	int	Integer equality.
=	pixel	pixel	pixel	Pixel equality.
=	image	int	int	Are all pixels in the image equal to the given integer?
–	image	image	image	Set difference, LEFT-RIGHT.
–	int	int	int	Integer subtraction.
–	pixel	pixel	pixel	Vector subtraction.
+	image	image	image	Union of images LEFT and RIGHT.
+	int	int	int	Integer addition.
+	pixel	pixel	pixel	Vector addition.
+	image	pixel	image	Add a pixel to an image (set value).
+	pixel	image	image	Add a pixel to an image (set value).
*	image	image	image	Intersection of LEFT and RIGHT.
*	int	int	int	Integer multiplication.
<<	image	string	image	Read an image from a PBM format file named by the string.
<<	int	string	int	Read an integer.
<<	pixel	string	pixel	Read a pixel (2 ints).
>>	image	string	image	Write an image to a PBM format file named by the string.
>>	int	string	int	Write an integer.
>>	pixel	string	pixel	Write a pixel (2 ints).
–>	image	pixel	image	Translate the image by the pixel.

OPERATOR	LEFT	RIGHT	RESULT	DESCRIPTION
<-	pixel	image	image	Translate the image by the pixel.
@	pixel	image	int	Membership: Is the pixel in the image?
[]	[int, int]			Pixel generator. Result is the pixel whose row is the first int and whose column is the second int.
.	image.rows image.cols image .origin_x image .origin_y			Each image has 4 attributes: number of rows and columns, and the row and column locations of the origin. These can be accessed by imagename.attrname.
.	pixel.row pixel.col			Each pixel has two attributes: the row and column indices.
UNARY OPERATORS				
~				Set complement.
!				Allocate a new image like the given one.
#				The (integer) number of isolated pixels in an image.
-				Integer negation.
IMAGE GENERATOR				
{PIXEL1, PIXEL2, "0110101 . . . "}				Generates an image, whose size is given by PIXEL1, whose origin is given by PIXEL2, and whose pixels are specified by the string.

Parentheses can be used in expressions to specify the order of evaluation. There is no precedence implicit in the operators other than that unary operators will be computed before binary ones, and evaluation is otherwise left to right.

There is a collection of test programs named `t1.max` through `t30.max`; any MAX program must have the `.max` suffix. Running the compiler is quite simple. It used to run from the command line, but it is rare these days to run from the

command prompt. The compiler now asks for the source file name from an input prompt:

```
MAX
MAX compiler: enter source code file name (ends in .max): t1.max
```

This will compile the program `t1.max` into C code file called `t1.c`. On most systems this will be automatically compiled into the object file `t1.exe` and then executed. In order for the C compilation to be successful, the files `max.h` and `maxlib.c` must be in the same working directory.

As a tool for education and as a test bed for morphological experimentation and testing of structuring elements, MAX is still unique (if perhaps not perfect). For example, the program `BinDil` can now be written:

```
// Dilation using ++
image a, b;
begin
              do (a << ~$1~)++(b<<~$2~) >> ~$3~
end;
```

Figure 3.17 shows a MAX program for doing a dilation the hard way: by translating the structuring element to all pixel positions in the image being dilated and accumulating the union of all these images (sets). From this point forward, programs illustrating morphological operations will be written in MAX.

3.3.7 The "Hit-and-Miss" Transform

The hit-and-miss transform is a morphological operator designed to locate simple shapes within an image. It is based on erosion; this is natural, because the erosion of A by S consists only of those pixels (locations) where S is contained within A, or matches the set pixels in a small region of A. However, it also includes places where the background pixels in that region do not match those of S, and these locations would not normally be thought of as a match. What we need is an operation that matches both the foreground and the background pixels of S in A.

Matching the foreground pixels in S against those in A is a ''hit'', and is accomplished with a simple erosion $A \ominus S$. The background pixels in A are those found in A^c, and while we could use S^c as the background for S a more flexible approach is to specify the background pixels explicitly in a new structuring element T. A ''hit'' in the background is called a ''miss,'' and is found by $A^c \ominus T$. We want the locations having both a ''hit'' and a ''miss,'' which are the pixels:

$$A \otimes (S, T) = (A \ominus S) \cap (A^c \ominus T) \qquad \text{(EQ 3.22)}$$

```
// MAX Program to perform a dilation the hard way.
//
int i,j;
image x, y, z;
begin
        i := 0; j := 0;
        y := !(x<<"$1");                // Allocate a result image like x.
        do z<<"$2";                     // Read the structuring element.

        loop                            // For all indices i
        j := 0;
                loop                    // For all indices j
                        if ([i,j] @ x) then // Is pixel i,j in the image?
                                        // Translate structuring element
                                               by i,j
                                        // and OR the result (union)
                                                with y
                                y := y + (z->[i,j]);
                j := j + 1;                 // Next j
                exit when j >= x.cols; // j exceeds maximum column?
        end;
        i := i + 1;                         // Next i
       exit when i >= y.rows;                        // i exceeds max row?
 end;

 do y>>"$3";                                        // Output the result.
end;
```

Figure 3.17: MAX program to compute a dilation by repeated translations and unions.

As an example, let's use this transform to detect upper-right corners. Figure 3.18a shows an image that could be interpreted as being two overlapping squares. A corner will be a right angle consisting of the corner pixel and the ones immediately below and to the left, as shown in Figure 3.18b. The figure also shows the ''hit'' portion of the operation (c), the complement of the image (d), and the structuring element used to model the background (e), the ''miss'' portion (f), and the result of the intersection of the ''hit'' and the ''miss'' (g). The set pixels in the result both correspond to corners in the image.

Also notice that the background-structuring element is *not* the complement of the foreground-structuring element; indeed, if it had been then the result would have been an empty image because there is no match to its peculiar shape in the complement image. The set pixels in the background-structuring element are those that *must* be background pixels in the image in order for a match to take place. Over-specification of these pixels results in few matches, and under-specification results in too many. Careful selection, possibly through experimentation, is needed.

By the way, the upper and right pixels in Figure 3.18f are white because they correspond to locations where the structuring element 3.17e has black pixels

placed outside of the bounds of the image. The complement operator produces an image of the same size as the image being complemented, although when using sets this would not be so. This problem can be avoided by copying the input image to a bigger image before doing the complement.

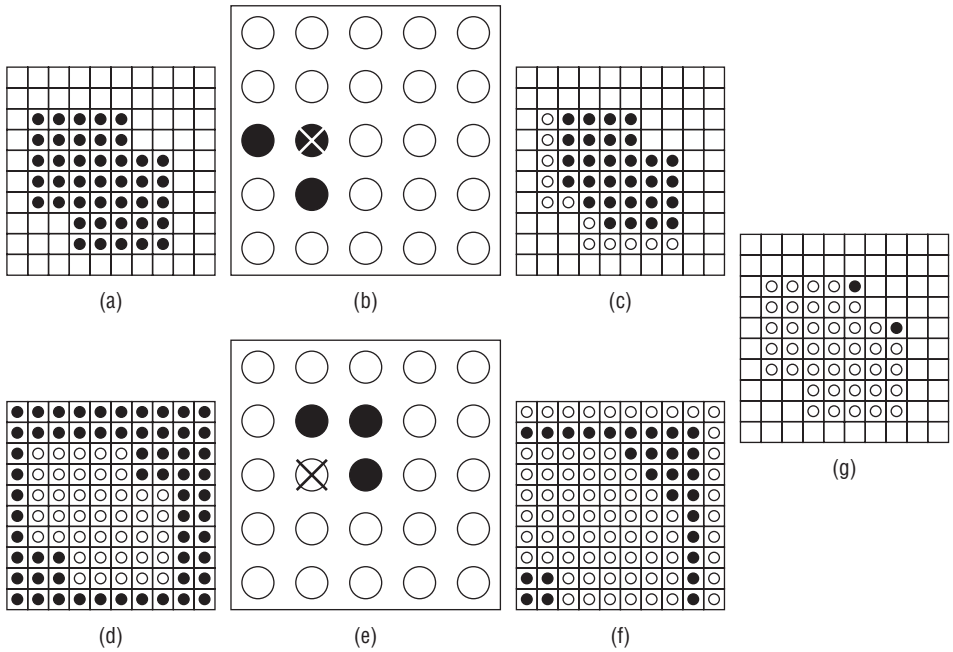


Figure 3.18: Illustration of the hit-and-miss transform. (a) The image to be examined. (b) Foreground-structuring element for the location of upper-right corners. (c) The erosion of (a) by (b) — the "hit" portion of the computation. (d) The complement of (a). (e) The background-structuring element, showing that the three pixels to the upper right of the corner must be background pixels. (f) The erosion of (d) by (e), or the "miss" portion of the computation. (g) The intersection of (c) and (f) — the result, showing the location of each of the two upper-right corners in the original image.

The MAX program that performs a hit-and-miss transform is:

```
// Hit-and-miss transform
image a, se1, se2;
begin
       do a << "$1";
       se1 := {[5,5], [2,1], "00000000001100001000000000"};
       se2 := {[5,5], [2,1], "00000011000010000000000000"};

       a := (a--se1)*(~a -- se2);
       message a;
end;
```

3.3.8 Identifying Region Boundaries

The pixels on the boundary of an object are those that have at least one neighbor that belongs to the background. Because *any* background neighbor is involved it cannot be known in advance which neighbor to look for, and a single structuring element that would allow an erosion or dilation to detect the boundary can't be constructed. This is in spite of the fact that an erosion by the simple structuring element removes exactly these pixels!

On the other hand, this fact can be used to design a morphological boundary detector. The boundary can be stripped away using an erosion and the eroded image can then be subtracted from the original. This should leave only those pixels that were eroded — that is, the boundary.

A MAX program for this is:

```
// Boundary extraction
image a, b, c;
begin
       do a << "$1";
       b := {[3,3], [1,1], "111111111"};   // Simple structuring element
       c := (a - (a--b));
       message c;
       do c >> "boundary.pbm";
end;
```

Figure 3.19 shows this method used to extract the boundaries of the ''squares'' image of Figure 3.18a. A larger example, that of a quarter rest scanned from a page of sheet music, also appears in the figure.

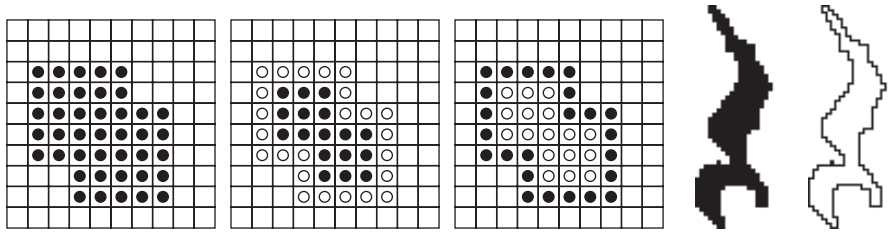


Figure 3.19: Morphological boundary extraction. (a) The squares image. (b) The squares image after an erosion by the simple structuring element. (c) Difference between the squares image and the eroded image: the boundary. (d) A musical quarter rest, scanned from a document. (e) The boundary of the quarter rest as found by this algorithm.

3.3.9 Conditional Dilation

There are occasions when it is desirable to dilate an object in such a way that certain pixels remain immune. If, for example, an object cannot occupy certain

parts of an image then a dilation of the object must not intrude into that area. In that case, a *conditional dilation* can be performed. The forbidden area of the image is specified as a second image in which the forbidden pixels are black (1). The notation for conditional dilation will be:

$$A \oplus (S_e, A') \qquad \text{(EQ 3.23)}$$

where S_e is the structuring element to be used in the dilation, and A' is the image representing the set of forbidden pixels.

One place where this is useful is in segmenting an image. Determining a good threshold for grey-level segmentation can be difficult, as discussed later in Chapter 3. However, sometimes two bad thresholds can be used instead of one good one. If a very high threshold is applied to an image, only those pixels that have a high likelihood of belonging to an object will remain. Of course, a great many will be missed. Now a very low threshold can be applied to the original image, giving an image that has too many object pixels, but where the background is marked with some certainty. Then the following conditional dilation is performed:

$$R = I_{high} \oplus (\text{simple}, I_{low}) \qquad \text{(EQ 3.24)}$$

The image R is now a segmented version of the original, and it is in some cases a superior result than could be achieved using any single threshold (Figure 3.20).

The conditional dilation is computed in an iterative fashion. Using the notation of Equation 3.23, let $A_0 = A$. Each step of the dilation is computed by:

$$A_i = (A_{i-1} \oplus S_e) \cap A' \qquad \text{(EQ 3.25)}$$

The process continues until $A_i = A_{i-1}$, at which point A_i is the desired dilation. A MAX program for conditional dilation is:

```
// Conditional dilation
image a, b, c, d;
begin
       do a << ^$1";                          // Input image.
       do c << ^$2";                          // Forbidden image.
       b := {[3,3], [1,1], ^111111111"};      // Simple structuring element.

       loop
        d := (a ++ b) * c;
        exit when d=a;
        a := d;
       end;
       do a >> ^$3";
end;
```

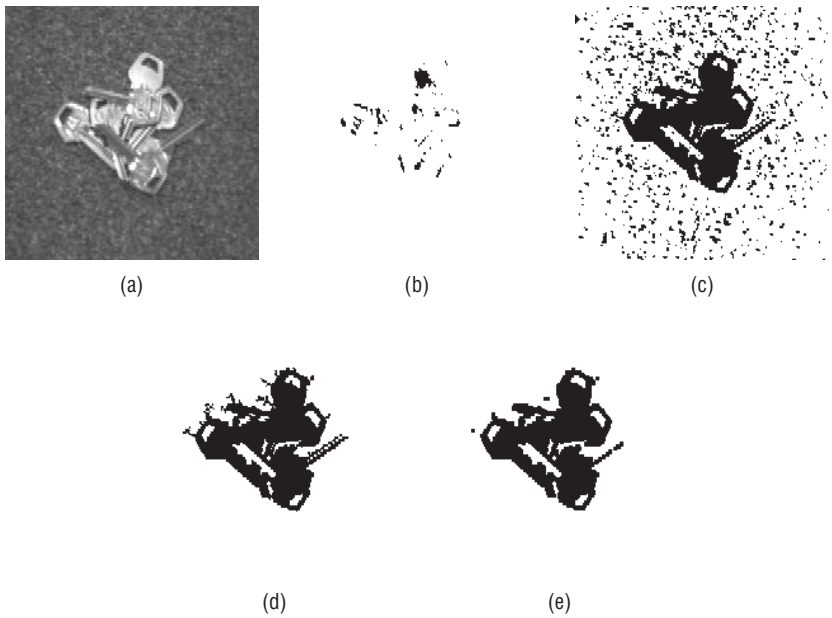


Figure 3.20: Conditional dilation. (a) Image of a pile of keys. (b) Negative image resulting from use of a high threshold. (c) Result of using a low threshold. (d) Conditional dilation of (b) using the simple structuring element, conditional on (c). (e) The result after being cleaned up — in this case, by using an opening.

Another application of conditional dilation is that of filling a region with pixels, which is the inverse operation of boundary extraction. Given an outline of black pixels and a pixel inside of the outline, we are to fill the region with black pixels. In this case, the forbidden image will consist of the boundary pixels; we want to fill the region up to the boundary, but never set a pixel that is outside. Because the outside pixels and the inside pixels have the same value, the boundary pixels are forbidden and the dilation continues until the inside region is all black. Then this image and the boundary image are combined to form the final result.

The conditional dilation is:

$$\text{Fill} = P \oplus (S_{cross}, A^c) \qquad \text{(EQ 3.26)}$$

where P is an image containing only the seed pixel, which is any pixel known to be inside the region to be filled, and A is the boundary image for the region to be filled. S_{cross} is the cross-shaped structuring element seen in Figure 3.21b. The same figure shows the steps in the conditional dilation that fills the same boundary that was identified in section 3.2.8. The seed pixel used in the example is [3,3], but any of the white pixels inside the boundary could have been used.

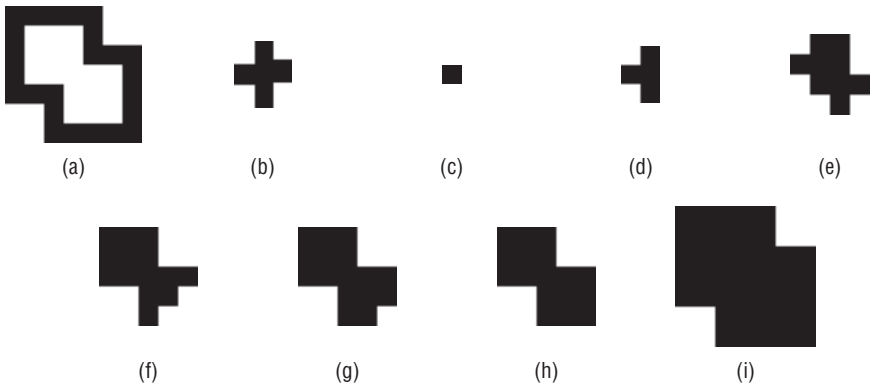


Figure 3.21: Filling a region using conditional dilation. (a) The boundary of the region to be filled. This is the boundary found in Figure 3.19. (b) The structuring element. (c) The seed pixel, and iteration 0 of the process. (d) After iteration 1. (e) After iteration 2. (f) After iteration 3. (g) After iteration 4. (h) After iteration 5 the dilation is complete. (i) Union of (h) with (a) is the result.

A MAX program for region filling requires the input of the coordinates of the seed pixel, which is the first time that integer input has been performed in a MAX program. It is:

```
// Fill a region with 1 pixels - Conditional Dilation
pixel p;
int i,j;
image a, b, c, d;
begin
       do a << "$1";
       message "FILL: Enter the coordinates of the seed pixel ";
       do i << "";       do j << "";
       p := [i, j];       // SEED pixel.
       b := {[3,3], [1,1], "010111010"};
       c := !a + p;
       a := ~a;
       loop
              d := (c ++ b) * a;
              exit when d=c;
              c := d;
       end;
       do c + ~a >> "$2";
end;
```

3.3.10 Counting Regions

As a final example of the uses of morphology in binary images, it is possible to count the number of regions in an image using morphological operators. This

method, first discussed by Levialdi, uses six different structuring elements. The first four elements are used to erode the image, and were carefully chosen so as not to change the connectivity of the regions being eroded. The last two elements are used to count isolated ''1'' pixels; in MAX this is done using the # operator, and so these structuring elements will not be needed.

Figure 3.22 shows the four structuring elements, named L1 through L4. The initial count of regions is the number of isolated pixels in the input image A, and the image of iteration 0 is A:

$$\begin{aligned} \text{count}_0 &= \#A \\ A_0 &= A \end{aligned} \qquad \text{(EQ 3.27)}$$

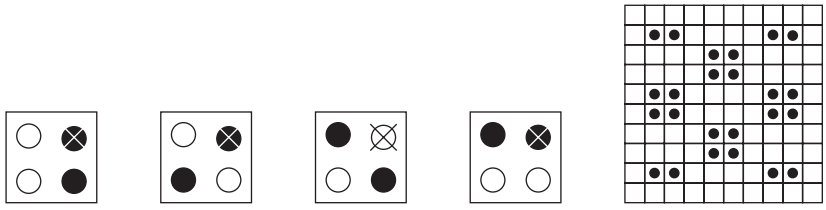


Figure 3.22: Counting 8-connected regions. (a)–(d) The structuring elements L1 through L4. (e) An example image having eight regions. The algorithm counts these correctly.

The image of the next iteration is the union of the four erosions of the current image:

$$A_{n+1} = (A_n \ominus L_1) \cup (A_n \ominus L_2) \cup (A_n \ominus L_3) \cup (A_n \ominus L_4) \qquad \text{(EQ 3.28)}$$

And the count for that iteration is the number of isolated pixels in that image:

$$\text{count}_{n+1} = \#A_{n+1} \qquad \text{(EQ 3.29)}$$

The iteration stops when A_n becomes empty (all 0 pixels). The overall number of regions is the sum of all the values count_i. The MAX program that does this is:

```
// Count 8-connected regions.
image L1, L2, L3, L4, a, b;
int count;
begin
       L1 := {[2,2], [0,1], `0101"};
       L2 := {[2,2], [0,1], `0110"};
       L3 := {[2,2], [0,1], `1001"};
       L4 := {[2,2], [0,1], `1100"};
       do a << "$1";
```

```
        count := 0;
        loop
         count := #a + count;
         b := (a--L1) + (a--L2) + (a--L3) + (a--L4);
         exit when b = 0;
         a := b;
        end;
        message `Number of 8 regions is ´; message count; message;
end;
```

This program counts eight regions in Figure 3.22e, which is correct for 8-connected regions. It also counts two regions in Figure 3.11a, which is also correct. The algorithm for 4-connected regions is the same but uses different structuring elements.

3.4 Grey-Level Morphology

The use of multiple grey levels introduces an enormous complication, both conceptually and computationally. A pixel can now have any integer value, so the nice picture of an image being a set disappears. There is also some question about what dilation, for example, should *mean* for a grey-level image. Rather than being strictly mathematical here, we will take a more intuitive approach, in the hope that the result will make sense.

Consider the image of a line in Figure 3.23a.

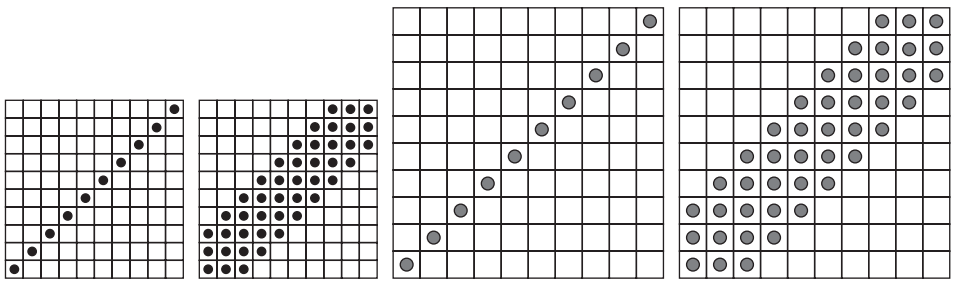


Figure 3.23: Grey-scale dilation. (a) A bi-level image of a line. (b) Binary dilation of (a) by simple. (c) A grey-scale image of a line; background is 0, and the line pixels have the value 20. (d) This is what the grey line should look like after a dilation.

This is a bi-level image, and the dilation of this image by the simple structuring element can be computed (Figure 3.23b). Now imagine that instead of having levels 0 and 1, the pixels in the line have the value 20 and the background is 0. What should a dilation of this new image by simple look

like? The binary dilation spreads out the line, as determined by the locations of the ''1'' pixels, making it three pixels wide instead of only one. The grey-level image should have a corresponding appearance after dilation, where the difference between the foreground and background pixels should be about the same as in the original and the line should be about three pixels wide. An example of how the dilated grey-level line (Figure 3.23c) might appear is given in Figure 3.23d.

This appears to be a reasonable analogue of dilation for the grey-level case, at least for a simple image. The image in Figure 3.23d was computed from Figure 3.23c as follows:

$$(A \oplus S)[i,j] = \max\{A[i-r, j-c] + S[r,c] \big| [i-r, j-c] \in A, [r,c] \in S\} \quad \text{(EQ 3.30)}$$

where S is the simple structuring element and A is the grey-level image to be dilated. This is one definition of a grey-scale dilation, and it can be computed as follows:

1. Position the origin of the structuring element over the first pixel of the image being dilated.
2. Compute the sum of each corresponding pair of pixel values in the structuring element and the image.
3. Find the maximum value of all these sums, and set the corresponding pixel in the output image to this value.
4. Repeat this process for each pixel in the image being dilated.

The values of the pixels in the structuring element are grey levels as well, and can be negative. Because negative-valued pixels can't be displayed there are two possible ways to deal with negative pixels in the result: they could be set to 0 (underflow), or the entire image could have its levels shifted so that the smallest became 0 and the rest had the same values relative to each other as they did before. We choose the former approach for simplicity.

Given the definition of dilation in Equation 3.30, a possible definition for grey-scale erosion would be:

$$(A \ominus S)[i,j] = \min\{A[i-r, j-c] - S[r,c] \big| [i-r, j-c] \in A, [r,c] \in S\} \quad \text{(EQ 3.31)}$$

This definition of erosion works because it permits the same duality between erosion and dilation that was proved in section 3.2.3.

Figure 3.24 shows an application of grey-scale erosion and dilation to the image of keys first seen in Figure 3.20. The structuring element was simple, made into grey levels. While it is not immediately clear why this operation is useful, the parallel with binary dilation and erosion is plain enough. Note,

for instance, that dilation makes the small hole in the top of each key smaller, whereas the erosion made them larger.

(a) (b) (c)

Figure 3.24: Grey-scale erosion and dilation. (a) Original. (b) Dilated by simple. (c) Eroded by simple.

3.4.1 Opening and Closing

Opening and closing a grey-scale image is done in the same way as before, except that the grey-scale erosion and dilation operators are used — that is, an opening is a grey-scale erosion followed by a grey-scale dilation using the same structuring element, and a closing is vice versa. However, intuitively it is more useful to use a geometric model to see what is happening.

Consider a grey-level image to be a three-dimensional surface, where the horizontal (x) and vertical (y) axes are as before, and the depth (z) axis is given by the grey value of the pixel. A structuring element will also be a grey-level image, and in particular consider one that is spherical such as:

0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 2 1 1 1 0 0
0 1 1 2 2 3 2 2 1 1 0
0 1 2 3 3 4 3 3 2 1 0
0 1 2 3 5 5 5 3 2 1 0
0 2 3 4 5 6 5 4 3 2 0
0 1 2 3 5 5 5 3 2 1 0
0 1 2 3 3 4 3 3 2 1 0
0 1 1 2 2 3 2 2 1 1 0
0 0 1 1 1 2 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0

Of course, this is only half of a sphere and is only approximately spherical because of truncation error and sampling. Nonetheless, imagine this sphere being rolled over the underside of the surface represented by the image being opened. Whenever the center of the sphere is directly beneath an image pixel, the value of the opened image at that point is the highest (maximum) point achieved by any part of the sphere. The closing would be modeled by rolling the structuring element over the top of the surface and taking the lowest point on the sphere at all pixels as the value of the corresponding pixel in the closed image.

Figure 3.25 shows this process in two dimensions, as if viewing a cross section of the image. An opening, in this case, can be seen as a smoothing process that *decreases* the average level of the pixels, whereas closing appears to *increase* the levels.

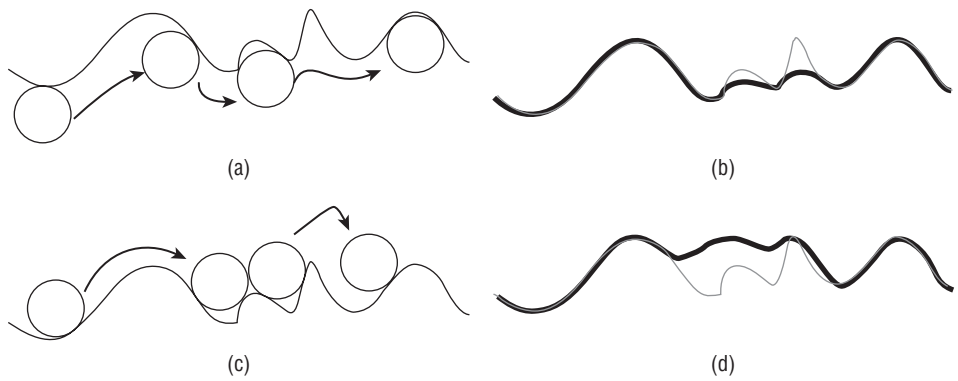
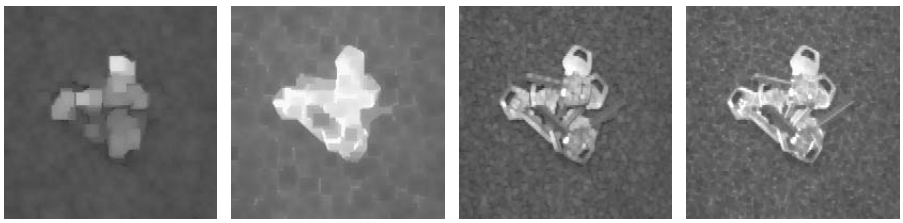


Figure 3.25: Geometric interpretation of grey-level opening and closing. (a) A "slice" through the image being opened, showing four positions of the structuring element. (b) The opened slice — the highest points of the circle at all pixels. (c) Rolling the circle over the top of the slice. (d) The closed slice — the lowest points of the circle at all pixels. These figures are approximations.

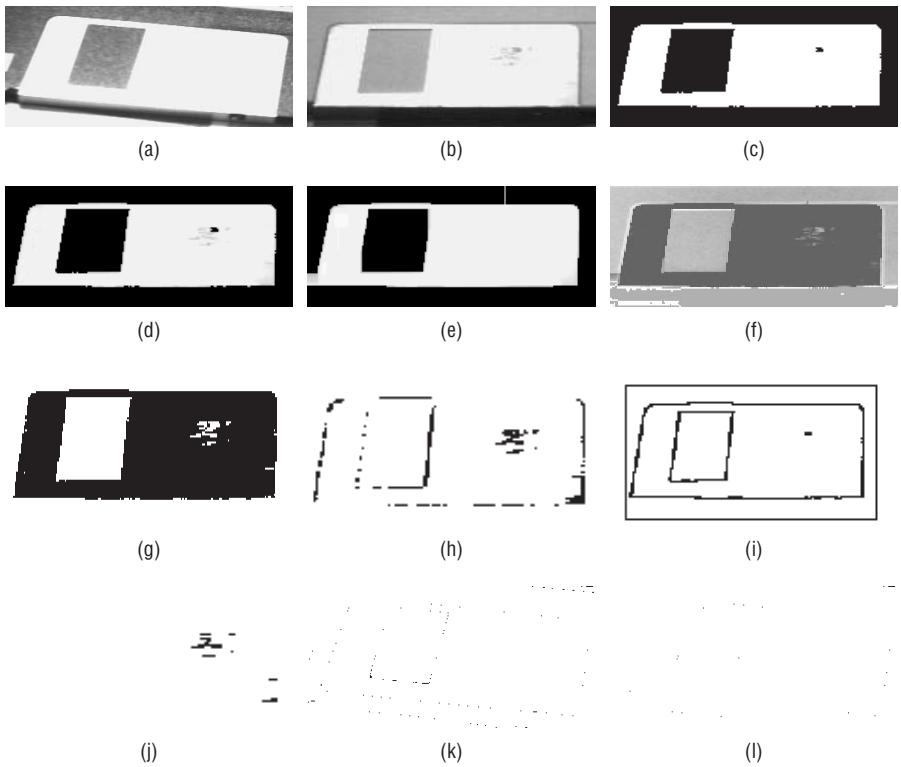
Figure 3.26 shows grey-level opening and closing applied to the keys image of Figure 3.24.

One interesting application of opening and closing is in the visual inspection of objects. For example, when an object is cut or polished there can be scratches left in the material. These become more visible if light is reflected off of the surface at a low angle and the object is seen from the side opposite the lighting source. Figure 3.27 shows an example of this, using a pair of disk guards from 3½ inch floppy disks. The guard on the right (3.27b) is scored, which can be seen clearly in the image.



(a) (b) (c) (d)

Figure 3.26: Examples of grey-level opening and closing. (a) Opened "keys" image, using spherical structuring element. (b) Closed "keys" image using spherical structuring element. (c) Opened using simple structuring element. (d) Closed using simple structuring element.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 3.27: Use of grey-level morphology for the inspection of disk guards. (a) An acceptable disk guard. (b) A scored disk guard. (c) Image (b) after thresholding. (d) Thresholded image used as a mask to give an image showing mainly the guard. (e) Image (d) closed using a circle (rod). (f) Residual of images (b) and (d). (g) Thresholded version of the residual. (h) Image obtained by ANDing image (c) and image (g), showing some edges plus the defect in the guard. (i) Boundary of the thresholded guard image (c). (j) All pixels in ANDed image (j) that are near a boundary have been deleted. Remaining black pixels are potential defects. The same process applied to the "good" guard image a showing (k) the ANDed image and (l) the image in (k) after removing edge pixels. No defects are reported here.

The first step in the inspection process is to gain a good estimate of the location of the object being inspected. This is done by thresholding the image and then using the thresholded image as a mask: into an all-black image, copy those pixels from the original that correspond to a white pixel in the thresholded image (3.26d). The masked image is then closed using a circular structuring element with grey values. The result will be to raise the grey levels of the pixels in the defects to near that of the surrounding pixels, giving a clear image of the guard in the sampled orientation. When the original is subtracted from this image, the defect will stand out in contrast to the guard (3.26f)

Thresholding will increase the contrast further, and the pixels that are white in both this image and the original thresholded image are of special interest (3.26h). Unfortunately, some of the pixels near the edges of the object(s) have been blurred a little, and so any pixels near the original boundary (as found by a morphological operation as well) are deleted, giving an image showing only potential defects as black pixels. As confirmation of this process, the image having no defects was also processed in the same way, and it shows no black pixels in the final image.

Many kinds of inspection tasks can be carried out in a similar way, including the inspection of paper for dirt, glass for bubbles, and wood and plastic for defects.

3.4.2 Smoothing

One possible smoothing operation involves a grey-level opening followed by a closing. This will remove excessively bright and excessively dark pixels from the image; such pixels can be the result of a noise process, but unfortunately might also be legitimate data values. The price to be paid for noise reduction is a general blurring of the image.

Figure 3.28a shows an image of a disk guard that has been subjected to Gaussian (normal distribution) noise with a standard deviation of 30. Figure 3.28c shows the result of morphological smoothing applied to this image; initially, it is not clear which image is to be preferred. However, the same two images after thresholding (Figures 3.28b and 3.28d) clearly demonstrate that the smoothing process eliminates much of the problem noise, which now shows up as a ''salt and pepper'' effect, and which would certainly create problems in later processing.

The structuring element used to smooth the disk guard image was simple, but the choice would depend on the type of noise being cleaned up. One common problem is the appearance of *scan lines* in an image that was obtained by photographing a television of video screen. Figure 3.29 shows an example of this sort of structured noise. The structuring element was constructed by looking at the original noisy image in detail; the distance between scan lines was about nine pixels, and the grey values in the structuring element

were chosen to be the differences between the grey-level at the darkest point of the scan line and that of each of the following eight pixels in each column.

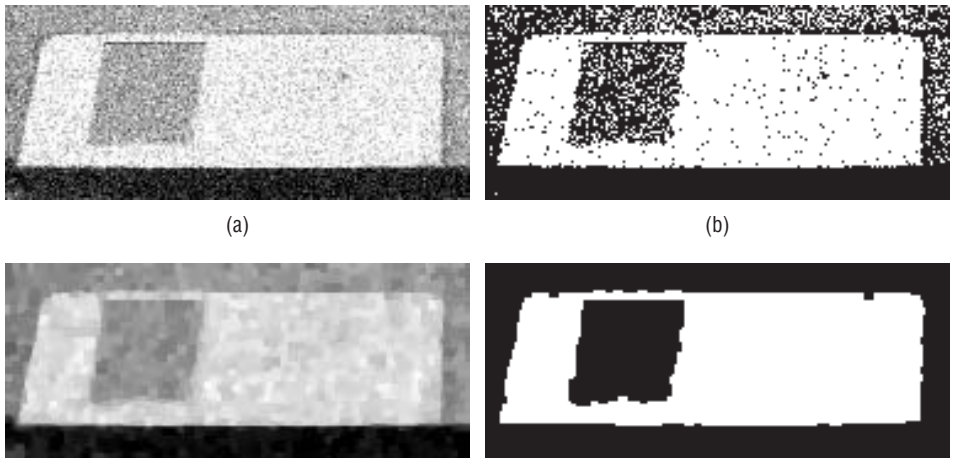


Figure 3.28: Grey-level smoothing. (a) Disk guard image subjected to Gaussian noise having a standard deviation of 30. (b) Thresholded version of a showing salt and pepper effect of thresholding the noise. (c) Image (a) after morphological smoothing. (d) Smoothed image after thresholding, showing less noise in the thresholded image.

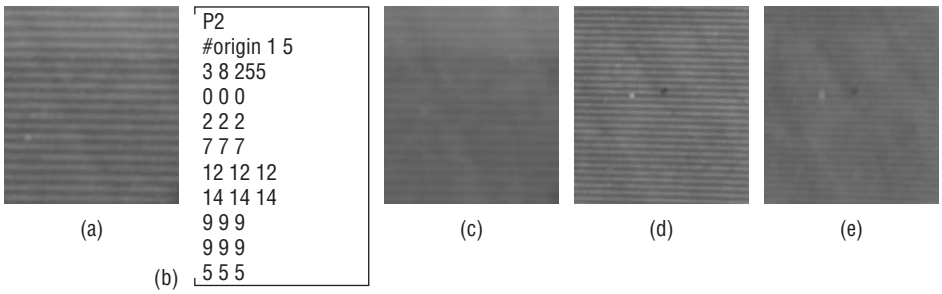


Figure 3.29: Morphological smoothing used to reduced structured noise. (a) Small image section showing scan lines. (b) Structuring element used to reduce the scan lines. (c) Image (a) after smoothing. (d) A second example with scan lines. (e) Image (d) after smoothing with the element in (b).

The result is surprisingly good. Figure 3.29c is the smoothed version of 3.29a, and while it is a little blurry the scan line noise has been significantly reduced; the same can be said for Figures 3.29d and 3.29e, which are the before and after versions of another sample taken from the same larger image.

3.4.3 Gradient

In Section 3.2.8 a method for identifying the boundaries of a bi-level object was discussed. The basic idea was to erode an image using the simple structuring element and then subtract the result from the original, leaving only the pixels that were eroded. This can be done with grey-level images too. The boundary detection operator can be expressed in the same manner as in Equation 3.14, and results in an operation not unlike *unsharp masking*, in which an average of a small (say, 3x3) region is subtracted from the original pixel at the center of the region. This procedure has an analog in photography.

Because the contrast is not as great in a grey-level image as in a bi-level one the results of the boundary detection are not as good. However, an improvement can be achieved by using the formula:

$$G = (A \oplus S) - (A \ominus S) \qquad \text{(EQ 3.32)}$$

where S is a structuring element. Instead of subtracting the eroded image from the original, Equation 3.32 subtracts it from a dilated image. This increases the contrast and width of the extracted edges. Equation 3.32 is the definition of the morphological gradient, which detects edges in a manner that is less dependant on direction than is the usual gradient operator. Figure 3.30 shows both the boundary detection algorithm and the morphological gradient applied to the disk guard image. In all cases, the simple structuring element was used.

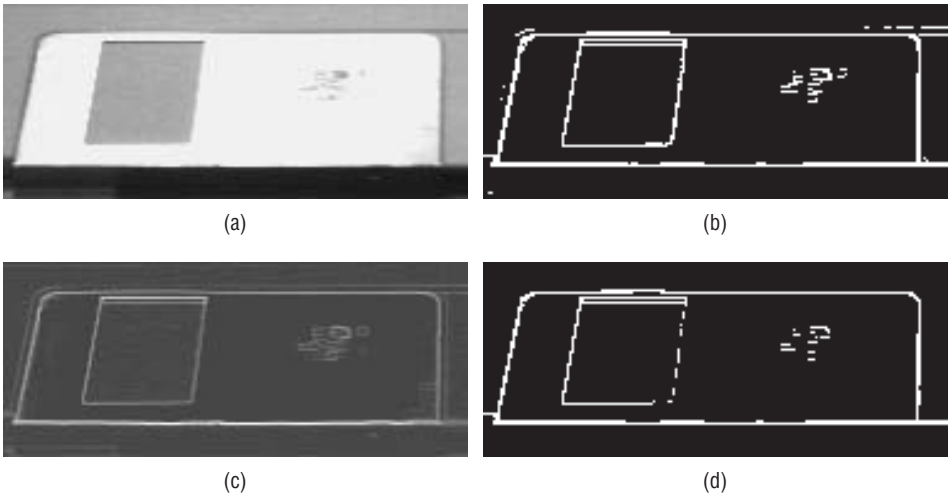


Figure 3.30: Morphological gradient. (a) Disk guard image. (b) Edges enhanced by the grey-level boundary extraction method applied to grey-level images, then thresholded. (c) The morphological gradient. (d) Image (c) after thresholding.

3.4.4 Segmentation of Textures

Closing removes dark detail, and opening joins dark regions. This suggests an application to textures, and the identification of regions in a image based on the textural pattern seen there. While this subject will be explored more fully in Chapter 5, a simple example at this point will probably not detract from later revelations.

If, for instance, one texture consists of small dark blots and another consists of larger dark blots then closing by the size of the small blots will effectively remove them, but will leave some remnant of the larger ones. Now an opening by the size of the gaps between the large blots will join them into one large dark area. The boundary between the two regions should now be easy to identify.

An example of this can be seen in Figure 3.31. The original image has two regions filled with different textures; this image was created by a drawing package, so the textures repeat exactly, but this is not a requirement. Closing removes all traces of the smaller texture, and closing creates a solid black region where the larger texture had appeared. The morphological boundary extraction procedure can then be applied, giving a solid line along the margin between the two textures. The line is jagged wherever the boundary cuts a large blot in two, creating a small one.

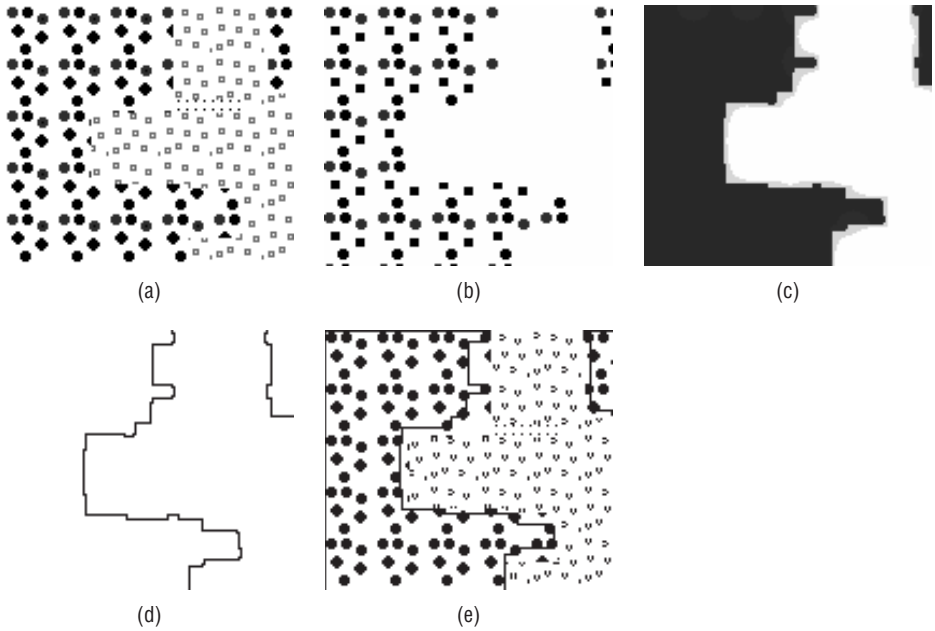


Figure 3.31: Texture segmentation. (a) The image to be segmented. (b) After closing by the size of the small blots. (c) After further closing by the size of the spaces between the large texture blots. (d) The boundary seen in (c). (e) The boundary superimposed over the original texture image.

This method can be applied to a variety of textures, although some experimentation with structuring elements may be needed to achieve good results.

3.4.5 Size Distribution of Objects

The use of morphology for segmenting regions by texture suggests another application: the classification of objects by their size or shape. Because the use of shape would require quite a bit of experimenting with different structuring elements, size classification will be explored here. Quite a variety of objects are regularly classified according to their size, from biological objects under a microscope to eggs and apples. A ''grade A large'' egg, for example, should be noticeably bigger than a ''medium'' egg, and it should be possible to create a program for classifying eggs using grey-level morphology. However, because eggs are often graded using their weight, we will examine another case close to all of us — that of money.

As it happens, and not by accident, coins vary in size according to their value. A dime is the smallest, and a one-dollar coin is, if you can find one, the biggest. Figure 3.32a shows an image of a small collection of coins on a dark background. It is a mixture of U.S. and Canadian coins, since it was easy to obtain a Canadian one-dollar coin (called a *loon*), but a U.S dollar coin would have worked as well.

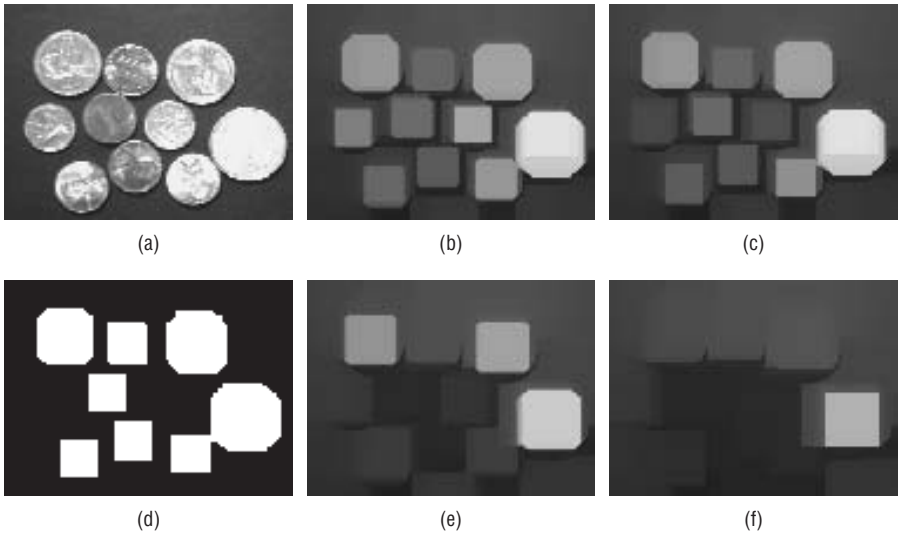


Figure 3.32: Classifying coins by their size. (a) The image containing coins to be classified. (b) After opening by a structuring element of radius 6. (c) After opening by radius 6.5. (d) Thresholded version of (c), showing that the dimes can be removed. (e) After opening by radius 8, showing that the pennies have been removed. (f) After opening by radius 10; the only coin remaining is the one-dollar coin.

Because a grey-level opening will decrease the level of an object, the image was opened with circular structuring elements of gradually increasing radius. At some point, when the radius of the structuring element exceeds that of the coin, the coin will be removed from the image. The radii actually used were those from 5 to 14; opening by a circular structuring element of radius 14 actually removes all coins, leaving a dark and empty image.

The first change is at radius 6.5 (diameter 13), where the dimes are reduced in level sufficiently that thresholding can delete them. An opening using a radius of 8 removes the pennies, allowing them to be counted. Finally, an opening using a radius of 10 removes the quarters, leaving only the loon. By counting the regions that vanish after each iteration, it should be possible to accumulate the total value of the coins in the image. In many countries the paper money also varies in size, allowing bills to be classified by size, as well.

3.5 Color Morphology

Color can be used in two ways. As before, we can assume that the existence of three color components (red, green, and blue) is an extension of the idea of a grey level, or each color can be thought of as a separate domain containing new information. In either case, morphology is not commonly applied to color images, possibly because the construction of the structuring elements necessary to perform a particular task is really quite complex. Color morphology will only be touched on here through the use of a single example.

Figure 3.33a is a grey-level version of a color image, which shows an insect sitting on a leaf. Both the insect and the background are basically green, so automatically locating the insect could be a little tricky. On close examination of each of the three color basis images (red, green, and blue) it is observed that there are slight variations in each component: the insect seems to be brighter in the red and blue images, whereas the background is brighter in the green image.

Closing the red and blue images should brighten the insect further, and opening the green image should suppress the background a little. A circular structuring element with a radius of four was used in each case. Following the closings and opening the three component images were recombined to form a single color image. The insect is now a bright pinkish color, and can be seen in sharp contrast to the darker green background. Figure 3.33e is a grey version of this color image, but the insect is still clearly present. Using this image as a mask of the original will give an isolated picture of the insect, or at least most of it, as seen in Figure 3.33f.

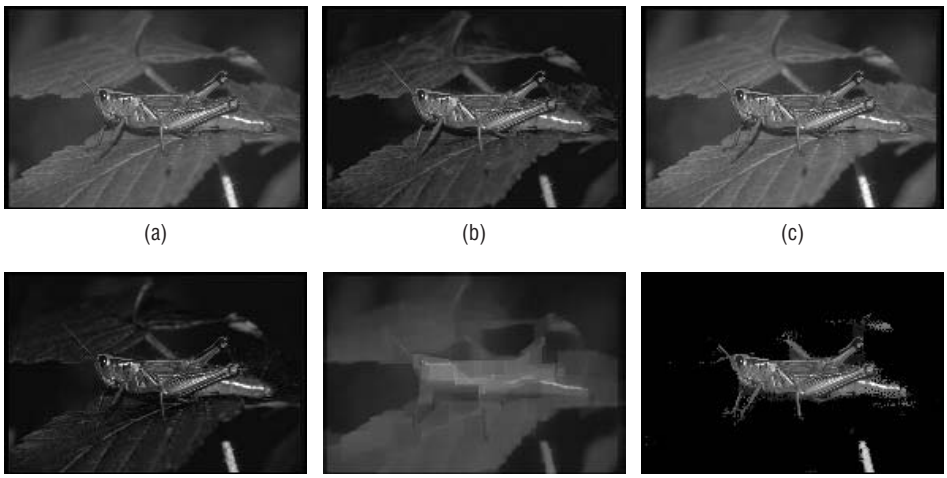


Figure 3.33: Color morphology. (a) Image of a grasshopper. (b) RED component of the RGB color image. (c) GREEN component. (d) BLUE component. (e) Image resulting from closing the red and blue components and opening the green part. (f) Original image masked with the processed one, showing the insect.

3.6 Website Files

`bindil.exe`	Binary dilation
`binerode.exe`	Binary erosion
`maxcompiler.exe`	MAX language compiler, command line
`maxcompilerb.exe`	MAX language compiler, interactive
`maxg.exe`	MAX grey-level compiler
`bindil.c`	C source code, binary dilation
`binerode.c`	C source code, binary erosion
`grey.c`	Library for grey-level MAX
`max.c`	Source code, command line MAX
`maxb.c`	Source code, interactive MAX
`maxg.c`	Source code, grey-level MAX
`maxlib.c`	Library, command line MAX

maxlibb.c	Library, interactive MAX
mlib.c	Morphology library
max.h	MAX include file
morph.h	C morphology library include file.
circle4.pgm	Image of a circle, radius 4
circle5.pgm	Image of a circle, radius 5
circle6.pgm	Image of a circle, radius 6
circle7.pgm	Image of a circle, radius 7
circle8.pgm	Image of a circle, radius 8
circle9.pgm	Image of a circle, radius 9
circle10.pgm	Image of a circle, radius 10
circle11.pgm	Image of a circle, radius 11
circle12.pgm	Image of a circle, radius 12
circle13.pgm	Image of a circle, radius 13
circle14.pgm	Image of a circle, radius 14
coin3.pgm	Coin image, bright
coin5.pgm	Coin image, dark
disk1.pgm	Disk guard image
disk2.pgm	Disk guard image, scratched
keys.pgm	Image of keys on textured background
knight.pgm	Image of a chess piece
noisy.pgm	Disk guard image, with noise
rod.pgm	Small radius grey structuring element
scans.pgm	Image showing raster scan lines
simple.pgm	3x3 grey structuring element
texture.pgm	two textures, grey level (Figure 3.31)
and.max	MAX program, AND operation
bindil.max	MAX program, binary dilation

binerode.max	MAX program, binary erosion
boundary.max	MAX program, boundary extraction.
close.max	MAX program, closing
dil.max	MAX program dilation
dilg.max	MAX program, grey-level dilation
dual.max	MAX demo: erosion/dilation duality
erog.max	MAX, grey-level erosion
fill.max	MAX: conditional dilation to do a fill
gclose.max	MAX, grey-level close
gopen.max	MAX, grey-level open
gradient.max	MAX, gradient edge detector
mitmiss.max	Hit-and-miss transform, (exercise 5.1 from Haralick and Shapiro)
hitmiss2.max	Hit-and-miss transform (exercise 5.11 from Haralick and Shapiro)
iotest.max	MAX, test of input/output
open.max	MAX, opening
smooth.max	MAX, smoothing
t1.max t15.max	MAX test functions.
tophat.max	MAX, tophat example.
3_8A.pbm	Image of Figure 3.8
3-8SE1.pbm	Structuring element, SE1 Figure 3.8
3_8SE2.pbm	Structuring element, SE2 Figure 3.8
3_8SE3.pbm	Structuring element, SE3 Figure 3.8
3_8SE4.pbm	Structuring element, SE4 Figure 3.8
3_8SE5.pbm	Structuring element, SE5 Figure 3.8
3_8SE6.pbm	Structuring element, SE6 Figure 3.8
A1.pbm	Figure 3.4
B1.pbm	Figure 3.4
circ.pbm	Circuit board image
circr.pbm	Circuit board, reversed

countme.pbm	Figure 3.22e
elise.pbm	Fur Elise, music image (Figure 3.9)
elise_se.pbm	Structuring element (Figure 3.9)
qrest.pbm	Image of a quarter rest
simple.pbm	Simple structuring element
squares.pbm	Image of squares (Figure 3.11)

3.7 References

Angulo, J. and J. Serra ''Automatic Analysis of DNA Microarray Images Using Mathematical Morphology,'' *Bioinformatics* 19, no. 5 (2003): 553–562.

Biomedical Imaging Group. ''Imaging Web Demonstrations,'' Ecole Polytechnique Federale de Lausanne, http://bigwww.epfl.ch/demo/jmorpho/index.html (accessed February 4, 2010).

Bloomberg, D. ''Grayscale Morphology,'' http://www.leptonica.com/grayscale-morphology.html (2010).

Dougherty, E. R. *An Introduction to Morphological Image Processing* Bellingham: SPIE Press, 1992.

Dougherty, E. R. and C. R. Giardina *Matrix Structured Image Processing* Englewood Cliffs: Prentice Hall, 1987.

Ehrlich, R., S. J. Crabtree, S. K. Kennedy, and R. L. Cannon ''Petrographic Image Analysis, I. Analysis of Reservoir Pore Complexes.'' *Journal of Sedimentary Petrology* 54, no. 4, (1981): 1365–1378.

Giardina, C. R. and E. R. Dougherty, *Morphological Methods in Image and Signal Processing*. Englewood Cliffs: Prentice-Hall, Inc., 1988.

Gonzalez, R. C. and R. E. Woods, *Digital Image Processing*. Reading: Addison-Wesley Publishing Company, 1992.

Haralick, R. M. and L. Shapiro. *Computer and Robot Vision,* Vol. 1., Reading: Addison Wesley, 1992.

Haralick, R. M., S. R. Sternberg, and X. Zhuang,. ''Image Analysis Using Mathematical Morphology,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 9, no. 44 (1987): 532–550.

Levialdi, S. ''On Shrinking Binary Picture Patterns,'' *Communications of the ACM* 15, no. 1 (1972): 7–10.

Meyer, H. and S.Beucher. ''Morphological Segmentation,'' *Journal of Visual Communication and Image Representation* 1, no. 1 (1990): 21–46.

Parker, J. R. and D. Horsley. ''Grey Level Encoding of Openings and Closings,'' paper presented at SPIE Vision Geometry II, Boston, Massachusetts, September. 9–10, 1993.

Parker, J. R. ''A High-Level Programming Language For Digital Morphology,'' paper presented at Vision Interface '98, Vancouver, British Columbia, June 17–20, 1998.

Schonfeld, D. and J. Goutsias. ''Optimal Morphological Pattern Restoration From Noisy Binary Images,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13, no. 1 (1991): 14–29.

Serra, J. *Image Analysis and Mathematical Morphology*. New York: Academic Press, 1982.

Serra, J. *Image Analysis and Mathematical Morphology*, Vol. 2, New York: Academic Press, 1988.

Shih, F. Y. C. and O. R. Mitchell ''Threshold Decomposition of Grey Scale Morphology Into Binary Morphology,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11, no. 1, (1989): 31–42.

Sternberg, S. R. ''Grayscale Morphology,'' *Computer Vision, Graphics, and Image Processing*, 35 (1986): 333–355.

CHAPTER

4

Grey-Level Segmentation

4.1 Basics of Grey-Level Segmentation

Grey-level segmentation, or *thresholding*, is a conversion between a grey-level image and a *bi-level* (or *monochrome*, or *black-and-white*) image. A bi-level image should contain all the essential information concerning the number, position, and shape of objects while containing a lot less information. The essential reason for classifying pixels by grey level is that pixels with similar levels in a nearby region usually belong to the same object, and reducing the complexity of the data simplifies many recognition and classification procedures. Thresholding is almost essential before thinning, vectorization, and morphological operations.

The most common way to convert between grey-level and bi-level images is to select a single threshold value. All the grey levels below this value will be classified as black (0), and those above will be white (1). The segmentation problem becomes one of selecting the proper value for the threshold T. Since most grey-level images possess only one byte per pixel, this means that there are usually only 256 different possible thresholds. Picking the best one could easily be done by eye but is less trivial to do using an algorithm.

What is being assumed here is that the pixels in an image I belong to one of two classes based on their grey level. The first class is the collection of black pixels, which will be given the value one, and for this class:

$$I(i,j) < T \qquad \text{(EQ 4.1)}$$

The other class consists of those pixels that will become white:

$$I(i,j) \geq T \qquad \text{(EQ 4.2)}$$

This assumption is only true in some real images because of noise and illumination effects. It is not generally true that a single threshold can be used to segment an image into objects and background regions, but it is true in enough useful cases to be used as an initial assumption. For example, documents scanned on any reasonable scanner these days can be thresholded into text and background with one threshold.

The threshold must be determined from the pixel values found in the image. Some measurement or set of measurements are made on the image, and from these, and from known characteristics of the image, the threshold is computed. One simple, but not especially good, example of this is the use of the mean grey level in the image as a threshold. This would cause about half of the pixels to become black and about half to become white. If this is appropriate, it is an easy computation to perform. However, few images will be half black. The program that thresholds an image in this way appears on the website, and is named `thrmean.c`. It takes two arguments: the first is the image to be thresholded, and the second is the name of the file in to which the thresholded image will be written.

Although fixing the percentage of black pixels at 50% is not a good idea, there are some image types that have a relatively fixed ratio of white to black pixels; text images are a common example. On a given page of text having known type styles and sizes the percentage of black pixels should be approximately constant. For example, on a sample of ten pages from this book the percentage of black pixels varied from 8.46% to 15.67%, with the smaller percentage being due to the existence of some equations on that page. Therefore, a threshold that would cause about 15% of the pixels to be black could be applied to this sort of image with the expectation of reasonable success.

An easy way to find a threshold of this sort is by using the histogram of the grey levels in the image. A histogram in this context is a vector having the same number of dimensions as the image does grey levels. The value assigned to each component (or *bin*) in the histogram h_i is the number of pixels with the grey level i. Obviously, the sum of all the components in the histogram equals the number of pixels. Given a histogram and the percentage of black pixels desired, we can determine the number of black pixels by multiplying the percentage by the total number of pixels. Then simply count the pixels in consecutive histogram bins, starting at bin 0, until the count is greater than or equal to the desired number of black pixels. The threshold is the grey level associated with last bin counted. This method appears on the website as the program `thrpct.c`; the program asks for the percentage of black pixels, where

50 would be 50% (as opposed to 0.5, which would be 0.5%). This method is quite old, and is sometimes called the *p-tile* method.

Using the histogram to select a threshold is a very common theme in thresholding. One observation frequently made is that when a threshold is obvious, it occurs at the low point between two peaks in the histogram. If the histogram has two peaks, then this selection for the threshold would appear to be a good one. The problem of selecting a threshold automatically now consists of two steps: locating the two peaks, and finding the low point between them.

Finding the first peak in the histogram is simple: it is the bin having the largest value. However, the second largest value is probably in the bin right next to the largest, rather than being the second peak. Because of this, locating the second peak is harder than it appears at first. A simple trick that frequently works well enough is to look for the second peak by multiplying the histogram values by the square of the distance from the first peak. This gives preference to peaks that are not close to the maximum. So, if the largest peak is at level j in the histogram, select the second peak as:

$$\max \left\{((k-j)^2 h[k]) | (0 \leq k \leq 255)\right\} \qquad \text{(EQ 4.3)}$$

where h is the histogram, and there are 256 grey levels, 0..255. This method is implemented by the program called `twopeaks.c`.

A better way to identify the peaks in the histogram is to observe that they result from many observations of grey levels that should be approximately the same except for small disturbances (noise). If the noise is presumed to be normally distributed, the peaks in the histogram could be approximated by Gaussian curves. Gaussians could be fit to the histogram, and the largest two used as the major peaks, the threshold being between them. This is an expensive proposition, with no promise of superior performance; we don't know how many Gaussians there are really, how near the means are to each other, or their standard deviations (still, see Section 4.2.1).

4.1.1 Using Edge Pixels

An edge pixel must be near to the boundary between an object and the background, or between two objects; that is why it is an edge pixel. As a result, the levels of the edge pixels are likely to be more consistent. Because they will sometimes be inside the object and sometimes be a little outside due to sampling concerns, the histogram of the levels of the edge pixels will be more regular than the overall histogram.

This idea was used to produce a thresholding method based on the digital Laplacian, which is a non-directional edge-detection operator [Weszka, 1974]. The threshold is found by first computing the Laplacian of the input image.

There are many ways to do this, but a simple one is to convolve the image with the mask:

$$\begin{matrix} 0 & 1 & 0 \\ 0 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

Now a histogram of the original image is found *considering only those pixels having large Laplacians*; those in the 85th percentile and above will do nicely. Those pixels having a Laplacian greater than 85% of their peers will have their grey level appear in the histogram, whereas all other pixels will not. Now the threshold is selected using the histogram thus computed.

Using a better approximation to the Laplacian should give better results, but in many cases this simple procedure will show an improvement over the previous methods. The C program on the website that computes a threshold using this method is called `thrlap.c`, and requires that the user enter the percentage value used to select the Laplacian values.

4.1.2 Iterative Selection

Iterative selection is a process in which an initial guess at a threshold is refined by consecutive passes through the image [Ridler, 1978]. It does not use the histogram directly, but instead thresholds the image into object and background classes repeatedly, using the levels in each class to improve the threshold.

The initial guess at the threshold is simply the mean grey level. This threshold is then used to collect statistics on the black and white regions obtained; the mean grey level for all pixels below the threshold is found (i.e., the black pixels) and called t_b, and the mean level of the pixels greater than or equal to the initial threshold (the white pixels) is called t_o. Now a new estimate of the threshold is computed as $(t_b + t_o)/2$, or the average of the mean levels in each pixel class, and the process is repeated using this threshold. When no change in threshold is found in two consecutive passes through the image, the process is stopped.

This is designed to work well in a hardware implementation, in which the initial estimate of a threshold assumes that the four corners of the image correspond to background regions and the rest of the image is used as an estimate of the object grey levels. However, in a software implementation the same threshold can be computed using the histogram rather than scanning the entire image during each iteration. This should be faster, since the histogram is a one-dimensional array of a fixed small size.

Starting with the initial estimate of the threshold T_0, the *kth* estimate of the threshold is [Thrussel, 1979]:

$$T_k = \frac{\sum_{i=0}^{T_{k-1}} i \cdot h[i]}{2\sum_{i=0}^{T_{k-1}} h[i]} + \frac{\sum_{j=T_{k-1}+1}^{N} j \cdot h[j]}{2\sum_{j=T_{k-1}+1}^{N} h[j]} \qquad \text{(EQ 4.4)}$$

where h is the histogram of the grey levels in the image. Again, when $T_k = T_{k+1}$ then T_k is the proper threshold. This is the actual method used in the C code for the program `thris.c`, which implements the iterative selection algorithm.

4.1.3 The Method of Grey-Level Histograms

The thresholding methods based on selecting the low point between two histogram peaks use the concept that object pixels and background pixels have different mean levels, and are random numbers drawn from one of two normal distributions. These distributions also have their own standard deviations and variances, where variance is the square of the standard deviation.

If there are two groups of pixels in the image, as suggested, then it is a simple matter to compute the overall, or *total*, variance of the grey level values in the image, denoted by ${\sigma_t}^2$. For any given threshold t, it is also possible to separately compute the variance of the object pixels and of the background pixels; these represent the *within-class* variance values, denoted by ${\sigma_w}^2$. Finally, the variation of the mean values for each class from the overall mean of all pixels defines a *between-classes* variance, which will be denoted by ${\sigma_b}^2$ This is the beginning of a method in statistics called *analysis of variance*, but we will not go too much further with it here. The important issue is that an optimal (in some respects) threshold can be found by minimizing the ratio of the between-class variance to the total variance [Otsu, 1979]; that is,

$$\eta(t) = \frac{\sigma_b^2}{\sigma_t^2} \qquad \text{(EQ 4.5)}$$

defines the needed ratio, and the value of t that gives the smallest value for h is the best threshold. Since ${\sigma_t}^2$ is the overall variance it is easy to calculate from the image, as is the overall mean μ_T. The between class variance is calculated by:

$$\sigma_b^2 = \omega_0\omega_1(\mu_0\mu_1)^2 \qquad \text{(EQ 4.6)}$$

where:

$$\omega_0 = \sum_{i=0}^{t} p_i \qquad \omega_1 = 1 - \omega_0 \tag{EQ 4.7}$$

and p_i is the probability of grey level i, or the histogram value at i divided by the total number of pixels. Also,

$$\mu_0 = \frac{\mu_t}{\omega_0} \qquad \mu_1 = \frac{\mu_T - \mu_t}{1 - \omega_0} \qquad \mu_t = \sum_{i=0}^{t} i \cdot p_i \tag{EQ 4.8}$$

All these values are quite easy to calculate from the histogram h of the image. Then $\eta(t)$ is computed for all possible values of t, and the t that gives the smallest value of $\eta(t)$ is the optimal threshold.

There is a program called `thrglh.c` on the website that thresholds an image using this method. It is run in exactly the same was as `thris`. A recent development is a fast version of this algorithm that gives the same results [Dong 2008].

4.1.4 Using Entropy

Entropy is a measure of information content, or how organized a system is. In information theory terms, assume that there are n possible symbols x (e.g., letters or digits) and that symbol i will occur with probability $p(x_i)$. Then the entropy associated with the source of the symbols X is

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log(p(x_i)) \tag{EQ 4.9}$$

where entropy is measured in bits/symbol if the base of the logarithm is 2.

The term ''symbol'' is really a more concrete term that, in this context, means *state*, and a system has a probability of being in a particular state with a particular probability. If the system has only one state and is always in that state, then the entropy is $-1.0 * \log(1.0) = 0$. If the system is a coin, then there are two states (= heads, tails) and the entropy is $-0.5\log(0.5) - 0.5\log(0.5) = 2 * 0.5 * 0.301 = 0.3$. It's more interesting if a logarithm base 2 is used: $-0.5\log_2(1/2) - 0.5\log_2(1/2) = -\log_2(1/2) = 1$. Units in this instance are bits, and a coin toss represents one bit of information. It can be seen that entropy is smaller when the system is more organized and larger when it is more random.

An image can be thought of as a source of symbols, or grey levels. The entropy associated with the black pixels, having been thresholded using some threshold value t, is as follows [Pun, 1980]:

$$H_b = -\sum_{i=0}^{t} p_i \ \log(p_i) \tag{EQ 4.10}$$

where p_i is the probability of grey level i. Similarly, the entropy of the white pixels is

$$H_w = -\sum_{i=t+1}^{255} p_i \, log(p_i) \qquad \text{(EQ 4.11)}$$

for an image with levels 0 . . . 255. The p_i values are really just scaled histogram bin values, the number of pixels in the image having a value i divided by the total number of pixels. The suggested algorithm attempts to find the threshold t that maximizes $H = H_b + H_w$, which Pun shows to be the same as maximizing:

$$f(t) = \frac{H_t}{H_T} \frac{\log P_t}{\log(\max\{p_0, p_1, \ldots p_t\})} + \left[1 - \frac{H_t}{H_T}\right] \frac{\log(1 - P_t)}{\log(\max\{p_{t+1}, p_{t+2}, \ldots p_{255}\})} \qquad \text{(EQ 4.12)}$$

where

$$H_t = -\sum_{i=0}^{t} p_i \log p_i \qquad \text{(EQ 4.13)}$$

is the entropy of the black pixels as thresholded by t,

$$H_T = -\sum_{i=0}^{255} p_i \log p_i \qquad \text{(EQ 4.14)}$$

is the total entropy, and

$$P_t = \sum_{i=0}^{t} p_i \qquad \text{(EQ 4.15)}$$

is the cumulative probability up to the grey level t, or the probability that a given pixel will have a value less than or equal to t. These three factors can be computed from the grey-level histogram, and EQ 4 .14 does not depend on t.

The threshold is found by computing H for all possible values of t, and selecting the t at the maximum point of H. The program `thrpun.c` thresholds an image using this algorithm.

In a variation on this theme, Kapur [1985] attempts to define an object probability distribution A and a background distribution B as follows:

$$A : \frac{p_0}{P_t}, \frac{p_1}{P_t}, \ldots, \frac{p_t}{P_t}$$
$$B : \frac{p_{t+1}}{1 - P_t}, \frac{p_{t+1}}{1 - P_t}, \ldots, \frac{p_{255}}{1 - P_t} \qquad \text{(EQ 4.16)}$$

Now the entropy of the black and white pixels is computed in a similar way to Equations 4.10 and 4.11, but using these new distributions:

$$H_b = -\sum_{i=0}^{t} \frac{p_i}{P_t} \log\left(\frac{p_i}{P_t}\right) \quad \text{(EQ 4.17)}$$

$$H_w = -\sum_{i=t+1}^{255} \frac{p_i}{1-P_t} \log\left(\frac{p_i}{1-P_t}\right) \quad \text{(EQ 4.18)}$$

The optimal threshold is the value of t that maximizes $H = H_b(t) + H_w(t)$. Once again, all thresholds between 0 and 255 are tried, and the one that gives the largest value of H is chosen. The C program `thrkapur.c` implements this algorithm.

Still another variation proposes to divide the grey levels into two parts so as to minimize the interdependence between them [Johannsen, 1982]. Without pursing this in too much detail, the method amounts to minimizing $S_b(t) + S_w(t)$ where:

$$S_b(t) = \log\left(\sum_{i=0}^{t} p_i\right) + \frac{1}{\sum_{i=0}^{t} p_i}\left[E(p_t) + E\left(\sum_{i=0}^{t-1} p_i\right)\right] \quad \text{(EQ 4.19)}$$

and

$$S_w(t) = \log\left(\sum_{i=t}^{255} p_i\right) + \frac{1}{\sum_{i=t}^{255} p_i}\left[E(p_t) + E\left(\sum_{i=t+1}^{255} p_i\right)\right] \quad \text{(EQ 4.20)}$$

and where again $E(x) = -x\log(x)$ is the entropy function. When implementing this algorithm, great care must be taken not to evaluate $S_b(t)$ and $S_w(t)$ for values of t, where $p_t = 0$. As a detailed example of an entropy-based method, the C code for the function `thrjoh.c` is given in Figure 4.1.

This is the function that does the bulk of the work for the program named `thrjoh.c` in implementing the algorithm outlined above, and contains code common to most of the entropy methods.

Most recently, researchers [Portes de Albuquerque, 2004] modify Kapur's method by using Tsallis entropy [Tsallis, 2001]. It introduces another parameter q (the degree of *nonextensivity*) and different versions of the expressions of Equations 4.17, 4.18, and, especially, 4.19. However, the basic idea is the same: to maximize the (Tsallis) entropy between the foreground and background. This is an interesting idea, but has not really yielded significantly better results than other techniques.

```
void thr_joh (IMAGE im)
{
             int i, j, t= -1, start, end;
             float Sb, Sw, Pt[256], hist[256], F[256], Pq[256];
             unsigned char *p;

/* Histogram */
              histogram (im, hist);

/* Compute the factors */
        Pt[0] = hist[0];      Pq[0] = 1.0 - Pt[0];
        for (i=1; i<256; i++)
        {
         Pt[i] = Pt[i-1] + hist[i];
         Pq[i] = 1.0 - Pt[i-1];
        }

        start = 0;             while (hist[start++] <= 0.0) ;
        end = 255;             while (hist[end--] <= 0.0) ;

/* Calculate the function to be minimized at all levels */
        for (i=start; i<=end; i++)
        {
         if (hist[i] <= 0.0) continue;
         Sb = (float)log((double)Pt[i]) + (1.0/Pt[i])*
                (entropy(hist[i])+entropy(Pt[i-1]));
         Sw = (float)log ((double)Pq[i]) + (1.0/Pq[i])*
                (entropy(hist[i]) + entropy(Pq[i+1]));
         F[i] = Sb+Sw;
         if (t<0) t = i;
         else if (F[i] < F[t]) t = i;
        }

/* Threshold */
        p = im->data[0];
        for (i=0; i<im->info->nr*im->info->nc; i++)
         if (*p < t) *p++ = 0;
         else*p++ = 255;
}

void histogram (IMAGE im, float *hist)
{
        int i;
        unsigned char *p;

        for (i=0; i<256; i++) hist[i] = 0.0;
        p = im->data[0];
        for (i=0; i<im->info->nc*im->info->nr; i++) hist[(*p++)] += 1.0;
        for (i=0; i<256; i++) hist[i] /= (float)im->info->nc*im->info->nr;
}

float entropy (float h)
{
        if (h > 0.0) return (-h * (float)log((double)(h)));
        else return 0.0;
}
```

Figure 4.1: Source code for thrjoh.c, an entropy-based thresholding algorithm.

4.1.5 Fuzzy Sets

In standard set theory, an element either belongs to a set or it does not. In a *fuzzy* set, an element x belongs to a set S to a particular degree u_x. When thresholding an image, we are attempting to classify pixels as belonging either to the set of background pixels or to the set of object pixels. The applicability of fuzzy sets to this problem seems apparent: some pixels can belong to the foreground and the background to a particular degree.

Fuzzy sets are a greatly misunderstood construct. In traditional set theory, membership is a simple true or false. In fuzzy sets there exists a membership function that gives the degree to which something belongs to the set. This corresponds to reality better; consider the set of young people, a standard example in fuzzy set theory. The membership function μ(age) indicates set membership as a function of age, and is largely a matter of opinion, but let's say that YOUNG is defined as:

μ_y (age) $= 1$ if age < 20

μ_y (age) $= -$age$/20 + 2$ for ages between 20 and 40, and

μ_y (age) $= 0$ for age > 40

Someone 30 years old has μ_y (30) $= 0.5$. Now define the set OLD as:

μ_o (age) $= 1$ if age < 30

μ_o (age) $=$ age$/30 - 1$ for ages between 30 and 60, and

μ_o (age) $= 0$ for age > 60

For a 35-year-old person, their degree of membership in the set YOUNG is $-35/20 + 2 = 0.25$ and their degree of membership in OLD is $35/30 - 1 = 0.166$. The same individual is a member of YOUNG and OLD to a certain degree. The membership is not a probability, though, even though it is a number between 0 and 1. Think of a car in a parking lot, straddling the yellow line making two stalls. Does this car have a 50% probability of being in stall 12, or is it parked in stall 12 (and 13) to the 0.5 degree (that is, halfway)?

There have been a number of attempts to use fuzzy sets in image segmentation, but the one to be described here uses a measure of *fuzziness*, which is a distance between the original grey-level image and the thresholded image [Huang, 1995]. By minimizing the fuzziness, the most accurate thresholded version of the image should be produced.

The first step is to determine the *membership function*, or the probability associated with the classification of each pixel as object or background. If the average grey level of the background is μ_0 and that of the objects is μ_1. The

smaller the difference between the level of any pixel x and the appropriate mean for its class, the greater will be the value of the membership function $u_x(x)$. A good membership function is:

$$u_x(x) = \begin{cases} \dfrac{1}{1 + |x - \mu_0|/C} & \text{if } x \leq t \\ \dfrac{1}{1 + |x - \mu_1|/C} & \text{if } x > t \end{cases} \qquad \text{(EQ 4.21)}$$

for a given threshold t. C is a constant, and is the difference between the maximum and minimum grey levels. Any pixel x will be in either the background set or the object set depending on the relationship between the grey level of the pixel and the threshold t. For an object pixel ($x > t$), the degree to which it belongs to the object set is given by $u_x(x)$, which should be a value between ½ and 1.

Given the membership function, how is the degree of fuzziness of the segmentation measured for a given t? For example, if the original image is already bi-level then a threshold of 0 should give exactly the same image back, and the fuzziness here should be zero. The maximum possible value of the fuzziness measure should probably be one. One way to measure fuzziness is based on the entropy of a fuzzy set, which is calculated using Shannon's function, rather than as we have been doing. Shannon's function is

$$H_f(x) = -x\log(x) - (1 - x)\log(1 - x) \qquad \text{(EQ 4.22)}$$

and so the entropy of the entire fuzzy set (the image) would be

$$E(t) = \frac{1}{MN}\sum_g H_f(\mu_x(g))h(g) \qquad \text{(EQ 4.23)}$$

for all grey levels g, where N and M are the number of rows and columns, and h is the grey-level histogram. This is a function of t because u_x is. Where $E(t)$ is a minimum t is the appropriate threshold that minimizes fuzziness.

Another measure of fuzziness is based on the idea that for a normal set A there are no elements in common between A and its complement. For a fuzzy set, on the other hand, each element may belong to A and to A^c with certain probabilities. The degree to which A and its complement are indistinct is a measure of how fuzzy A is [Yager, 1979]. This can be calculated using the expression

$$D_p(t) = \left[\sum_g |\mu_x(g) - \mu_{\overline{x}}(g)|^p\right]^{1/p} \qquad \text{(EQ 4.24)}$$

for levels g, where p is an integer and $\mu_{x'}(g) = 1 - \mu_x(g)$.

The value of p used defines a distance measure; $p = 2$ is used in the software here, which corresponds to a Euclidean distance.

Whichever fuzziness measure is used, an estimate for both m_0 and m_1 is needed. For a given threshold t we have:

$$\mu_0(t) = \frac{\sum_{g=0}^{t} g \cdot h(g)}{\sum_{g=0}^{t} h(g)} \qquad \text{(EQ 4.25)}$$

as the estimate of the background mean, and

$$\mu_1(t) = \frac{\sum_{g=t+1}^{254} g \cdot h(g)}{\sum_{g=t+1}^{254} h(g)} \qquad \text{(EQ 4.26)}$$

as the estimate of the object mean, where both values depend on the threshold.

Now everything needed to minimize the fuzziness is known; simply try all possible thresholds t and select the one that yields the minimum value of the fuzziness measure. The C program `thrfuz.c` implements both measures described; user must enter a code for the entropy calculation to be performed: 1 for standard entropy and 2 for Yager entropy (Equation 4.24).

4.1.6 Minimum Error Thresholding

The histogram of the image can be thought of as a measured probability density function of the two distributions (object pixels and background pixels). These are, as has been discussed, usually thought of as normal distributions, so the histogram is an approximation to

$$p(g) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\left(\frac{(g-\mu_1)^2}{2{\sigma_1}^2}\right)} + \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\left(\frac{(g-\mu_2)^2}{2{\sigma_2}^2}\right)} \qquad \text{(EQ 4.27)}$$

where σ_1 and μ_1 are the standard deviation and mean of one of the classes, and σ_2 and μ_2 are the standard deviation and mean of the other. After taking the log of both sides and rearranging, we get a quadratic equation that could be solved for g:

$$\frac{(g-\mu_1)^2}{\sigma_1^2} + \log\sigma_1 - 2\log P_1 = \frac{(g-\mu_2)^2}{\sigma_2^2} + \log\sigma_2 - 2\log P_2 \qquad \text{(EQ 4.28)}$$

However, the values of σ, μ, and P are not known, and they can be estimated only with some difficulty. Instead of that, Kittler and Illingworth [Kittler, 1986] created a new criterion function to be minimized:

$$J(t) = 1 + 2(P_1(t)\log\sigma_1(t) + P_2(t)\log\sigma_2(t)) - 2(P_1(t)\log P_1(t) + P_2(t)\log P_2(t)) \quad \text{(EQ 4.29)}$$

using formulas that should be starting to look familiar:

$$P_1(t) = \sum_{g=0}^{t} h(g) \qquad P_2(t) = \sum_{g=t+1}^{255} h(g) \quad \text{(EQ 4.30)}$$

$$\mu_1(t) = \frac{\sum_{g=0}^{t} g \cdot h(g)}{P_1(t)} \qquad \mu_2(t) = \frac{\sum_{g=t+1}^{255} g \cdot h(g)}{P_2(t)} \quad \text{(EQ 4.31)}$$

$$\sigma_1^2(t) = \frac{\sum_{g=0}^{t} h(g)(g-\mu_1(t))^2}{P_1(t)} \quad \text{(EQ 4.32)}$$

$$\sigma_2^2(t) = \frac{\sum_{g=t+1}^{255} h(g)(g-\mu_2(t))^2}{P_2(t)} \quad \text{(EQ 4.33)}$$

The value of t that minimizes $J(t)$ is the best threshold. This is often referred to as *minimum error thresholding* and is implemented by the program `thrme.c` on the website.

4.1.7 Sample Results From Single Threshold Selection

To this point, 13 different threshold selection methods have been discussed, and it would be interesting to see how they compare to one another by applying them to a set of sample images. Figures 4.2–4.4 show such a set, each having different properties that may present problems. Figure 4.2 is an example of an outdoor scene, which will be re-examined later (Chapter 5) when discussing texture segmentation; this will be called the *sky* image.

Figure 4.3 illustrates a typical problem for grey-level segmentation: an image containing printed text. This will be called the *pascal* image.

Finally, Figure 4.4 is a human face, which presents difficulties for many segmentation algorithms; it will be called the *face* image.

Any discussion of the quality of the results would be subjective. Indeed, if a perfect quality measure were available, it could be used as a thresholding algorithm. Still, it seems clear that the different methods perform well on different kinds of image. The minimum error method seems to deal best with the *sky* image, Otsu's Grey Level Histogram (GLH) algorithm with the *pascal* image, and Pun entropy with the *face* image.

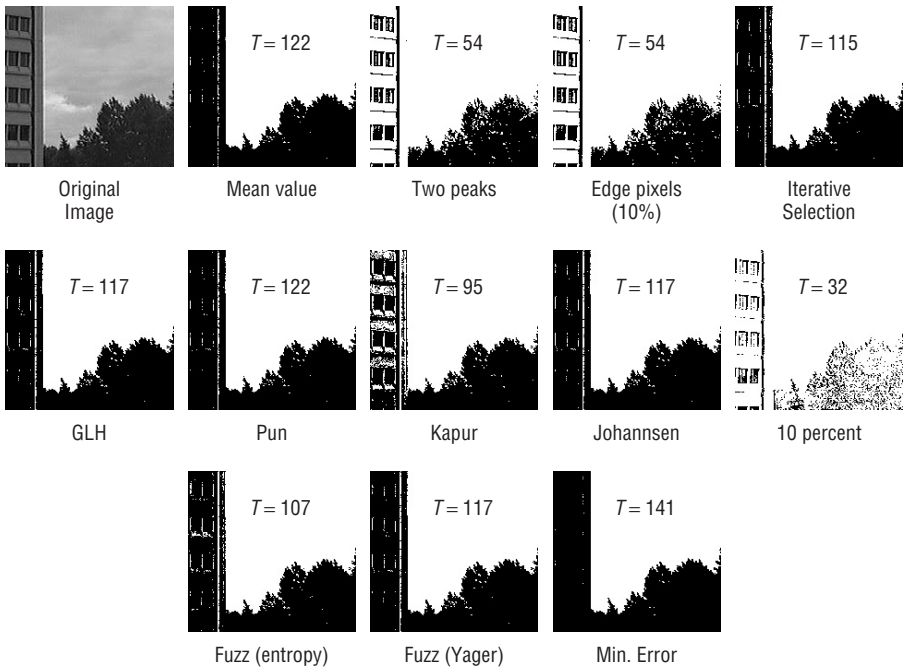


Figure 4.2: All 13 thresholding methods studied so far, applied to the sky image.

Figure 4.3: Sample results from the single threshold selection methods, using a hand-printed text image (pascal).

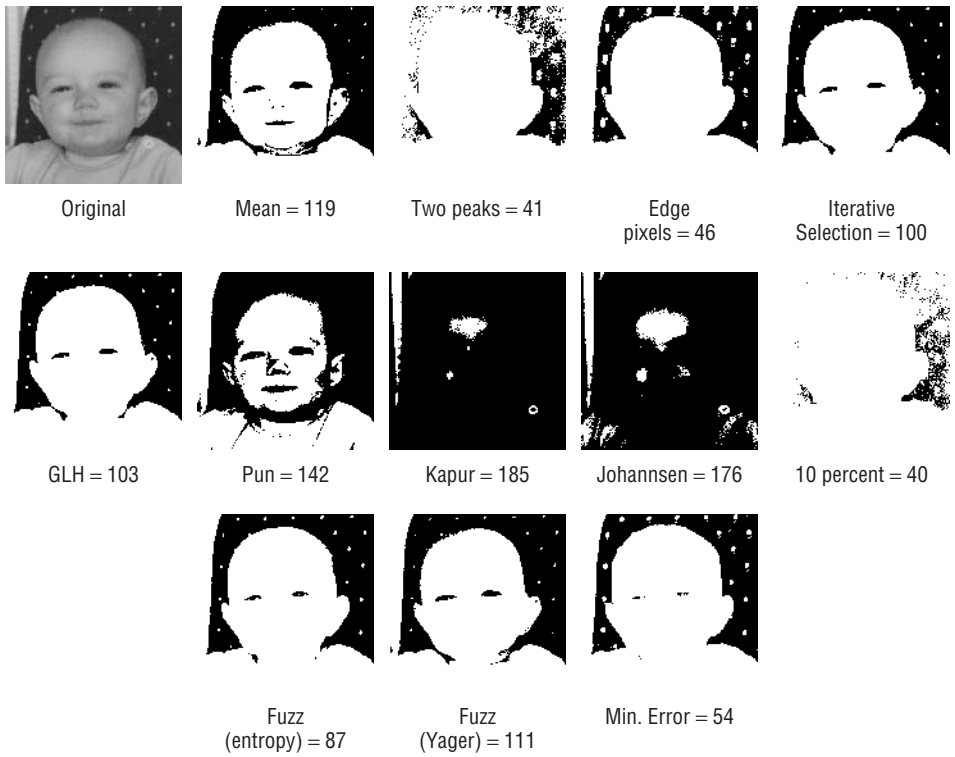


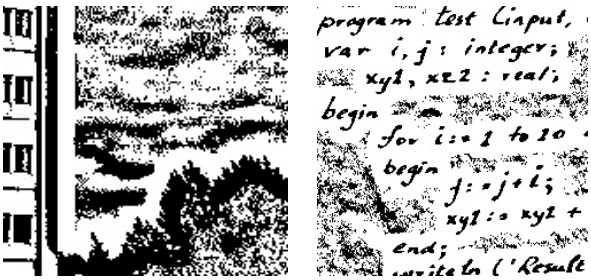
Figure 4.4: Sample results from the single threshold selection methods, using an image of a human face (face). This is the most difficult of the three images to threshold.

4.2 The Use of Regional Thresholds

So far in this discussion it has been presumed that the object pixels and the background pixels have non-overlapping grey levels. This need not be true, and leads to the conclusion that the selection of a single threshold for an image is not possible in all cases, perhaps not even in most. However, all that is needed is for the two classes of pixel not to overlap over each of a set of regions that collectively form the image. It may be that, for example, there is no single pixel that can threshold the entire image, but that there are *four* thresholds each of which can threshold a quarter of the image. This situation still results in a segmentation of the whole image but is simply more difficult to calculate.

The first issue with regional thresholds is the determination of how many regions are needed, and what the sizes of these regions are. Once that is done, it may be that any of the previously discussed algorithms can be used to give a threshold for each region, and the thresholding will simply be done in pieces. The number of regions can simply be dictated, by deciding to break up the original image to M sub-images.

As an illustration, let's do an experiment. An image will be thresholded using iterative selection on overlapping 21x21 regions centered on each pixel. The threshold found in each region will be used only as a threshold on the pixel at the center, giving one threshold per pixel. There will, of course, be a 10-pixel wide margin around the outside of the image that does not get thresholded, but that will be fixed later. What happens? The results for the sky image and the pascal image appear in Figure 4.5. (These images were created by the program `thrmulis.c`.)



(a) (b)

Figure 4.5: Using iterative selection to find a threshold for each pixel in the image. In all cases a square region of 21x21 pixels was used. (a) The sky image thresholded. (b) The pascal image thresholded. The method tries too hard to find object pixels, resulting in noise pixels being promoted to object pixels.

This is less than could be hoped for, but there is a simple explanation. The thresholding algorithm applied to each region attempts to divide the pixels into two groups, object and background, even when the region does not contain samples of both classes. When the region consists only of background pixels, the algorithm tries too hard, and creates two classes where only one exists. It is therefore necessary, when using regional methods, to make sure that either each region contains a sample of both object and background pixels, or that no thresholding is attempted when only one pixel class exists.

4.2.1 Chow and Kaneko

This method divides the image into 49 overlapping regions, each being 64x64 pixels [Chow, 1972]; this division is for 256x256 pixel images, and is not carved into stone. The histogram is found for each region, and a test of bimodality is performed. A bi-modal histogram is presumed to have two classes of pixels represented, and a threshold will therefore exist between the two peaks. Each bi-modal histogram then has a pair of Gaussian curves fit to it, using a least squares method. This should more precisely locate the two peaks, allowing an ''optimal'' threshold to be selected for each region.

The thresholds for regions not having bi-modal histograms are then interpolated from those regions that do, the assumption being that the region

with the interpolated threshold is probably all background or all object, and a neighboring threshold will suffice. Finally, a pixel-by-pixel interpolation of the threshold values is done, giving every pixel in the image its own threshold. This algorithm historically forms the foundation of regional thresholding methods, and is frequently cited in the literature. Although it was originally devised for the enhancement of boundaries in heart X-rays (cineangiograms, actually), it is a very nice example of how to approach a vision problem.

A bimodal histogram can be expressed as the sum of two Gaussians, as expressed mathematically in Equation 4.27. We want to obtain the values for the mean, standard deviation, and scaling factor for each of the two Gaussians, and can do this using a least squares approach. First, the histogram for the current window (a size of 16x16 was used) is found, and is smoothed in the following way:

$$F_s(i) = \frac{F(i-2) + 2F(i-1) + 3F(i) + 2F(i+1) + F(i+2)}{9} \qquad \text{(EQ 4.34)}$$

The smoothed version is less susceptible to noise than is the original. Now the histogram is divided into two parts at the lowest point in the smoothed version, which will be at index v. This assumption is that one of the Gaussians is to the left of this point, and the other is to the right of it. The initial parameters of each one can be estimated from the relevant portions of the histograms. The estimates are:

$$N_1 = \sum_{i=0}^{v} F(i) \qquad N_2 = \sum_{i=v+1}^{255} F(i) \qquad \text{(EQ 4.35)}$$

$$\mu_1 = \sum_{i=0}^{v} F(i) \cdot i \qquad \mu_2 = \sum_{i=v+1}^{255} F(i) \cdot i \qquad \text{(EQ 4.36)}$$

$$\sigma_1 = \sqrt{\frac{1}{N_1} \sum_{i=0}^{v} F(i) \cdot (i - \mu_1)^2} \qquad \text{(EQ 4.37)}$$

$$\sigma_2 = \sqrt{\frac{1}{N_2} \sum_{i=v+1}^{255} F(i) \cdot (i - \mu_2)^2} \qquad \text{(EQ 4.38)}$$

$$P_1 = \frac{\sigma_1 N_1}{\sum_{i=0}^{v} e^{-\left(\frac{(i-\mu_1)^2}{2\sigma^2{}_1}\right)}} \qquad \text{(EQ 4.39)}$$

$$P_2 = \frac{\sigma_2 N_2}{\sum_{i=0}^{v} e^{-\left(\frac{(i-\mu_2)^2}{2\sigma^2{}_2}\right)}} \qquad \text{(EQ 4.40)}$$

This is a slightly different form of a Gaussian; the leftmost Gaussian is defined as:

$$G_1(x) = \frac{P_1}{\sigma_1} e^{-\left(\frac{(x-\mu_1)^2}{2\sigma^2{}_1}\right)} \qquad \text{(EQ 4.41)}$$

and the rightmost is the same but with subscript 2.

With these estimates used as our initial guess for the parameters of the two Gaussians, the sum of the squared residuals is minimized:

$$R(P_1, \mu_1, \sigma_1, P_2, \mu_2, \sigma_2) = \sum_{i=0}^{255} (G_1(i) + G_2(i) - F(i))^2 \qquad \text{(EQ 4.42)}$$

The original program from 1972 was implemented using the FORTRAN language, and this program is no longer available. However, the procedure `powell` from the book *Numerical Recipes in C* [Press, 1988] seems to do an acceptable job in most cases. This procedure will minimize R by refining the estimates of the parameters.

When the fit is complete, the bi-modality of the two Gaussians is evaluated using four criteria. First, the means must differ by more than four grey levels ($\mu_2 - \mu_1 > 4$); the ratio of the standard deviations must be small, reflecting the fact that they are the same size within reasonable bounds ($0.05 < \sigma_1/\sigma_2 < 2.0$); and the ratio of the valley to peak must also be within a reasonable range. This last value is the smallest histogram value found between the two means divided by the smaller of the two values $F(\mu_1)$ and $F(\mu_2)$; its value should be less than 0.8.

If the histogram for the current window is not bi-modal, no threshold is selected for it. If it is bimodal, the point of intersection between the two Gaussians is selected as the threshold. This point is found by solving the quadratic equation:

$$\left(\frac{1}{\sigma_2^2} + \frac{1}{\sigma_2^2}\right) t^2 + 2\left(\frac{\mu_2}{\sigma_2^2} - \frac{\mu_1}{\sigma_1^2}\right) t + 2\log\left(\frac{P_2\sigma_1}{P_1\sigma_2}\right) = 0 \qquad \text{(EQ 4.43)}$$

When two solutions exist, use the value of t that is between μ_1 and μ_2. In this way, a threshold is chosen (or not) for each window. For each window not having a threshold, one is estimated from its neighbors, using a linear interpolation or simple weighting scheme. These are then smoothed by local weighted averaging using the following convolution-type mask:

$$\begin{matrix} \frac{1}{\sqrt{2}} & 1 & \frac{1}{\sqrt{2}} \\ 1 & 2 & 1 \\ \frac{1}{\sqrt{2}} & 1 & \frac{1}{\sqrt{2}} \end{matrix}$$

Finally, each pixel in the image is assigned a threshold estimated from the threshold of the surrounding four windows by linear interpolation. Figure 4.6 illustrates the situation for a pixel in between windows A, B, C, and D.

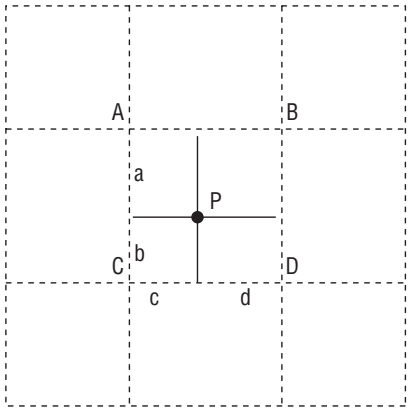


Figure 4.6: Linear interpolation of individual pixel thresholds. A, B, C, and D are the centers of adjacent windows, having thresholds Ta, Tb, Tc, and Td. The distance a is the distance of the pixel P vertically to the point A; b is the distance from P vertically to C; c is the distance horizontally to C; and d is the distance horizontally to D.

For this case, the threshold at the point P would be

$$T = \frac{bdT_A + bcT_B + adT_C + acT_D}{(a+b)(c+d)} \qquad \text{(EQ 4.44)}$$

The thresholds for pixels not having enough valid neighboring windows to perform an interpolation are simply taken from the nearest window having a threshold. This applies to pixels on the boundary of the image as well.

The algorithm outlined above is not exactly the Chow-Kaneko method, but is probably fairly close. It could be applied to the three test images of Figures 4.2–4.4, but this would not properly show off the advantages of multiple region thresholding. Instead, an intensity gradient will be imposed on the images, as shown in Figure 4.7.

A linear gradient, a Gaussian spot, and a sine-wave were super-imposed over the existing images. The results, if thresholded using the best algorithm previously found for that image, illustrate the problem resulting from using a single threshold.

Selecting one threshold per pixel or per region always takes longer than selecting a single threshold, sometimes substantially longer. Whether the results justify the extra time is something that must be judged on a case-by-case basis.

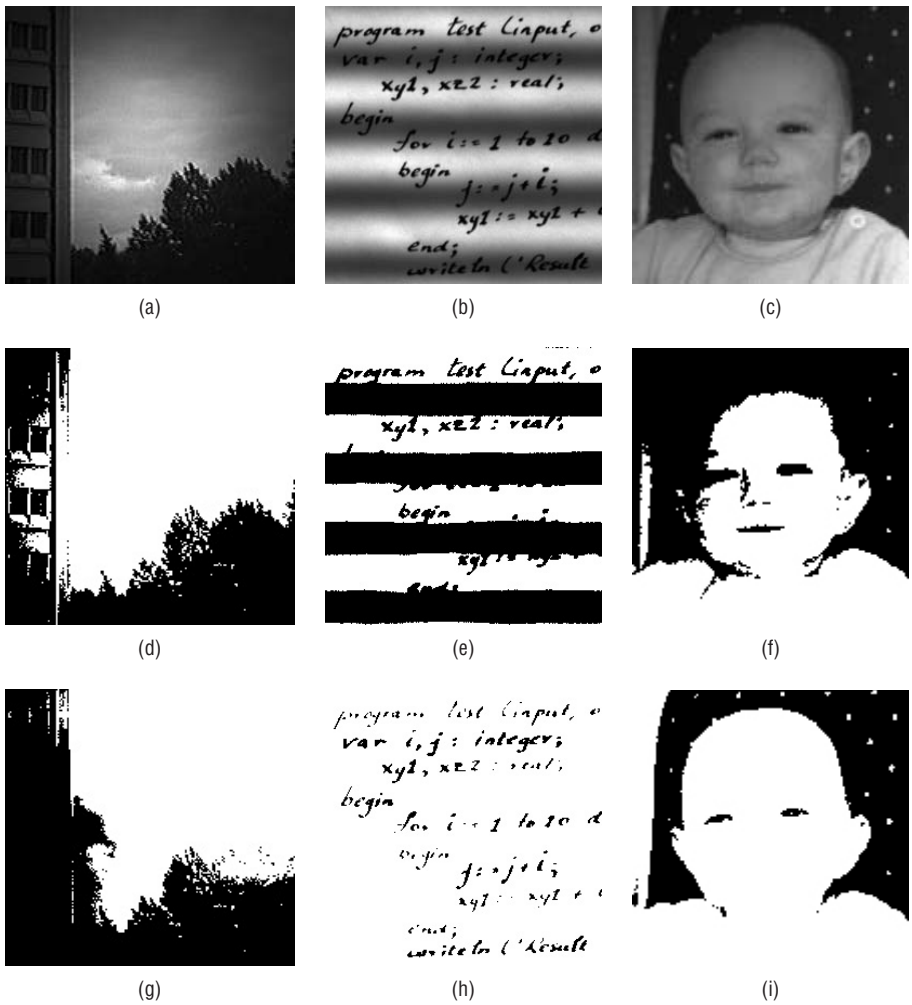


Figure 4.7: Badly illuminated images threshold by Chow-Kaneko and by standard methods. (a–c) The original images with imposed illumination (sky, Gaussian; pascal, sine wave; face, linear). (d–f) The badly illuminated images thresholded using the best thresholding method from the previous trial. (g–i) The same images thresholded by Chow-Kaneko.

4.2.2 Modeling Illumination Using Edges

If the illumination falling on an object is known, then the task of segmenting the pixels belonging to the object is much simpler. This is because the intensity of a pixel in an image is proportional to the product of the illumination at that point and the color (reflectivity) of the object there. If the illumination gradient were known, it could be factored out leaving a relatively simply task of thresholding based only on the nature of the objects.

Figure 4.8 shows some images that have illumination problems, making them difficult to threshold.

(a) (b) (c)

Figure 4.8: Difficult images to threshold, either due to illumination gradients or low contrast. (a) The pascal image with a linear illumination gradient. (b) The face image with a Gaussian gradient. (c) An X-ray image having generally low contrast, which is typical of X-rays. Used with permission of Big Hill Veterinary Clinic, Cochrane, Alberta.

A single threshold will certainly not do the job for any of these images, and small regions will result in artifacts at the region boundaries. One threshold per pixel is best here, since any larger region may be subject to distortion through illumination effects. The method will decide what these threshold values are to be based on *local* properties of the image, specifically based on the levels of known object pixels. The confidence in the local threshold decreases with the distance from a known object pixel.

One approach to thresholding is based on the principle that objects in an image provide the high spatial frequency component and illumination consists mainly of lower spatial frequencies. These two were multiplied together to produce the image. Another way to look at this is to say that the objects in an image will produce small regions with a relatively large intensity gradient, those being at the boundaries of objects, whereas other areas ought to have a relatively small gradient; this fact is used in many edge enhancement algorithms. In this way a sample of the object pixels in an image can be found by looking for regions of high gradient and assuming that these pixels belong to an object that would appear as distinct in a thresholded picture.

This thresholding method involves first locating ''objects'' in an image by using the Shen-Castan edge detector (see Section 2.5) to locate pixels that belong to boundaries of objects. This edge detector has good localization properties, and a pixel that has been determined to be on an edge will be assumed to be a part of an object. The grey levels at edge pixels should be indicative of the levels elsewhere in object regions. A surface is produced that fits the levels at the edges, and this surface is presumed to give reasonable estimates of likely grey levels at object pixels that do not lie on an edge. Pixels

significantly above this surface will be assumed to belong to the background, and those at or below it will belong to the object. This method is capable of thresholding images that have been produced in the context of variable illumination, and is called *edge-level thresholding* (ELT).

The method used to fit a surface to the edge points is a moving least-squares (MLS) scheme [Salkauskas, 1981]. This involves solving a weighted least-squares problem at each point in the plane. That is, if

$$J(x,y) = \sum_{i=1}^{N} w_i(x,y)(I(x_i,y_i) - S(x_i,y_i)) \qquad \text{(EQ 4.45)}$$

where N is the number of data points which are given by $I(x_i, y_i)$, $s(x, y) = ax + by + c$ and $w_i(x, y)$ are weights, then we find values for a, b, and c so that $J(x,y)$ is minimized at each point (x,y) in the plane. The weights depend on the evaluation points, and hence the requirement that we perform this minimization at each point.

The weight function $w_i(x, y)$ has several important properties. It essentially weights the data point (x_i, y_i) inversely according to its distance from the current evaluation point (x,y). If the data is further than some specified distance h from (x,y), we assume that it should have no bearing whatsoever on the height of the surface at that point, and so the weight is zero. Another parameter for the weight function, d, determines the fidelity of the surface to the data. When d is zero, the weight is essentially infinite when the evaluation point is also a data point; the result is a surface that actually passes through all the data, and this can lead to extreme fluctuations. As d increases towards 1, the fidelity increases and the surface relaxes, tending to average out fluctuations. When $d = 1$, our weight function is defined to be constant for all data and no longer has compact support. The resulting surface is simply the standard least-squares planar approximation to the entire data set.

Since at every point we are looking for a least-squares plane, there is a rigid mathematical requirement that we have at least three data values in the disk of radius h centered at each point in the image. Without these, the linear system will be under-determined, and we won't be able to find a solution to the least-squares problem.

Others working on this problem have suggested methods for getting an approximation to the edge data, but these are all interpolants, and as we have previously mentioned, this is not necessarily desirable. One method described is a moving weighted average, which is simply an MLS method with $S(x,y) = a$. The resulting surface will have horizontal tangent planes or ''flat-spots'' at each of the data points.

In addition, as the evaluation moves away from data, the value of the weighted average will tend to the actual mean of the data. This may cause

unusual artifacts if the illumination gradient is actually linear. Figure 4.9 shows an example of weighted averages, restricted to the one-dimensional case. Figure 4.9b shows an example of the MLS method applied to the same data.

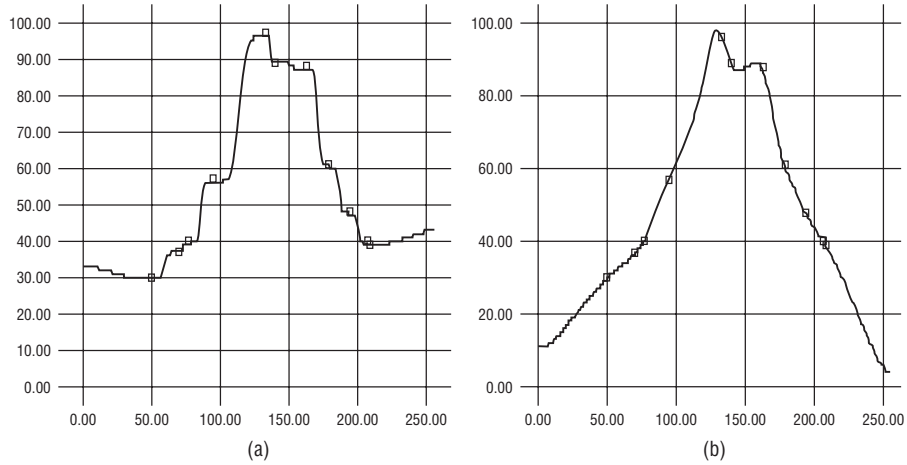


Figure 4.9: Weighted averaging versus MLS. (a) For a one-dimensional sample, the weighted average method shows flat spots at each data point. (b) MLS gives a smooth curve.

4.2.3 Implementation and Results

The ELT thresholding software consists of three major modules: the ISEF edge detector, the MLS surface fit, and the thresholding module. The function and sequence of execution is best described by using an example. Consider the image of Figure 4.8b, in which a bright Gaussian spot has been superimposed on the image of a map. The first step is edge detection by ISEF, and the result is shown in Figure 4.10a. Notice that clean edges are found even in the dark areas of the image. This is the secret of the ELT method: ISEF finds edges *very* well, and these edges are well localized.

Next, the grey levels at all edge pixels are used to form the basis of a surface, and the levels at the non-edge pixels are estimated from this surface, as found by the MLS procedure. For this case the surface is shown in Figure 4.10b as grey levels. The value of the function at the edge pixels will be very near to the actual value of the corresponding image pixel, and will be assumed to be near to the value of non-edge object pixels as well. The final stage is a pass through the image, setting all pixels to zero if they are less than the value of the fit function +10, and setting them to MAX (255) otherwise. Figures 4.10c–e show the results.

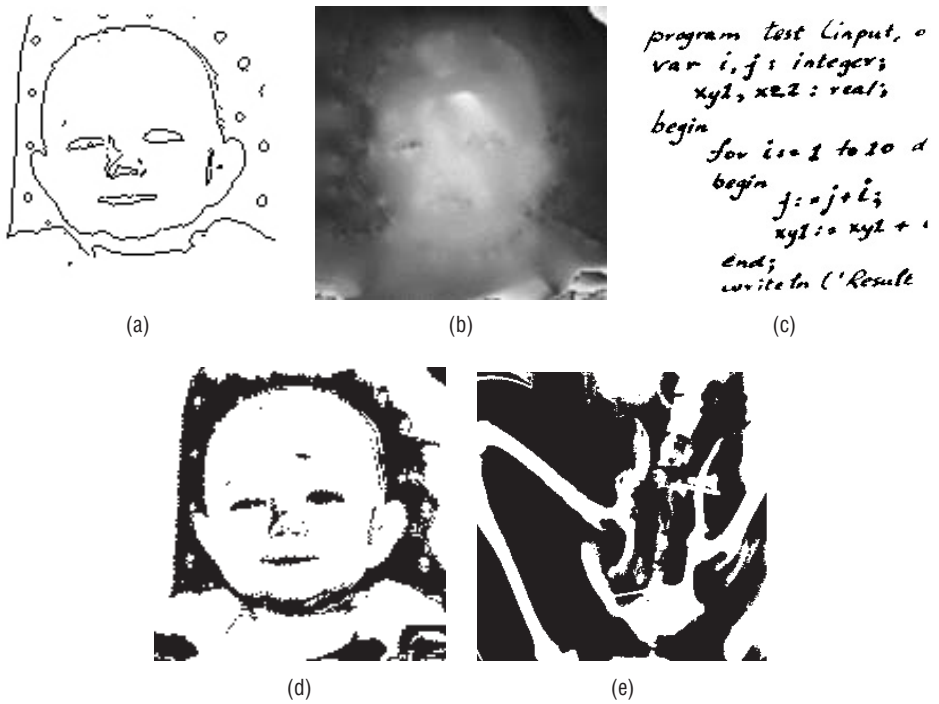


Figure 4.10: ELT example. (a) Shen-Castan edges for Figure 4.8b. (b) Surface fit to the edge pixels. (c) The thresholded version of Figure 4.8a. (d) The thresholded version of 4.8b. (e) The thresholded version of 4.8c.

In general, the results of ELT thresholding are better than other algorithms in situations of poor illumination, especially when compared to single-threshold methods. Standard methods give results that have large black areas where the illumination reaches low levels, and the objects can't be determined from the background. ELT permits widely varying thresholds across the image.

4.2.4 Comparisons

The ELT algorithm has been compared with other thresholding methods, and none has given the same results in widely varying illumination environments. A major problem is that it is quite slow at this time, taking many minutes to threshold even a 256x256 image. However, there are a few parameters to the algorithm that could be adjusted: for example, the MLS code used relies on a fixed value for the radius h for the entire image. This presents some problems since in some regions a large radius is required so that we have at least three points, while in other areas the points are so dense that the same size disk will include hundreds of points.

4.3 Relaxation Methods

Relaxation is an iterative process. For the specific problem of image thresholding, the thresholds for any given iteration are computed as a function of those in the same neighborhood at the previous iteration. The following algorithm illustrates this process:

1. Create an initial guess at a segmentation for the image. Provide an estimate of the confidence in that guess at each pixel.
2. For each pixel, modify the segmentation and the confidence estimate based on the pixels in the local region; the surrounding eight pixels will do.
3. Repeat step 3 until the segmentation is complete. This might occur when no further changes are seen in successive steps.

The confidence estimates have the appearance of probabilities, although they may not be accurate in that regard; all that matters is that they are fairly accurate with respect to the other pixels, especially those in the local neighborhood.

One way to find an initial classification is to use the mean grey level as a benchmark. A pixel greater than the mean has a probability of being white — that is, in proportion to the relative distance of its grey level from the level ¾ along the total range. For a pixel less than the mean we use ¼ of the grey-level range. Thus, one possibility for the initial classification is [Rosenfeld, 1981]:

$$p_i^0 = \frac{1}{2} + \frac{1}{2}\frac{g_i - \mu}{\max - \mu} \qquad \text{(EQ 4.46)}$$

This describes the situation for a pixel greater than the mean, where m is the mean value, max is the largest grey level, and g_i is the grey level of pixel i. For pixels less than the mean, we have:

$$q_i^0 = \frac{1}{2} + \frac{1}{2}\frac{\mu - g_i}{\mu - \min} \qquad \text{(EQ 4.47)}$$

The value $p_i{}^0$ is the initial probability that pixel i is white, and $q_i{}^0$ is the probability that it is black. The superscript refers to the iteration, which is currently zero.

Now the problem is: given that the probabilities of being white and black are known for a pixel and its neighborhood, how can the probabilities be refined so that they are driven to either end of the spectrum more clearly? Ideally these probabilities should become one or zero, giving a firm segmentation. What is needed is some measure of *compatibility*, which can be used to decide whether a particular classification is reasonable or not. For example, if a pixel is black and all its neighbors are white, it would seem likely that the black

pixel should become white. The compatibility of that pixel with its neighbors is low, suggesting a change.

Compatibility will be estimated by a function $C(i, c_1, j, c_2)$ which returns a measure, between -1 and 1, of how compatible pixel i, which is class c_1, is with pixel j, which is class c_2. For a thresholding problem, there are only two classes, black or white, and for a small neighborhood the pixels i and j will be adjacent to each other. It is not possible to know for certain what this function should be, because it depends on probabilities found in the final thresholded image, and that is not known. However, a simple implementation would have $C = 1$ when $c_1 = c_2$, and $C = -1$ otherwise; that is, pixels are compatible if they agree.

Now, since there are two classes possible for any pixel, the average of these could be used as an overall compatibility between any two pixels:

$$Q_{ij} = C(i, c_1, j, \text{ white})p_j + C(i, c_1, j, \text{ black})q_j \qquad \text{(EQ 4.48)}$$

The compatibility of a region around the pixel i can be defined as the average compatibility of all eight neighbors:

$$Q_i(c_1) = \frac{1}{8}\sum_{j \in N} C(i, c_1, j, \text{ white})p_j + C(i, c_1, j, \text{ black})q_j \qquad \text{(EQ 4.49)}$$

for the one-pixel neighborhood N centered at i. This will be the net increment to p_i each time the probabilities are updated. However, to ensure that the p and q values remain positive, add 1 to Q. Then the values should be normalized over the region. This gives the following updating scheme:

$$p_i^{k+1} = \frac{p_i^k(1 + Q_i^k)}{p_i^k(1 + Q_i^k(\text{white})) + q_i^k(1 + Q_i^k(\text{black}))} \qquad \text{(EQ 4.50)}$$

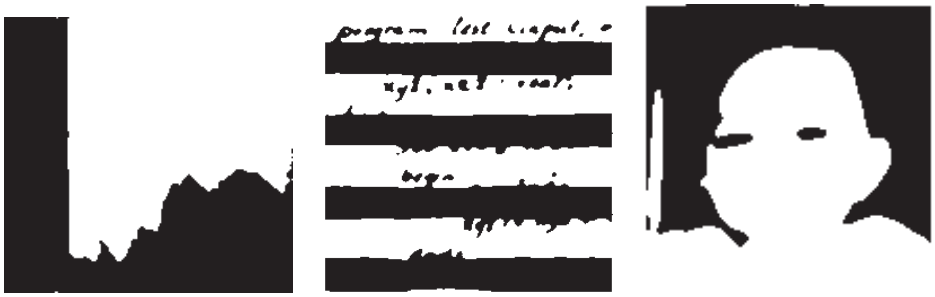
where the superscript reflects the iteration number. A similar expression holds for the q values.

Each iteration of the relaxation process involves looking at all pixels in the image and updating the p (and q) values. Once a p becomes 0 or 1, it stays that way; thus, the initial classification is very important to the success of the method. In fact, the actual pixel values are never examined after the initial classification is complete; all further processing is performed on the probabilities.

Figure 4.11 shows some of the segmentations that result from the method, implemented directly from Equations 4.46–4.50. These were created by the program `relax.c`, the source code of which can be found on the website. It is clear, especially from the pascal image with sine-wave illumination, that something is wrong. Because a single mean for the whole image was used in the initial segmentation the whole process gets off to a bad start. Areas that are too dark initially can never recover, except a little bit at the boundaries.

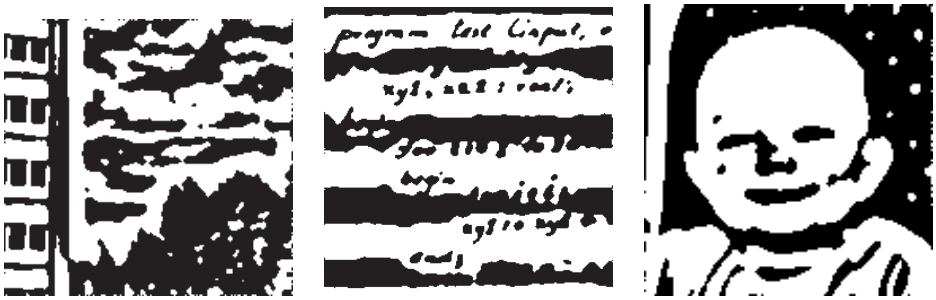
For this reason, a modification was made to the program, which is found in the program `relax2.c`. In this version, the initial classification is done using small regions of the image, instead of the whole thing. The hope is that the more localized initial classification will now permit some of the illumination effects

to be accounted for by the relaxation process. Figure 4.12 shows the results of this exercise. These images are a little better. It would appear that the relaxation approach, as implemented here, is quite sensitive to the initial classification.



(a) (b) (c)

Figure 4.11: Relaxation thresholding: (a) the sky image, (b) the pascal image with sine-wave illumination, (c) the face image with linear illumination.



(a) (b) (c)

Figure 4.12: Relaxation thresholding, using an initial classification based on regional thresholds. These are better than those in Figure 4.11 but are not as good as some of the methods examined already.

Finally, as an illustration of the kind of experimentation that can be done, a new version of the relaxation program (called `relax3.c` on the website) was devised. The function Q_i in Equation 4.49 does not make use of the actual grey levels in the image, and as a result the levels are never used after the initial classification is performed. This is changed in `relax3.c` so that the function Q_i returns a ''probability'' related to the difference in level of the two pixels involved. Pixels that are both supposed to be black in the segmented image should have levels that are near to each other; pixels that are supposed to be different should have differing levels.

Although the results are not significantly different from those of Figure 4.12, this version of the program was selected to appear in Figure 4.13 because it reflects an approach: experimentation is to be encouraged.

```
/* Relaxation method 2 - Rosenfeld & Kak  */

void thr_relax (IMAGE im)
{
        float res = 0.0, minres = 10000.0, **p, **q;
        int iter = 0, i, j, count = 0;

/* Space allocation */
        p = f2d (im->info->nr, im->info->nc);
        q = f2d (im->info->nr, im->info->nc);
        pp = f2d (im->info->nr, im->info->nc);
        qq = f2d (im->info->nr, im->info->nc);

/* Initial assignment of pixel classes */
        assign_class (im, p, q);

/* Relaxation */
        do
        {
          res = update (im, p, q);
          iter += 1;
          printf ("Iteration %d residual is %f\n", iter, res);
          if (res < minres)
          {
                minres = res;
                count = 1;
          } else if (fabs(res-minres) < 0.0001)
          {
                if (count > 2) break;
                else count++;
          }
        } while (iter < 100 && res > 1.0);

        thresh (im, p, q);
}

/* Threshold */
void thresh (IMAGE im, float **p, float **q)
{
        int i,j;

        for (i=0; i<im->info->nr; i++)
          for (j=0; j<im->info->nc; j++)
            if (p[i][j] > q[i][j]) im->data[i][j] = 0;
             else im->data[i][j] = 255;
}
float R(IMAGE im, int r, int c, int rj, int cj, int l1, int l2)
{
```

Figure 4.13: C source code for the program relax3.c; it thresholds an image using a relaxation method.

```
        float xd = 0.0;

        xd = 1.0 - (im->data[r][c] - im->data[rj][cj])/256.0;
        if (l1 == l2) return xd*0.9;
        return -(1.0-xd)*0.9;
}

void assign_class (IMAGE im, float **p, float **q)
{
        int i,j;
        float ud2, y, z, u, lm2;

        for (i=0; i<im->info->nr; i++)  /* Mean of local area */
          for (j=0; j<im->info->nc; j++)
          {
                meanminmax (im, i, j, &u, &z, &y);
                ud2 = 2.0*(u-z);
                lm2 = 2.0*(y-u);
                if (im->data[i][j] <= u)
                {
                  p[i][j] = 0.5 + (u-im->data[i][j])/ud2;
                  q[i][j] = 1.0-p[i][j];
                } else {
                  q[i][j] = (0.5 + (im->data[i][j]-u)/lm2);
                  p[i][j] = 1.0-q[i][j];
                }
          }
}

void meanminmax (IMAGE im, int r, int c, float *mean, float *xmin,
float *xmax)
{
int i,j, sum = 0, k=0;
        unsigned char *y;

        y = im->data[0];
        *xmin = *xmax = im->data[r][c];;
        for (i=r-10; i<=r+10; i++)
          for (j=c-10; j<=c+10; j++)
          {
            if (range(im, i, j) != 1) continue;
            if (*xmin > im->data[i][j]) *xmin = im->data[i][j];
            else if (*xmax < im->data[i][j]) *xmax = im->data[i][j];
            sum += im->data[i][j];
            k++;
          }
        *mean = (float)sum/(float)(k);
}
```

Figure 4.13: (continued)

```
float Q(IMAGE im, float **p, float **q, int r, int c, int class)
{
        int i,j;
        float sum = 0.0;

        for (i=r-1; i<=r+1; i++)
          for (j=c-1; j<=c+1; j++)
            if (i!=r || j!=c)
              sum += R(im, r, c, i, j,class, 0)*p[i][j] +
                        R(im, r, c, i, j,class, 1)*q[i][j];
        return sum/8.0;
}

float update (IMAGE im, float **p, float **q)
{
        float z, num, qw, pk, qb;
        int i,j;

        for (i=1; i<im->info->nr-1; i++)
          for (j=1; j<im->info->nc-1; j++)
          {
            qb = (1.0 + Q(im, p, q, i, j, 0));
            qw = (1.0 + Q(im, p, q, i, j, 1));
            pk = p[i][j]*qb + q[i][j]*qw;

            if (pk == 0.0)
            {
              continue;
            }

            pp[i][j] = p[i][j]*qb/pk;
            qq[i][j] = q[i][j]*qw/pk;
          }

        z = 0.0;
        for (i=1; i<im->info->nr-1; i++)
          for (j=1; j<im->info->nc-1; j++)
          {
                z += fabs(p[i][j]-pp[i][j]) + fabs(q[i][j]-qq[i][j]);
                p[i][j] = pp[i][j];
                q[i][j] = qq[i][j];
                qq[i][j] = pp[i][j] = 0.0;
          }
        return z;
}
```

Figure 4.13: (continued)

There are a great number of ways to update the probabilities, to weight the Q values, and to perform the initial classification. A good algorithm is soundly based in mathematics and good sense, but there is a lot of leeway in how it might be implemented.

4.4 Moving Averages

The relaxation method has one serious drawback that has not been mentioned — it is very slow. If speed is a criterion of interest, then a method that uses *moving averages* is quite appealing [Wellner, 1993]. This algorithm yields one threshold per pixel very quickly, and gives surprisingly good segmentations. It is designed for images containing text; for example, scanned documents. In these cases the illumination can be expected to be good, as can the general image quality.

A moving average is just the mean grey level of the last n pixels seen. The image can be treated as a one-dimensional stream of pixels, which is common in C anyway, and the average can either be computed exactly or estimated via:

$$M_{i+1} = M_i - \frac{M_i}{n} + g_{i+1} \qquad \text{(EQ 4.51)}$$

where M_{i+1} is the estimate of the moving average for pixel $i + 1$ having grey level g_{i+1}, and M_i is the previous moving average (i.e., for pixel i).

Any pixel less than a fixed percentage of its moving average is set to black, otherwise to white. To avoid a bias for one side of the image over the other, a novel scanning method called *boustrophedon*[1] scanning was employed. This means traversing pixels in opposite directions in every other line. That is, the pixel following the last one in row i is the last one in row $i + 1$, followed by the second last in row $i + 1$, and so on back to the start of row $i + 1$; this is followed by the start pixel in row $i + 2$, and so to the final one. This avoids the discontinuity at the end of the line that occurs with the usual C method of scanning a two-dimensional array.

The process begins with an estimate of the moving average; a value of $127 * n$ was selected, and this will only affect the first few pixels in the image. The value of n used is the number of columns divided by 8. Now Equation 4.51 is used to compute the moving average estimate for the next pixel (the first), which is used immediately as a threshold:

$$V = \begin{cases} 0 & \text{if } g_i < \left(\frac{M_i}{n}\right) \times \left(\frac{100 - pct}{100}\right) \\ 255 & \text{otherwise} \end{cases} \qquad \text{(EQ 4.52)}$$

[1]Greek, meaning ''as the ox plows.''

where V is the thresholded pixel and pct is the fixed percentage value; $pct = 15$ was used for the examples shown here.

A simple extension of this process averages the current threshold with the one from the row above, allowing the vertical propagation of grey-level variations and illumination changes. This is not done in the program `thrdd.c`, which implements this scheme, but is an easy addition.

Figure 4.14 contains some of the images segmented by this method. The results are fairly good, at least until 4.14d. The white region in the margins seems to have fooled it, at least in this case. Still, the program is very quick, and is 64 lines of C code (without the I/O functions), requiring at most two rows to be in memory at one time.

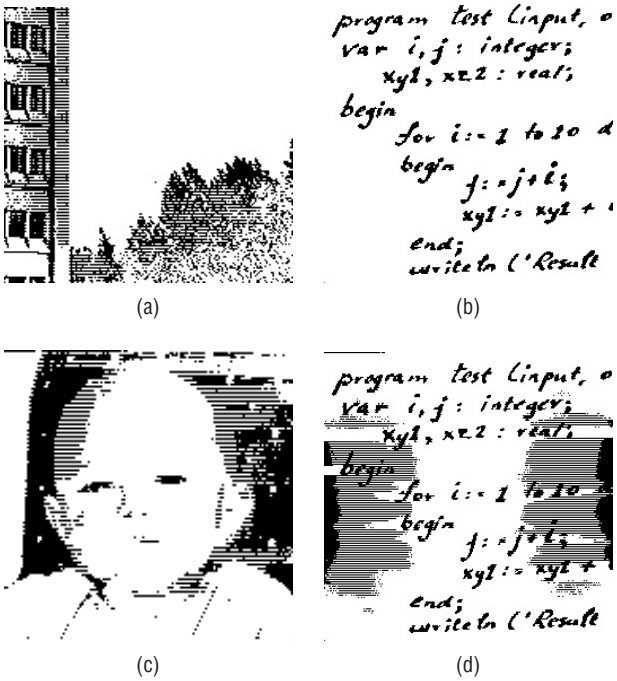


Figure 4.14: Images segmented by the moving averages method. (a) The sky image. (b) The pascal image with superimposed sine-wave illumination. (c) The face image with linear illumination. (d) The pascal image with Gaussian illumination. This method works best on images of text, for which it was designed.

A likely pitfall is the fixed percentage value used to select the threshold from the mean. It is unlikely that a single value will be appropriate for use with a variety of image types. It does, however, seem highly appropriate for thresholding text, a function that it seems it was designed to perform.
Figure 4.15 gives the complete source code for `thrdd.c`.

```
#define MAX
#include "lib.h"
void thrdd (IMAGE im);

void main (int argc, char *argv[])
{
        IMAGE data;

        if (argc < 3)
        {
          printf ("Usage: thrdd <input file> <output file>\n");
          exit (0);
        }

        data = Input_PBM (argv[1]);
        if (data == NULL)
        {
                printf ("Bad input file '%s'\n", argv[1]);
                exit(1);
        }
        thrdd (data);
        Output_PBM (data, argv[2]);
}

void thrdd (IMAGE im)
 {
         int NC, row, col, inc;
         float mean, s, sum;
         unsigned char *p;
         long N, i;

         N = (long)im->info->nc * (long)im->info->nr;
         NC = im->info->nc; s = (int)(float)(NC/Navg);
         sum = 127*s; row = col = 0; inc = 1;
         p = &(im->data[0][0]);

         for (i=0; i<N-1; i++) {
           if (col >= NC) {
             col = NC-1; row++; inc = -1;
             p = &(im->data[row][col]);
           } else if (col < 0)
           {
             col = 0; row++; inc = 1;
             p = &(im->data[row][col]);
           }
           sum = sum - sum/s + *p;
           mean = sum/s;
           if (*p < mean*(100-pct)/100.0) *p = 0; else *p = 255;
           p += inc; col += inc;
         }
 }
```

Figure 4.15: Source code for the program thrdd.c: adaptive thresholding using a moving average.

4.5 Cluster-Based Thresholds

The prior discussion of the use of the grey levels on edge pixels to build local thresholds leads naturally to a discussion of the role of distance and local geometry in determining thresholds [Kwon, 2004]. Kwon suggests the use of a cluster-analysis technique to group the pixels into foreground and background based on a threshold and geometric distances. In particular, vectors that represent the mean of the two classes are created by scanning the image with a trial threshold, t. Then the sum of the squared distances between the class mean and each pixel in the class is computed and used as a significant part of an objective function $J(t)$ to be minimized. This function is computed for all values of t and the threshold corresponding to the smallest value of J.
The class mean vectors are:

$$\overline{v}_1 = \frac{1}{N_1} \sum_{x_k \in X_1} \overline{x}_k \qquad \overline{v}_2 = \frac{1}{N_2} \sum_{x_k \in X_2} \overline{x}_k \tag{EQ 4.53}$$

where N_1 and N_2 are the number of pixels in the foreground (black) and background (white) classes, respectively, and X_1 and X_2 are the sets of pixels comprising each class. The $\overline{x}_k$ are vectors representing the pixels: $\overline{x}_k = ({x_k}^1, {x_k}^2)$, and the variables $\overline{v}_1$ and $\overline{v}_2$ also represent vectors with components being the mean i and j coordinate of pixels in each class.
An overall mean could be calculated as

$$\overline{v} = \frac{1}{N_1 + N_2} \sum_{x_k \in X} \overline{x}_k \tag{EQ 4.54}$$

Given these components, the objective function to be minimized by this thresholding algorithm is:

$$J_k(t) = \frac{\sum_{x_k \in X_1} p^2 \left\| \overline{x}_1 - \overline{v}_1 \right\|^2 + \sum_{x_k \in X_1} p^2 \left\| \overline{x}_2 - \overline{v}_2 \right\|^2 + \sum_{i=1}^{2} \left\| \overline{v}_i - \overline{v} \right\|^2}{\left\| \overline{v}_1 - \overline{v}_2 \right\|^2} \tag{EQ 4.55}$$

The value p is a normalizing factor and is given by $p = 1/(N_1 + N_2)$. Computing the threshold is a matter of finding the t for which $J_k(t)$ is a minimum, which means calculating J_k for all possible t. Don't forget that each time t changes so do the sets x_1 and x_2. Note that this has some significant similarities in basic design to the minimum error method discussed in Section 4.1.6 and some of the other methods discussed.

Sample results from this algorithm are given in Figure 4.16. The results on our standard images are not spectacular, but on some pathological images it works better than most. However, this method has an advantage: it is relatively simple to add more pixel classes and to use more than one threshold to distinguish between them. It is also a natural extension to use this method

over multiple sub-images within a larger one, where each sub-image will have a distinct $\overline{v}$.

(a) (b) (c)

Figure 4.16: Sample results from the cluster-based thresholding algorithm

4.6 Multiple Thresholds

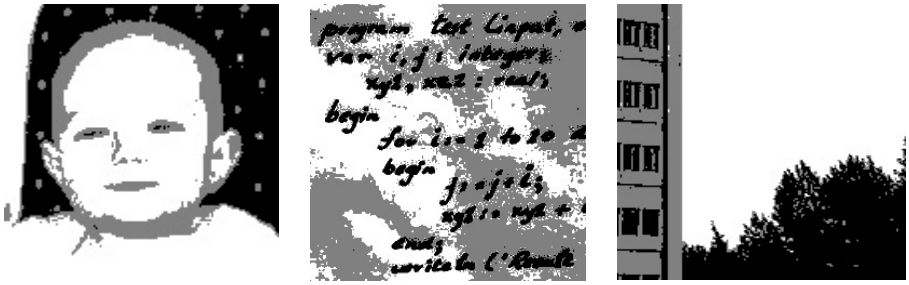
The clear object/background dichotomy leads to the idea that pixels belong to two classes and that there should be a way to distinguish between them. The use of a simple, single threshold value for this purpose is elegant, in spite of the large number of ways in which it may be determined. There are some situations for which the dichotomy is less relevant, and some for which more than one pixel class can be found. For these cases, multiple thresholds can be used. An early discussion of this subject can be found in [Wang, 1984]. However, many of the methods discussed so far can be extended to more than one threshold. Specifically, the original description of the minimum error thresholding method [Kittler, 1986] has specifically included an extension for multiple thresholds, and will be used as an example.

For two thresholds (three regions), the objective function of Equation 4.29 is extended to become

$$J(t_1,t_2) = 1 + 2\left(\sum_{i=1}^{2} P_i(t_1,t_2)\log(\sigma_i(t_1,t_2))\right) \\ -2\left(\sum_{i=1}^{2} P_i(t_1,t_2)\log(P_i(t_1,t_2))\right) \qquad \text{(EQ 4.56)}$$

There are now three of each of the histogram-based functions P_i, σ_i, and μ_i representing the three regions segmented by the two thresholds, and each function needs the two thresholds as parameters. It is plain how to extend this to an arbitrary number of regions. Figure 4.17 shows the images resulting from using two thresholds to segment the familiar set of test images. In these

instances, the pixels $\leq t_1$ become 0, those $> t_2$ become 255, and those between are set to 128.



(a) (b) (c)

Figure 4.17: Sample results from the cluster-based thresholding algorithm

In most cases, multiple thresholds are used in particular circumstances, and those circumstances dictate the way the thresholds are computed. Mixed text and image documents would use a different method from X-rays and CAT scan images. It would be essential to characterize the nature of the pixels in each class and in what ways they are distinct from those in other classes for a multiple thresholding technique to be effective.

4.7 Website Files

`kwon.c`	Thresholding using clustering
`relax.c`	Relaxation method, Rosenfeld & Kak
`relax2.c`	Relaxation method, Rosenfeld & Kak
`relax3.c`	Relaxation method, Rosenfeld & Kak
`thrdd.c`	Adaptive thresholding (digital desk)
`thrfuz.c`	Fuzziness minimization
`thrglh.c`	Grey-level histograms (Otsu)
`thris.c`	Iterative selection
`thrjoh.c`	Johannsen method using entropy
`thrkapur.c`	Kapur method using entropy
`thrlap.c`	Use of Laplacian

thrme.c	Minimum error
thrme2.c	Minimum error, two thresholds
thrmean.c	Mean of image grey level
thrmulis.c	Iterative selection over multiple regions
thrpct.c	Percentage of black pixels
thrpun.c	Pun method using entropy
twopeaks.c	Find threshold between two histogram peaks
face.jpg	Face image
faceg.jpg	Face image with Gaussian illumination
facel.jpg	Face image with linear illumination
faces.jpg	Face image with sinusoidal illumination
pascal.jpg	Text image
pascals.jpg	Text image with sinusoidal illumination
sky.jpg	Sky image
skyg.jpg	Sky image with Gaussian illumination
skyl.jpg	Sky image with linear illumination

4.8 References

Bracho, R. and A. C. Sanderson, ''Segmentation of Images Based On Intensity Gradient Information,'' *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Francisco, June 19–23 (1985): 341–347.

Chow, C. K. and T. Kaneko. ''Automatic Boundary Detection of the Left Ventricle from Cineangiograms,'' *Computers and Biomedical Research* 5 (1972): 388–410.

Dong, L., G. Yu P. Ogunbona, and W. Li ''An Efficient Iterative Algorithm for Image Thresholding,'' *Pattern Recognition Letters* 29 (2008): 1311–1316

Huang, L. K. and M. J. Wang. ''Image Thresholding by Minimizing the Measures of Fuzziness,'' *Pattern Recognition* 28, no. 1 (1995): 41–51.

Johannsen, G., and J. Bille ''A Threshold Selection Method Using Information Measures,'' *Proceedings of the Sixth International Conference on Pattern Recognition*, Munich, Germany (1982): 140–143.

Kapur, J. N., P. K. Sahoo, and A. K. C. Wong. ''A New Method for Gray-Level Picture Thresholding Using the Entropy of the Histogram,'' *Computer Vision, Graphics, and Image Processing* 29, no. 3 (1985): 273–285.

Kittler, J. and J. Illingworth. ''On Threshold Selection Using Clustering Criteria,'' *IEEE Transactions on Systems, Man, and Cybernetics* 15, no. 5 (1985): 652–655.

Kwon, Soon H. ''Threshold Selection Based on Cluster Analysis,'' *Pattern Recognition Letters* 25 (2004): 1045–1050.

Lancaster, P. and K. Salkauskas ''Surfaces Generated by Moving Least Squares Methods,'' *Mathematics of Computation*. 37 (1981): 141–158.

Oh, W. and B. Lindquist. ''Image Thresholding by Indicator Kriging,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 21 (1999): 590–602.

Otsu, N. ''A Threshold Selection Method from Grey-level Histograms,'' *IEEE Transactions on Systems, Man, and Cybernetics* 9, no. 1 (1979): 377–393.

Parker, J. R. ''Grey Level Thresholding in Badly Illuminated Images,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13, no. 8: (1991).

Perez, A. and R. C. Gonzalez. ''An Iterative Thresholding Algorithm for Image Processing,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 9, no. 6 (1987):

Portes de Albuquerque, M., I. A. Esquef , and A. R. Gesualdi Mello ''Image Thresholding Using Tsallis Entropy,'' *Pattern Recognition Letters* 25 (2004): 1059–1065.

Press, W. H., *Numerical Recipes in C*. Cambridge: Cambridge University Press, 1988.

Prewitt, J. M. S. ''Object Enhancement and Extraction'', in *Picture Processing and Psychopictorics*, ed. B. Lipkin and A. Rosenfeld. New York: Academic Press, 1970.

Pun, T. ''A New Method for Grey-Level Picture Thresholding Using the Entropy of the Histogram,'' *Signal Processing* 2, no. 3 (1980): 223–237.

Ridler, T. W. and S. Calvard ''Picture Thresholding Using an Iterative Selection Method,'' *IEEE Transactions on Systems, Man, and Cybernetics* 8, no. 8 (1978): 630–632.

Rosenfeld, A. and R. C. Smith. ''Thresholding Using Relaxation,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 3 (1981): 598–606.

Rosenfeld, A. and A. C. Kak *Digital Picture Processing*. New York: Academic Press, 1976.

Sahoo, P. K., ''A Survey of Thresholding Techniques,'' *Computer Vision, Graphics, and Image Processing* 41, no. 2 (1988): 233–260.

Salkauskas, K. and P. Lancaster *Curve and Surface Fitting, An Introduction*. New York: Academic Press, 1981.

Sankur, B. and M. Sezgin ''A Survey Over Image Thresholding Techniques and Quantitative Performance Evaluation,'' *Journal of Electron Imaging* 13 no. 1 (2004): 146–165.

Sezgin, M. and R. Tasaltin ''A New Dichotomization Technique to Multilevel Thresholding Devoted to Inspection Applications, Pattern Recognition,'' *Pattern Recognition Letters* 21, no. 2 (2000): 151–161.

Theodoridis, S. and K. Koutroumbas *Pattern Recognition*. London: Academic Press, 2003.

Thrussel, H. J. ''Comments on ''Picture Thresholding Using an Iterative Selection Method,'' *IEEE Transactions on Systems, Man, and Cybernetics* 9, no. 5 (1979): 311.

Tsallis, C. ''Nonextensive Statistical Mechanics and its Applications,'' ed. S. Abe, Y. Okamoto Series Lecture Notes in Physics, Berlin: Springer-Verlag, 2001 (see `http://tsallis.cat.cbpf.br/biblio.htm`).

Wang, Shyuan and R. M. Haralick ''Automatic Multi-threshold Selection,'' *Computer Vision, Graphics, and Image Processing* 25, no. 1 (1984): 46–67.

Wellner, P. D. ''Adaptive Thresholding for the DigitalDesk,'' *EuroPARC Technical Report EPC-93-110* (1993):

Weszka, J., C. Dyer , and A. Rosenfeld ''A Comparative Study of Texture measures for Terrain Classification,'' *IEEE Transactions on Systems, Man, and Cybernetics* 6, no.4 (1976):

Wilson, R. and M. Spann *Image Segmentation and Uncertainty*. New York: John Wiley & Sons Inc., 1988.

Yager, R. R. ''On the Measures of Fuzziness and Negation, Part 1: Membership in the Unit Interval,'' *International Journal of General Systems*. 5 (1979): 221–229.

Yang, Y. and H. Yan ''An Adaptive Logical Method for Binarization of Degraded Document Images,'' *Pattern Recognition* 33 (2000): 787–807.

Yanowitz, S. D. and A. M. Bruckstein ''A New Method for Image Segmentation,'' *Computer Vision, Graphics, and Image Processing* 46, no. 1 (1989): 82–95.

Yin, P. Y. ''Maximum Entropy-Based Optimal Threshold Selection Using Deterministic Reinforcement Learning with Controlled Randomization,'' *Signal Processing* 82, no. 7 (2002): 993–1006.

CHAPTER

5

Texture and Color

5.1 Texture and Segmentation

When we look at a picture, we can easily connect regions having a similar grey or color value into objects. Indeed, we can even account for variations in level caused by illumination and distinguish those from changes caused by overlapping objects. The presence of *texture*, and to a lesser extent *color*, complicates the issue, especially from a computer vision perspective.

While there is no agreement on a formal definition of texture, a major characteristic is the *repetition of a pattern or patterns over a region*. The pattern may be repeated exactly, or as a set of small variations on the theme, possibly a function of position. There is also a random aspect to texture that must not be ignored: the size, shape, color, and orientation of the elements of the pattern (sometimes called *textons*) can vary over the region. Sometimes the difference between two textures is contained in the degree of variation alone, or in the statistical distribution found relating the textons. In either case, it is more difficult to characterize the degree of difference or similarity between two textured portions of an image than it is to find a difference in grey level.

Color does not represent a pattern or arrangement of elements, but it is also more complex than grey level. Color possesses three coordinates — whether RGB, HSV, or some other set — that are essentially orthogonal. A small change in one coordinate may represent a tiny perceptual change, but result in a large distance between the colors involved. Changes in hue are not necessarily reflected as concomitant changes in intensity. There are also many millions of

colors involved with most images, as opposed to the 256 or so normally found in grey-level pictures. This means that enumerating thresholds and other such computations require a lot of time to complete.

Some of the color segmentation work that will be done amounts to reducing the number of colors to a small number, one that represents the number or regions perhaps. Some will be concerned with reducing the dimensionality of the problem. But in both cases, for texture and for color, the idea is to simplify the image so as to see the larger components more clearly. The basic principle is: *Parts of the image that belong to a particular object have pixels that are more like each other than they are like pixels that belong to other objects.*

Figure 5.1 shows a small collection of textures, some natural and some artificial. The study of texture will be undertaken with the goal of segmenting regions rather than characterizing textures. That is, the very practical issue of determining which regions have texture A and which have texture B will be addressed. The result could be an image in which texture has been replaced by a unique grey level or color, or is given a descriptive label.

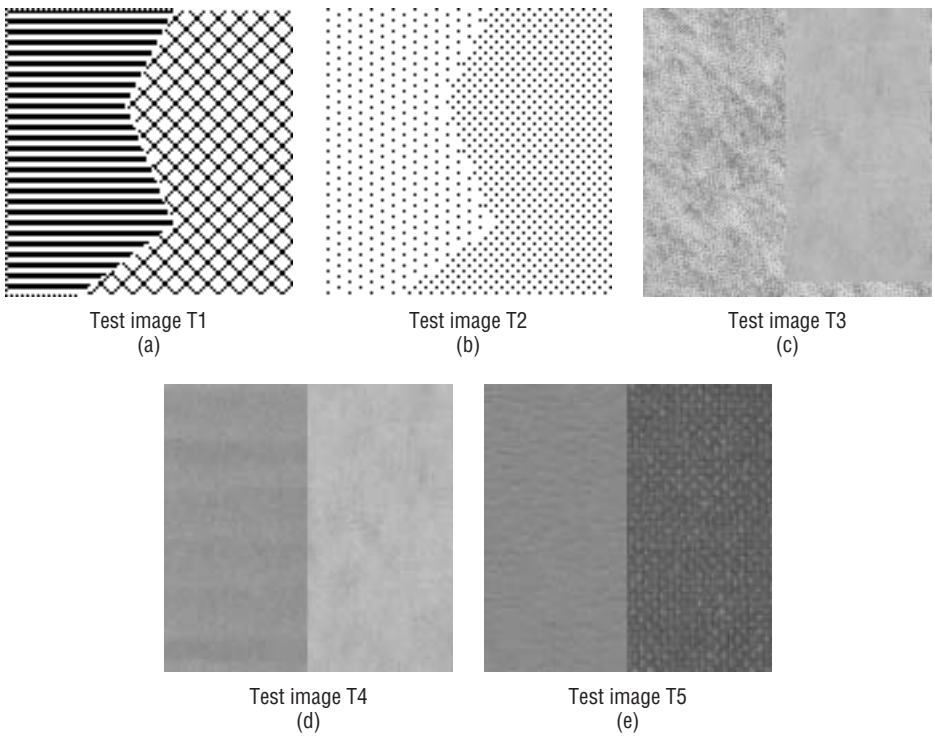


Figure 5.1: Some images displaying regions characterized by their texture. (a–b) Artificial textures, such as might be used on maps to delimit regions of interest. (c–e) Naturally occurring textures that delimit regions found in real scenes.

Texture is a property possessed by a region that is sufficiently large to demonstrate its recurring nature. A region cannot display texture if it is small compared with the size of a texton. This leaves us with a problem of scale, in addition to the other problems that texture presents. Indeed, the same texture at two different scales will be perceived as two different textures, provided that the scales are different enough. As the scales become closer together the textures are harder to distinguish, and at some point they become the same.

It is unlikely, given the preceding discussion, that any simple measure or operation will allow the segmentation of textured regions in a digital image. The lines drawn between textures are often arbitrary, a function of perception rather than mathematics. It is possible, on the other hand, that some combination of operations can yield reasonably good segmentations with a wide range of textures.

5.2 A Simple Analysis of Texture in Grey-Level Images

A region having a particular texture, while having a wide variety in its grey levels, must have some properties that allow animal visual systems to identify them. While the existence of texture elements is crucial, it is unlikely that a library of texture elements is maintained and recognized by any seeing creature. It is more likely that similarities and differences can be seen, and that a biological vision system can measure these and use them to delimit different textural regions.

One obvious way to delimit regions is by color or grey level alone. However, unlike grey-level segmentation, (in which each pixel is classed as white or black), the grey level associated with each pixel in a textured region could be the average (mean) level over some relatively small area. This area, which will be called a *window*, can be varied in size to capture a sample of the different scales to be found there.

The use of the average grey level is not recommended to distinguish between textures, but the use of a method this simple does help explain the general method that will be used to segment image regions according to texture. The use of windows of some sort is very common, since texture is only a concern in a region and not in individual pixels. As a result, the boundary between textured regions can only be determined to within a distance of about W pixels, where W is the width of the window.

Figure 5.2 illustrates this; we are using mean grey levels to segment the image seen in Figure 5.1b. The method is: For each pixel in the image, replace it by the average of the levels seen in a region $W \times W$ pixels in size centered on that pixel. Now threshold the image into two regions using the new

(average) levels. Figure 5.2a is the image of the mean grey levels and 5.2b is a thresholded version of this, showing the original boundary between the region superimposed. The exact location of the boundary between the regions depends on the threshold that is applied to the mean-level image. A reasonable threshold in this case is one that keeps the white region contiguous and on the left of the image, and keeps the black region contiguous and on the right. There is a range of reasonable thresholds, and the pixels that move from one region to the other as the threshold changes are those marked in grey in Figure 5.2c. This corresponds to the boundary between the regions, and is essentially an area of uncertainty. The actual boundary could be anywhere in the grey area, but is most likely in the middle.

Figure 5.2: Windowing in texture segmentation. (a) Image consisting of mean grey levels in a 14x14 window about all pixels in Figure 5.1b. (b) A thresholding of this image showing two possible regions. (c) The region of uncertainty. The actual region boundary could be anywhere within the grey area.

The use of mean grey level seems to work well enough for the sample image, but this happens to be an exceptional case. It does not work at all well for any of the remaining sample images when they are normalized for grey level. On the other hand, using the standard deviation of the grey levels in a small window works a lot better. There seems to be more consistency in the *changes* in the levels than there is in the levels themselves. This does make some degree of sense: We have seen that even in the presence of varying illumination, the difference in level between local pixels remains nearly the same. In addition, if the texture elements are small objects with a different set of levels than the background, then the standard deviation in a small region tells us something about how many pixels in that region belong to textons and how many belong to the background.

Figure 5.3 shows the regions identified by using local standard deviations to segment textural areas. Figure 5.1a can be segmented by this method; in fact, so can all except 5.1d. Note that the precision with which the boundary between regions is known is still a function of the size of the window used.

This will always be true, and so there is an advantage in using methods that permit the use of small windows.

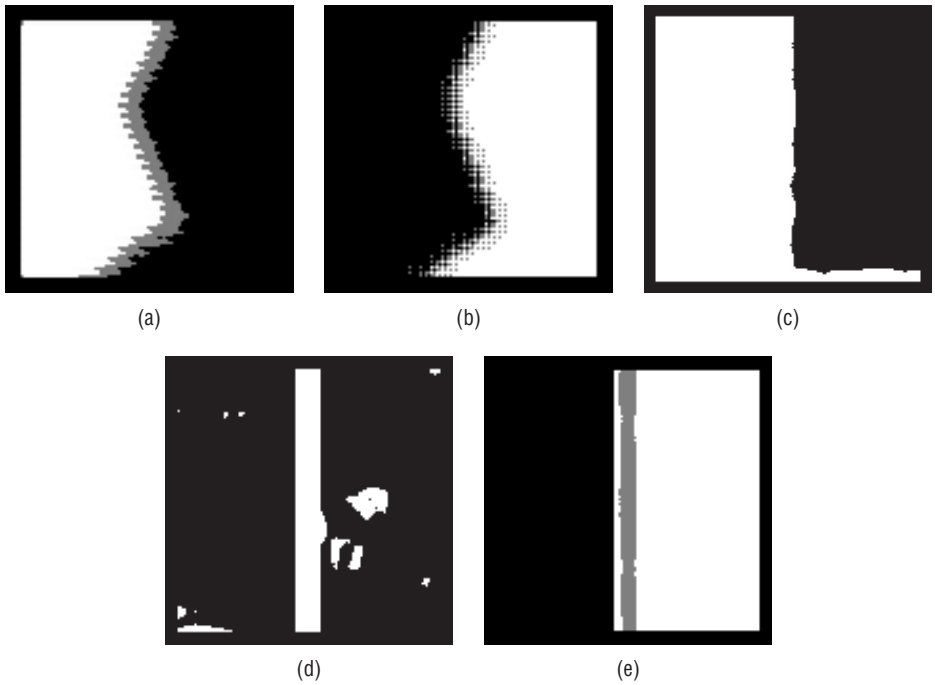


Figure 5.3: Use of standard deviation to segment the images of Figure 5.1. Note that the standard deviation tends to be large where the two textures meet.

The mean grey level and the standard deviation are known as statistical *moments*. The mean is related to the first moment, and the standard deviation depends on the second moment. There are others, and these could also be used to characterize textured areas. In general, we have:

$$M_n = \frac{\Sigma(x - \overline{x})^n}{N} \qquad \text{(EQ 5.1)}$$

where N is the number of data points, and n is the order of the moment. Now the *skewness* can be defined as:

$$Sk = \frac{1}{N}\Sigma\left(\frac{(x - \overline{x})}{\sigma}\right)^3 \qquad \text{(EQ 5.2)}$$

and the *kurtosis* as:

$$Kurt = \frac{1}{N}\Sigma\left(\frac{(x - \overline{x})}{\sigma}\right)^4 \qquad \text{(EQ 5.3)}$$

These can be used in the same way as standard deviation: compute the statistic over a window and let that be the value of the pixel at the center of the window, and then segment the resulting image. Rather large windows would be preferred, so that a statistical sample is gathered.

A single C program (included on the companion website) has been devised for segmenting texture based on the preceding measures and on all of the methods to be discussed in later sections. It is called simply `text.c` and will produce an output image that has been transformed by whatever segmentation algorithm is specified on the command line. The general way that this program is invoked is by:

```
text image.pgm KEYWORD
```

where `KEYWORD` is replaced by the name of one of the texture segmentation algorithms. For example, to use the average grey level over a window the program would be called as:

```
text image.pgm grey_level
```

If not executing from a shell, then the program will ask for the parameters from standard input. In that case, the exchange looks like this:

```
text
No command line args - asking for input.
Enter input file name:                  t1.pgm
Enter texture measure:                  grey_level
      ...
```

The result in this case is a collection of image files for windows of size 13×13, 21×21, 29×29, 27×27 and 45×45; these files are in PGM format and are named `txt6.pgm`, `txt10.pgm`, `txt14.pgm`, `txt18.pgm`, and `txt22.pgm`, respectively. The number in the file name refers to the number of pixels on each side of the pixel in the middle of the window. For the example above, each pixel in the resulting images represents the average level found in the window centered at that pixel. The other keywords for the simple statistical measures are `sd_level` (standard deviation), `kurtosis`, and `skewness`.

As more methods for texture segmentation are explored, these will be added to the `text.c` program, and the relevant keywords and other parameters needed will be explained. If the program is invoked without any parameters, a short listing of the legal keywords will be printed on the screen as a reminder.

5.3 Grey-Level Co-Occurrence

The statistical texture measures described so far are easy to calculate, but do not provide any information about the repeating nature of the texture. For example, the texture seen on the left of Figure 5.1a consists of repeating

horizontal lines. None of the measurements examined so far will reflect this property, and so a set of vertical lines with the same widths and separations will be indistinguishable from this texture, as would suitably constructed diagonal lines.

A *grey level co-occurrence matrix* (GLCM) contains information about the positions of pixels having similar grey-level values. The idea is to scan the image and keep track of how often pixels that differ by Δz in value are separated by a fixed distance d in position. Normally the direction between the two pixels is also a concern, and this is implemented by having multiple matrices, one for each direction of interest. Usually, there are four directions: horizontal, vertical, and the two diagonals. Therefore, for every value of d we have four images, each of which is 256×256 in size, for an original image with 256 levels of grey. This is really too much data, often more than there is in the original image. What is usually done is to analyze these matrices and compute a few simple numerical values that encapsulate the information. These values are often called *descriptors*, and eight of them will be examined here.

Collecting the value for the co-occurrence matrices is not especially difficult, but it is time-consuming. Consider the problem of determining the GLCM for an image I having 256 distinct levels of grey, in the horizontal direction and for $d = 2$. This matrix will be called M0, and M0[i,j] will contain the number of pixels p1 and p2 in I for which p1 $= i$ and p2 $= j$ where p1 and p2 are separated by two pixels horizontally. The indices for M0 are grey levels, not rows or columns of the image.

This could be calculated in the following manner. Examine all pixels in the image I:

```
for (y=0; y<Nrows; y++)
      for (x=0, x<Ncolumns; x++)
      {
```

Let p1 be the grey level at pixel I[y,x] and let p2 be the grey level at the pixel $I[y,x + d]$:

```
p1 = I[y] [x]; p2 = I[y] [x+d];
```

These two levels are used as indices into the matrix M0 being constructed; increment the entry in M0.

```
M0 [p1] [p2] += 1;
```

Since M0 is symmetrical, we could also increment the symmetrical element:

```
M0 [p2] [p1] += 1;
```

or we could merge the upper and lower triangles after M0 is constructed. When all of the pixels have been examined by this loop, the matrix M0 is complete.

A final pass through M0 is needed if normalized values are wanted. Dividing the elements of M0 by the number of pixels involved gives joint probabilities; specifically M0[*i,j*] will be the probability that a pixel having grey level *i* will have a pixel with level *j* a distance of *d* pixels away in the horizontal direction. A very similar process will create the other three matrices.

The artificial textures of Figure 5.1a and 5.1b are ideally suited to be an example, since they are bi-level images: The color black has level 0, and the color white has level 255. If the 255 pixels are set to 1, then the GLCMs will be 2×2 matrices! For the horizontal line texture on the left of Figure 5.1a the line separation is 2 pixels; the co-occurrence matrices for $d = 1$ are:

HORIZONTAL (0)	VERTICAL (90)
0.5000 0.0000	0.2468 0.2532
0.0000 0.5000	0.2532 0.2468

and the diagonal matrices are the same as the vertical. These results make sense; since the lines are horizontal, there will be no black-to-white level changes in the horizontal direction. If you start on a black pixel, its horizontal neighbor will be black (Mat0[0][0] = 0.5) and from any white pixel the horizontal neighbor will be white (Mat0[1][1] = 0.5). For $d = 2$ the matrices are more interesting:

HORIZONTAL (0)	VERTICAL (90)
0.5000 0.0000	0.0000 0.5000
0.0000 0.5000	0.5000 0.0000

where, again, the diagonal matrices are the same as M90. This means that, no matter what the color of the pixel we start at, its neighbor two pixels away horizontally is the same color, and the neighbor two pixels away vertically is always the opposite color. This is an accurate characterization of the horizontal line texture.

As the number of grey levels in the image increases by a factor of 2, the co-occurrence matrices increase in size by a factor of 4. It very quickly becomes difficult to use the matrices directly, and so once again the use of statistics becomes important. Rather than measure the properties of the image directly, the co-occurrence matrices are measured and, as mentioned already, characterized by a selection of numerical descriptors. The mean and standard deviation have been used as descriptors for the image data, and could be used for the co-occurrence matrices as well. However, many descriptors have been tried, and some work better than others. Five of the more popular ones follow; then each one will be tested on the sample textures of Figure 5.1.

5.3.1 Maximum Probability

This is simple the largest entry in the matrix, and corresponds to the strongest response or the most likely transition. This could be the maximum in any of the matrices, or the maximum overall; in fact, there is useful information in simply knowing which matrix contains the maximum, since this will indicate an important direction for the texture being examined.

5.3.2 Moments

The order *k element difference moment* can be defined as:

$$Mom_k = \sum_i \sum_j (i-j)^k M[i,j] \qquad \text{(EQ 5.4)}$$

This descriptor has small values in cases where the largest elements in M are along the principal diagonal; this was the situation when analyzing Figure 5.1a. The opposite effect can be achieved using the *inverse moment*, which is computed as:

$$Mom_k^{-1} = \sum_i \sum_j \frac{M[i,j]}{(i-j)^k}, i \neq j \qquad \text{(EQ 5.5)}$$

5.3.3 Contrast

An estimate of contrast is given by the following expression:

$$C(k,n) = \sum_i \sum_j |i-j|^k M[i,j]^n \qquad \text{(EQ 5.6)}$$

When $k = n = 1$, which is the situation implemented by the software on the website accompanying this book, this amounts to the expected value of the difference between two pixels.

5.3.4 Homogeneity

This value is given by:

$$G = \sum_i \sum_j \frac{M[i,j]}{1+|i-j|} \qquad \text{(EQ 5.7)}$$

A small value of G means that the large values of M lie near the principal diagonal. G is very similar to Mom_1^{-1}.

5.3.5 Entropy

Entropy is calculated by:

$$H = \sum_i \sum_j M[i,j] \log(M[i,j]) \qquad \text{(EQ 5.8)}$$

This is a measure of the information content of M. Large empty (featureless) spaces have little information content, whereas cluttered areas have a large information content.

5.3.6 Results from the GLCM Descriptors

The software that actually segments images using GLCM descriptors is very slow; one matrix must be computed for each window, followed by a calculation of the value for the descriptor — this yields a single pixel in the segmented image. Because of this, experimentation should be done using small images, smaller than the images in Figure 5.1. Each of these test images has been subjected to all five of the descriptors, and samples of the resulting segmented images appear in Figure 5.4.

The `text.c` program will compute the co-occurrence values described here. However, in addition to a keyword indicating which measure to apply, the co-occurrence calculation needs to be given a distance and a direction. These are specified after the keyword and in that order; distance followed by direction. Distance is specified as the number of pixels; direction is an integer, having the value 0 for horizontal, 1 for 45 degrees, 2 for 90 degrees, and 3 for 135 degrees.

So, in order to segment the image `t3.pgm` using the entropy metric applied to the co-occurrence matrix for a distance of two pixels in the horizontal direction the command would be:

```
text t3.pgm entropy 2 0
```

5.3.7 Speeding Up the Texture Operators

Creating a co-occurrence matrix for each window is not only slow and memory-intensive in practice, but is not even necessary; it is not the matrix that is needed, but some measure (descriptor) that will be used to characterize it. Unser [1986] devised a method for computing the statistics without actually creating the matrix.

The method is based on sum and difference histograms. The sum histogram S depends on displacements d_x and d_y, and is the histogram of the sums of all pixels d_x and d_y apart. For example, the pixel at (i,j) will be added to the

pixel at $(i + d_y, j + d_x)$ and the histogram bin corresponding to the sum is incremented. For a 256-level image, the sum histogram has bins 0 through 512. The difference histogram D is merely the histogram of the differences between pixels the specified distance apart. Histogram D has bins −255 through +255 for an eight-bit image. Now the histograms S and D are normalized so that the entries become probabilities.

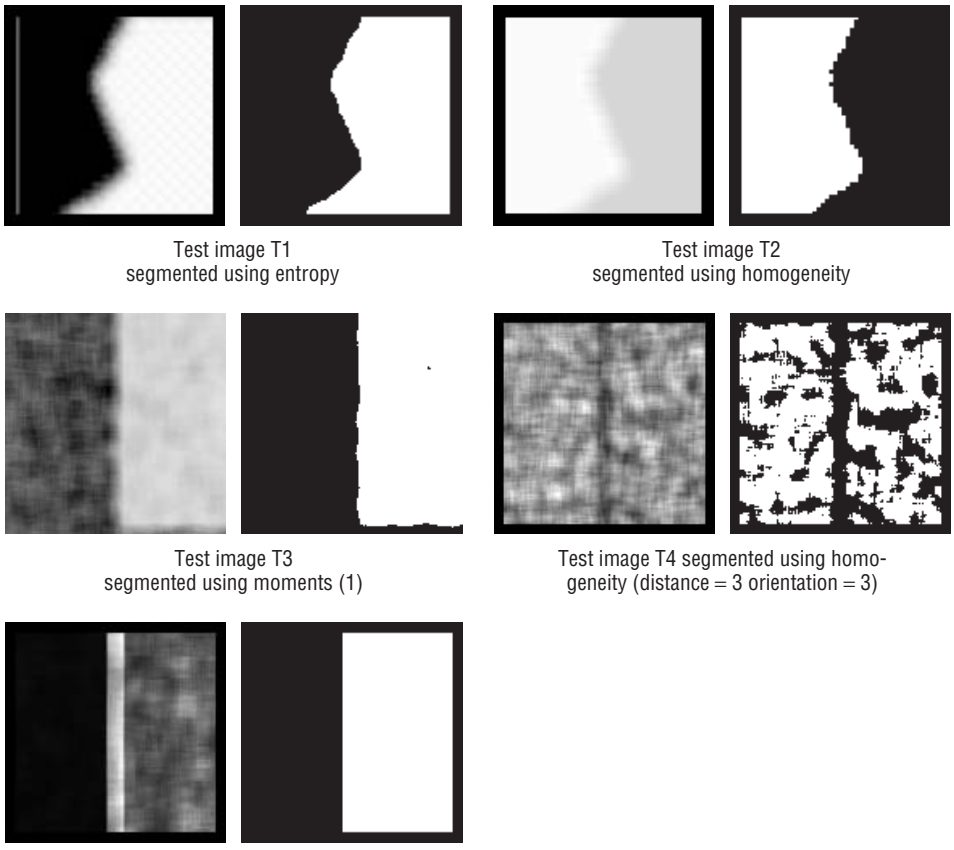


Figure 5.4: Co-occurrence texture measures applied to the set of test images of Figure 5.1. In all cases but one, the distance was 2 and the direction was horizontal.

The more common descriptors used to characterize co-occurrence matrices can be approximated using only these two histograms. This is much more efficient, since not only are the actual matrices not calculated, but the descriptors are now computed using two small one-dimensional arrays instead of a

large two-dimensional array. Some of the descriptors that are efficiently found using sum and difference histograms are:

$$\mu = \frac{1}{2}\sum_i i \cdot S(i) \qquad \text{Mean}$$

$$\sum_j j^2 \cdot D(j) \qquad \text{Contrast}$$

$$\sum_j \frac{1}{1+j^2} \cdot D(j) \qquad \text{Homogeneity}$$

$$-\sum_i S(i) \cdot \log(S(i)) - \sum_j D(j) \cdot \log(D(j)) \qquad \text{Entropy}$$

$$\sum_i S(i)^2 \cdot \sum_j D(j)^2 \qquad \text{Energy}$$

While these are approximations, they are good enough to be quite useful, and the speed up in the code is at least a factor of twenty. The C code that accomplishes this is called `fast.c`, and while perhaps not pretty, it enables texture analysis to be performed on a PC. Without the speedup, the use of co-occurrence matrices is not practical on small computers, such as single processor PCs.

`Fast.c` accepts almost the same parameters as does `text.c` when using the co-occurrence measures. It recognizes the keywords `average` (mean of the window), `stddev` (standard deviation in the window), `pmax` (maximum value), `contrast` (as before), `homo` (homogeneity, as before), and a new measure, energy, which is defined by the last definition in the list above. Most of the work is done by the procedure `sdhist`, which is given in Figure 5.5. This function is somewhat more opaque than the usual code shown here, but illustrates some of the methods that can be used to speed up image-analysis code. In particular, note the absence of any two-dimensional array references.

5.4 Edges and Texture

If a collection of objects called textons forms a texture, then it should be possible to isolate individual textons and treat them as objects. It should be possible to locate the edges that result from the grey-level transitions along the boundary of a texton. Moreover, since a texture will have large numbers of textons, there should be some property of the edge pixels that can be used to characterize the texture; this property may be a set of common directions, distances over which the edge pixels repeat, or simply a measure of the local density of the edge pixels.

```
/* Compute the sum and difference histograms, and the mean */

void sdhist (IMAGE im, int d, WINDOW *w)
{
        int ngl=256,p1=0,p2=0,i=0,j=0,k=0,l=0,r=0,t=0,b=0,id=0,nc;
        static float *Ps, *Pd;
        float sum=0.0;
        unsigned char *ptr1, *ptr2;

        nc = im->info->nc;

/* Allocate the matrix */
        if (Ps == 0)
        {
                Ps = (float *)calloc (ngl*2, sizeof(float));
                Pd = (float *)calloc (ngl*2, sizeof(float));
                Sum = Ps;
                Diff = Pd;
        }
        dir = (int)param[4];
        l = w->left; r = w->right; t = w->top; b = w->bottom;

/* Compute the histograms for any of 4 directions */
        ptr2 = im->data[t];
        for (i = t; i < b; i++)
        {
                ptr1 = ptr2+l;   id = d*nc;
                for (j = l; j < r; j++)
                {
                        p1 = *ptr1;
                        if (j+d < r && dir == 0)
                                p2 = *(ptr1+d);    /* Horizontal */
                        else if (i+d < b && dir == 2)
                                p2 = *(ptr1+id);           /* Vertical */
                        else if (i+d < b && j-d >= l && dir == 1)
                                   /* 45 degree diagonal */
                                p2 = *(ptr1 +id - d);
                        else if (i+d < b && j+d < r && dir == 3)
                                   /*135 degree diagonal */
                                p2 = *(ptr1+id+d);
                        else { ptr1++; continue; }
                        k++, ptr1++; Ps[p1+p2]++; Pd[p1-p2+ngl]++;
                }
                ptr2 += nc;
        }

/* Normalize */
        for (i=0; i<ngl+ngl; i++)
        {
                Ps[i] /= k;
                sum += Ps[i]*i;
                Pd[i] /= k;
        }
        Mean = sum/2.0;
}
```

Figure 5.5: Source code for the procedure that calculates the sum and difference histograms for a given window. The code has been designed to be relatively fast, rather than readable. Two-dimensional array references are absent, and most arrays are treated as pointers.

A perfect example of this is the test image Figure 5.1a. The leftmost texture consists of repeated horizontal lines; it is to be expected that a large number of edge pixels having direction 0° will be found in this region. The neighboring texture, on the other hand, consists only of diagonal lines, and should have few (if any) edge pixels in this direction. Thus, edge pixel direction can be used in this case to segment the two regions.

The density of the edge pixels is probably the simplest edge-based metric, and is easy to calculate. A fast edge detector is applied to a window, and the number of edge pixels in that window is divided by the area to give the density. Any abrupt change in the edge-pixel density likely marks a boundary between two regions. Since it is usually a simple matter to extract directional information from an edge detector, this can be used to augment the edge density. Easy measurements to make include the mean x and y component of the gradient at the edge and the relative number of pixels whose principal direction is x or y.

Actual angles are harder to deal with since, depending on the window size, there can be a larger number of different angles. The histogram of the angles found in a window could characterize the texture there, but to compare a large number of N-dimensional histograms would be computationally intense. However, the spatial relationships between the pixels having particular angles, and for that matter edge pixels in general, convey a great deal of information. Why not compute the co-occurrence matrix of an edge-enhanced image? This will, in many circumstances, give a better result, in terms of discriminating ability, than using the co-occurrence matrix without edge enhancement [Dyer 1980; Davis 1981].

Figure 5.6 illustrates the use of edge information and co-occurrence statistics. The original image shows a few trees, a cloudy sky, and part of a building. The contrast measure of the grey-level co-occurrence matrix for this image (5.6b) could give a classification, but enhancing the edges first (5.6c) increases the contrast in edge-prone regions, such as the trees. The same contrast measure gives somewhat better results (5.6d) and was used to produce the final classification (5.6e). The region containing clouds is quite well marked as black in this image.

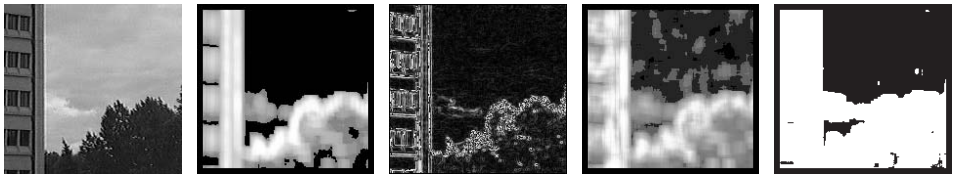


Figure 5.6: Using edges to enhance the results from co-occurrence matrices. (a) Original image. (b) Contrast of the co-occurrence matrix, distance 4 horizontal. (c) Edge enhanced image. (d) Contrast of the co-occurrence matrix of the edge image. (e) Classification based on (d).

Noise smoothing before edge detection will reduce the number of spurious edge pixels, and will give better results in some instances. A Laplacian edge detector is less directional, and could be used in place of the Sobel edge detector in the software provided. Some of the more advanced edge detectors (e.g., Canny or ISEF) actually smooth too well, and respond to texture as if it were noise. The contrast of the image may have to be increased greatly before one of these methods could be used.

The `text.c` program cheats a little with respect to edges. Running text with the `sobel` keyword will result in an edge-enhanced version of the original file being written to `sobtxt.pgm`. This file can then be used as input to `fast.c` or `text.c`, specifying the desired co-occurrence operator. For example, Figure 5.6d was created in the following way:

```
text sky.pgm sobel
FAST sobtxt.pgm contrast 4 0
```

When the angles of the edge pixels are desired, specify `sang` as the keyword. The result, again, is a file `sobtxt.pgm` containing the scaled angles. The keywords `dx` and `dy` specify direction components in the x and y directions, and `nx` and `ny` as keywords yield the number of edges in each window having the largest component in the x or y direction.

5.5 Energy and Texture

There are many ways in which the energy content of an image could be calculated, depending on how energy is defined and what kind of image is at hand. One measure of the energy of a textured region was seen in Section 5.3, where an energy measure was computed from a co-occurrence matrix. Laws [1980] devised a collection of convolution masks specifically for the purpose of computing the energy in a texture. These have been used successfully for many years for texture segmentation, and are now a standard for comparison for new algorithms.

Although various sizes for the masks are possible, three of the five pixel masks are:

$$\mathsf{E5} = (-1, -2, 0, 2, 1)$$

$$\mathsf{L5} = (1, 4, 6, 4, 1)$$

$$\mathsf{R5} = (1, -4, 6, -4, 1)$$

These can be used in combination to create nine different two-dimensional convolution masks. If they are treated as vectors, then E5×L5 gives a 5×5 matrix called E5L5 and having the value:

−1	−4	−6	−4	−1
−2	−8	−12	−8	−2
0	0	0	0	0
2	8	12	8	2
1	4	6	4	1

After the convolution with the specified mask, the energy is computed by:

$$E_n = \sum_r \sum_c \left| C(r,c) \right| \qquad \text{(EQ 5.9)}$$

where C is the convolved image. The size of the region used to determine the energy can vary; a 7×7 region seems to be quite common. If all nine masks that result from the combinations of E5, L5, and R5 are applied, the result is a nine-dimensional feature vector at each pixel of the image being analyzed. These vectors can be used with a statistical classifier (e.g., K nearest neighbor). In some cases only one or two of the energy values are sufficient.

At the boundary between two regions having different textures, this method (as well as many of the others we have seen) does not perform very well. The pixels near the boundary form a region of high variability that, statistically, has some properties of both adjoining regions. Sometimes this shows up in images as a thick black or white bar separating the regions, while other times it can be thresholded into one of the textures.

Something that has been suggested for use with the texture energy method specifically, but that has more general application, is to look carefully at the areas to the upper-left, upper-right, lower-left and lower-right of the window being processed. At the interior of a region these four areas should have statistically similar properties, but at the boundary between regions there will be variations observed. When this occurs, select the energy value for the region having the *smallest standard deviation*, this being most likely to be representative of the interior.

All nine of these energy operators are implemented by the `text.c` program. To apply an energy operator to an image, specify it as the keyword; for example,

```
text input.pgm E5E5
```

will apply the `E5E5` operator to `image.pgm`, producing a file named `txtN.pgm`, where N is 6, 10, 14, 18, or 24, as before. The program creates the correct

convolution mask and applies it to the input image, then determines the mean energy value for the window. The letters in the keyword must be specified in upper case. Figure 5.7 shows the result of each of the nine masks applied to test image `t4.pgm`.

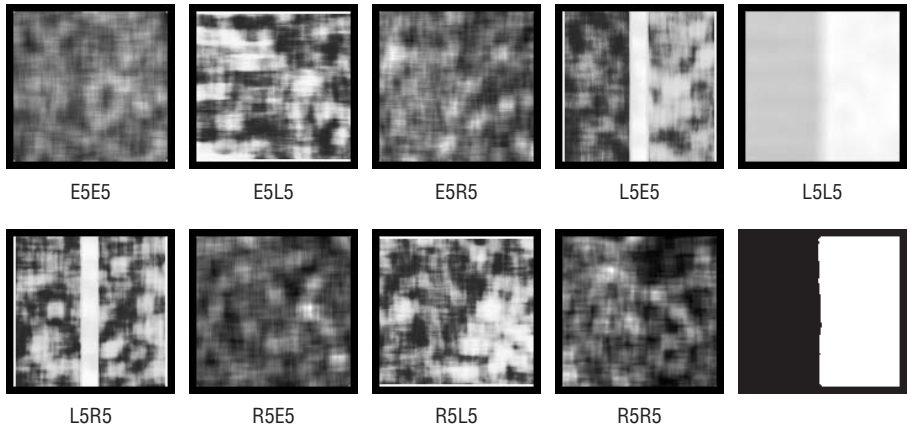


Figure 5.7: All energy convolution masks applied to the test image T4. The final image is the segmented version, using L5L5.

5.6 Surfaces and Texture

Some texture segmentation algorithms are based on a view of the grey-level image as a three-dimensional surface, where grey level is the third dimension. Depending on what sort of assumptions that are made concerning the nature of this surface, any number of descriptors can be devised. Two will be examined here, but the references give a few good pointers to other useful and interesting work.

5.6.1 Vector Dispersion

For the purposes of this algorithm, the texture image consists of a set of small planes, or facets. Each plane is really a small area of the image. The normal to each plane is a vector, and for a region having many facets the variation in the direction of the normals may produce a measure that can characterize the texture in that region.

Figure 5.8 shows a 3×3 region of a grey-level texture, in which the grey level is treated as a third dimension. The facets meet to form an edge every three

pixels both horizontally and vertically, and do not overlap. Since a plane is a linear equation in two dimensions, it can be written in the form:

$$I(i,j) = \alpha i + \beta j + \gamma \qquad \text{(EQ 5.10)}$$

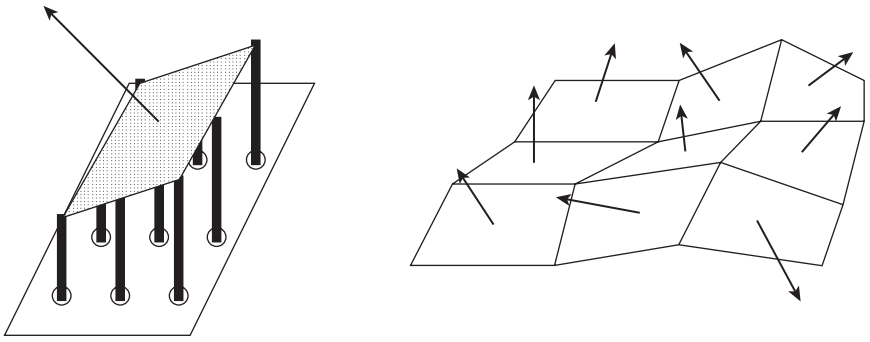


Figure 5.8: Vector dispersion. (a) Each facet is a small plane having an easily computed normal direction, indicated by the arrow. The posts supporting the facet represent the grey level at each of the nine pixels. (b) Texture metrics are computed over a local collection of facets, and represent how the directions of the normals are distributed about the local mean normal.

for image I over a small (3×3) area. The coefficients are easy to find using a least-squares best fit of a plane to the levels found in the small region of I. Details can be found in the math book of your choice, but the result is:

$$\alpha = \frac{\sum_{i=-1}^{1}\sum_{j=-1}^{1} i \cdot I(i,j)}{\Sigma\Sigma i^2} \qquad \text{(EQ 5.11)}$$

$$\beta = \frac{\sum_{i=-1}^{1}\sum_{j=-1}^{1} j \cdot I(i,j)}{\Sigma\Sigma j^2} \qquad \text{(EQ 5.12)}$$

$$\gamma = \frac{\sum_{i=-1}^{1}\sum_{j=-1}^{1} I(i,j)}{\Sigma\Sigma 1} \qquad \text{(EQ 5.13)}$$

The coefficients α, β, and γ can be thought of as a vector. The normal vector is perpendicular to the plane, and the normalized form (i.e., length = 1) for the ith facet is:

$$\begin{bmatrix} K_i \\ L_i \\ M_i \end{bmatrix} = \frac{1}{\sqrt{\alpha_i^2 + \beta_i^2 + 1}} \begin{bmatrix} \alpha_i \\ \beta_i \\ -1 \end{bmatrix} \qquad \text{(EQ 5.14)}$$

An estimate of the direction of the surface normal over the whole region can be obtained from the facet normals by simply averaging them. If it is assumed that the individual normals are measurements taken from the surface of a sphere, then the statistical distribution of the errors is related to $e^{\kappa COS\theta}$, where θ is the error in the angle [Fish 1952]. The value κ acts as a measure of precision, but in the case of the measurement of vector dispersion, a large κ value indicates a smooth texture, and values near zero indicate a rough texture. Given a set of normal vectors, the value of κ can be estimated by

$$\kappa = \frac{N-1}{N-R} \qquad \text{(EQ 5.15)}$$

where R is found from the normal vectors:

$$R^2 = \left(\sum_{i=1}^{N} K_i\right)^2 + \left(\sum_{i=1}^{N} L_i\right)^2 + \left(\sum_{i=1}^{N} M_i\right)^2 \qquad \text{(EQ 5.16)}$$

This leaves us with the following algorithm:

1. For a given window into the image I, locate some number of non-overlapping subregions.
2. For each subregion compute the coefficients of the plane, and from that compute the normal to the plane for that subregion; this is a vector (K_i,L_i,M_i).
3. Normalize the vectors from step 2, and then compute R using the vectors from all subregions.
4. Compute κ. This is the texture descriptor for this window. Repeat from step 1 for all windows in the image.

As usual, the κ value for the window centered at a pixel P will be the value of the pixel P in the segmented image. Thresholding the segmented image will still be needed.

Figure 5.9 shows this method applied to the test images seen in Figures 5.1c and 5.1e. When using the `text.c` program, simply specify the keyword `vd`. Figure 5.9 was created using the call:

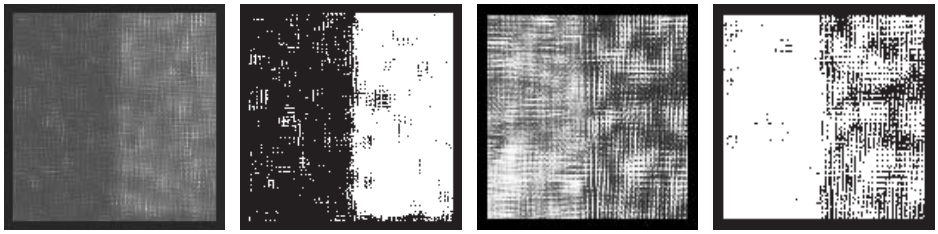
```
text t3.pgm vd
```

5.6.2 Surface Curvature

The vector-dispersion method fits a plane to the pixels in a small area, which is really a first approximation to what is really there. A better approximation

would be a polynomial surface, which can better conform to local variations in shape. A typical second-degree polynomial surface is defined by:

$$z(x,y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \quad \text{(EQ 5.17)}$$



(a) (b) (c) (d)

Figure 5.9: Segmentation of texture regions using vector dispersion. (a) Vector dispersion image obtained from Figure 5.1c. (b) Thresholded version, showing the two texture regions. (c) Vector dispersion image obtained from Figure 5.1e. (d) Thresholded version.

Curvature can be defined as the rate of change of the slope of the tangent to the surface at the point in question. Given this local approximation to the surface, there are a number of surface curvature measures that might be useful for characterizing the surface (and therefore the texture) at that point. This would involve the following steps for each pixel in the region being evaluated:

1. Use least squares to fit a polynomial surface to the local pixel region.
2. Compute the derivatives of the surface at the specified point.
3. Use the derivatives, the slope of the tangent, to compute curvature.

Fortunately, this has been done for us by Peet and Sahota [Peet 1985]. The values of a_{ij} are found in much the same way as were the values of α, β, and γ for vector dispersion. For a 3×3 region centered at $I(i,j)$ we have:

$$A_1 = \sum_{n=-1}^{1} \sum_{m=-1}^{1} I(i+n, j+m)$$

$$A_2 = \sum_{n=i-1}^{i+1} I(n, j+1) - \sum_{n=i-1}^{i+1} I(n, j-1)$$

$$A_3 = \sum_{m=j-1}^{j+1} I(i-1, m) - \sum_{m=i-1}^{j+1} I(i+1, m)$$

$$A_4 = \sum_{n=i-1}^{i+1} I(n, j-1) + \sum_{n=i-1}^{i+1} I(n, j+1)$$

$$A_5 = \sum_{m=j-1}^{j+1} I(i-1, m) - \sum_{m=j-1}^{j+1} I(i+1, m)$$

$$A_6 = I(i-1, j+1) + I(i+1, j-1) - I(i-1, j-1) - I(i+1, j+1)$$

The coefficients of the polynomial can now be expressed in terms of the A_i values:

$$\begin{aligned} a_{20} &= A_4/2 - A_1/3 \\ a_{11} &= A_6/4 \\ a_{02} &= A_5/2 - A_1/3 \\ a_{10} &= A_2/6 \\ a_{01} &= A_3/6 \\ a_{00} &= 5A_1/9 - A_4/3 - A_5/3 \end{aligned}$$

This is the least-squares fit of the surface to the data. The curvature can be calculated given only a little more algebra motivated by differential geometry, but it is really quite simple to compute. The following values are parameters to the *first and second fundamental forms* of the surface:

$$\begin{aligned} E &= 1 + a_{10}^2 \\ F &= a_{10}a_{01} \\ G &= 1 + a_{01}^2 \\ e &= (2a_{20})/\sqrt{EG - F^2} \\ f &= (2a_{11})/\sqrt{EG - F^2} \\ g &= (2a_{02})/\sqrt{EG - F^2} \end{aligned}$$

The minimum curvature at the point $I(i,j)$ is given by:

$$k_1 = \frac{gE - 2Ff + Ge - \sqrt{(gE + Ge + 2Ff)^2 - 4(egg - f^2)(EG - F^2)}}{2(EG - F^2)} \quad \text{(EQ 5.18)}$$

and the maximum curvature is:

$$k_2 = \frac{gE - 2Ff + Ge - \sqrt{(gE + Ge + 2Ff)^2 - 4(egg - f^2)(EG - F^2)}}{2(EG - F^2)} \quad \text{(EQ 5.19)}$$

The *Gaussian curvature* is defined as the product of k_1 and k_2; that is, $k_3 = k_1 \times k_2$. The mean curvature k_4 is simply $(k_1 + k_2)/2$. Peet and Sahota define two other curvature measures, claimed to be better than $k_1 \ldots \ k_4$:

$$\begin{aligned} k_5 &= \frac{(k2 - k1)}{2} \\ k_6 &= \max(|k_1|, |k_2|) \end{aligned} \quad \text{(EQ 5.20)}$$

Finally, from the sign of the expression $eg - f^2$, we can determine whether the point under consideration is a *saddle point* (<0), an *elliptic point* (>0) or a *parabolic point* (=0). The number of such points in each image window might provide some texture discrimination capability.

There are nine texture measures based on this discussion of surface curvature. Not all of these will be useful when applied to any particular image, but they are useful tools to have available. The `text.c` program implements each of these in the usual way; the algorithm names for the command line are: k_1, k_2, k_3, k_4, k_5, k_6, elliptic, parabolic, and saddle. Thus, the call

```
text t3.pgm saddle
```

will compute the number of saddle points in each window and create an image in which these values are the levels associated with each pixel. Figure 5.10 shows some of these measures applied to the test image.

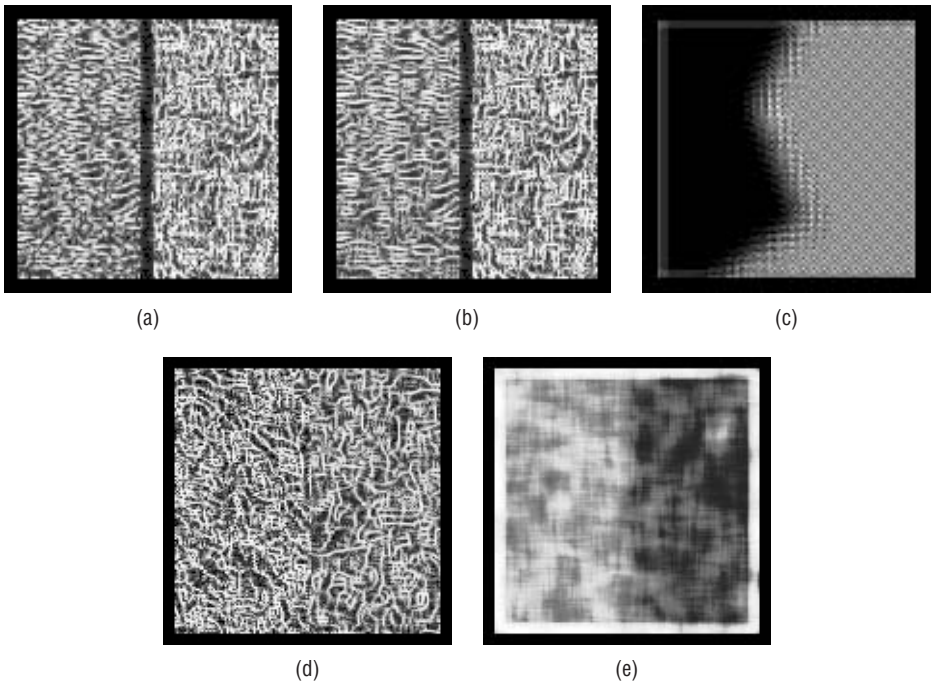


Figure 5.10: Surface curvature measures. (a) K5 measured on test image T5. (b) K6 measured on T5. (c) Elliptic points found in windows in image T1. (d) K6 measured on T3. (e) Standard deviations of 13×13 windows of image (d).

5.7 Fractal Dimension

Fractal geometry can be used on occasion to discriminate between textures. The word ''fractal'' is really more of an adjective than a noun, and it refers to entities (especially sets of pixels) that display a degree of self-similarity at different scales. A mathematical straight line displays a high degree of self-similarity; any portion of the line is the same as any other, at any magnification.

The fractal dimension D of a set of pixels I is specified by the relationship:

$$1 = Nr^D \qquad \text{(EQ 5.21)}$$

where the image I has been broken up into N non-overlapping copies of a basic shape, each one scaled by a factor of r from the original. It might be possible to measure D given a perfect synthetic image, but natural scenes with textures will not contain exact replicas of the basic shape. What we want is an estimate of D that can be calculated from a sampled raster representation. One such algorithm is the Differential Box Counting (DBC) algorithm [Sark, 1992], and another uses the Hurst coefficient [Russ, 1990]. Equation 5.21 can be rewritten as

$$D = \frac{\log N}{\log\left(\frac{1}{r}\right)} \qquad \text{(EQ 5.22)}$$

From this it can be seen that there is a log-log relationship between N and r. If $\log(N)$ were plotted against $\log(r)$ the result should be a straight line whose slope is approximately D.

The Hurst coefficient is an approximation that makes use of this relationship. Consider Figure 5.11, in which a 7×7 pixel region is marked according to the distance of each pixel from the central pixel. There are eight groups of pixels, corresponding to the eight different distances that are possible. Within each group the largest difference in grey level is found; this is the same as subtracting the smallest grey level in the group from the largest.

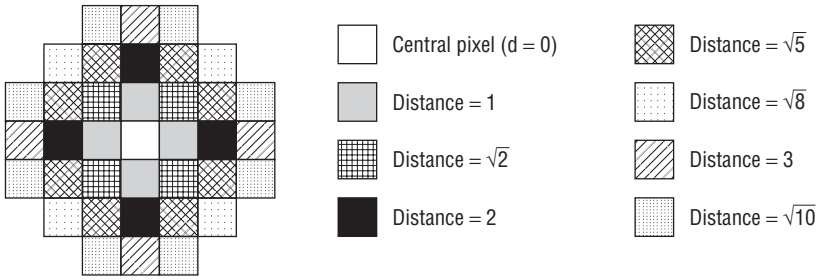


Figure 5.11: A 7×7 region for calculating the Hurst coefficient of the central pixel. There are eight classes of pixel, organized by their distance from the central pixel. The pixel at d = 0 is not really a class by itself.

The central pixel is ignored, and a straight line is fit to the log of the maximum difference (y coordinate) and the log of the distance from the central pixel (x coordinate). The slope of this line is the Hurst coefficient, and replaces the pixel at the center of the region.

As an example, consider the following 7×7 pixel region of an image:

85	70	86	92	60	102	202
91	81	98	113	86	119	189
96	86	102	107	74	107	194
101	91	113	107	83	118	198
99	68	107	107	76	107	194
107	94	93	115	83	115	198
94	98	98	107	81	115	194

The first step in computing the Hurst coefficient is to determine the maximum grey-level difference for each distance class of pixels. Starting at the pixels at distance one or less from the center, the maximum level is 113 and the minimum is 83, for a difference of 30. The next class has the range $113 - 74 = 39$, and the distance = 2 class has a range of $118 - 74 = 44$.

Completing Table 5.1 gives:

Table 5.1: Hurst Coefficient Example Data

CLASS:	d = 1	d = $\sqrt{2}$	d = 2	d = $\sqrt{5}$	d = $\sqrt{8}$	d = 3	d = $\sqrt{10}$
NUMBER:	30	39	44	50	51	130	138

A line to be fit to this data using a log-log relationship, so the next step is to take the log of both the distance and the grey-level difference, as shown in Table 5.2:

Table 5.2: Hurst Coefficient Calculation Data

LN(DISTANCE):	0.000	0.347	0.693	0.805	1.040	1.099	1.151
LN(DELTA G):	3.401	3.664	3.784	3.912	3.932	4.868	4.927

Now a straight line is fit to the points, using a least-squares approach. The line in this case has the equation:

$$y = 1.145x + 3.229$$

The slope of this line, $m = 1.145$, is the Hurst coefficient. The graph of the raw data and the fit line can be seen in Figure 5.12.

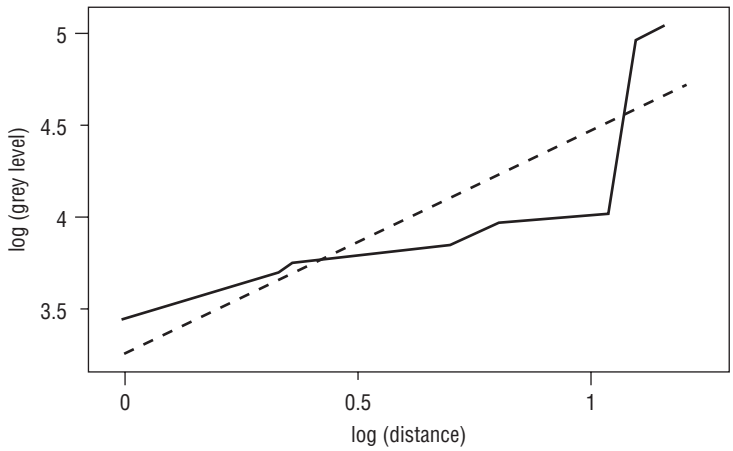


Figure 5.12: The straight-line fit to the log (distance) vs. log (grey-level change) data for the 7×7 region used as an example of the Hurst coefficient calculation. The slope of this line is the Hurst coefficient.

Fractal dimension can be estimated using the `text.c` program as follows:

```
text tl.pgm fractal
```

The resulting image will be found in the file `fractal.pgm`.

5.8 Color Segmentation

The basic RGB color model used in most digital image formats is not especially suitable for anything except image display. In particular, there are problems in segmentation, when compared to the use of grey-level thresholds. Consider that RGB values could be stored in a single computer word with eight bits of the blue component in the bottom of the work, followed by eight bits of green, and finally the red part. Any single threshold would really only cut one of the color channels into two, leaving the others intact. Moreover, if one component, let's say blue, were at full intensity (255), then adding 1 to the pixel means a vast change in color; there is now *no* blue and more green (numerically similar colors can be quite different perceptually). Treating color as three orthogonal coordinates seems a better idea, but the increased dimensionality now complicates matters.

An idea that has been used with success is to reduce the number of colors to a very few intense ones, colors that will be quite clearly distinct from each other. The resulting image will look like a cartoon, with a few highly contrasting colors. Each color can be given a distinct number or index, and each will

represent a segmented region. The way the image looks is really moot, as the goal is to identify these regions. This is called *color quantization*.

The simplest method for color quantization involves an *a priori* decision about how many regions will be created. This is normally not acceptable, since the number of regions or objects in the image is not known ahead of time. However, this does allow a clear illustration of what color quantization is and how it might be done. If the RGB values can be thought of as coordinates on three axes, each running from 0 to 255. If each is divided into two parts at the center (128) then eight volumes are created in this color space. Each of these volumes could be represented graphically by the RGB coordinates (color) at the center of the volume, meaning that there will be eight colors in total. All pixels in an image can now be replaced by one of these eight prototype colors. This method is called *uniform* quantization.

A pixel is an RGB vector in this scheme: $P_{ij} = (R_{ij}, G_{ij}, B_{ij})$. It is changed to become equal to the nearest prototype color, where nearest can mean many things but is usually Euclidean distance. So, the distance between P_{ij} and all prototype colors $Q_k = (R_k, G_k, B_k)$ is computed, and the Q_k having smallest distance replaces the color values in the pixel P_{ij}. That is, we compute

$$d_k = \sqrt{(R_{ij} - R_k)^2 + (G_{ij} + G_k)^2 + (B_{ij} + B_k)^2}$$

and choose prototype k for which d is a minimum.

Using eight colors was arbitrary, of course, and some images will need fewer, some more. The method is also naive in that the volumes associated with the prototype colors are the same, and are unconnected with the frequency of appearance of actual colors in the image.

If the nature of the images is known in advance, it is possible to select colors that would be especially useful for segmentation. For instance, the flower image in Figure 5.13 has a bright yellow flower, some green foliage, white twigs, and red berries. Selecting prototypes for each of these colors could lead to better segmentation.

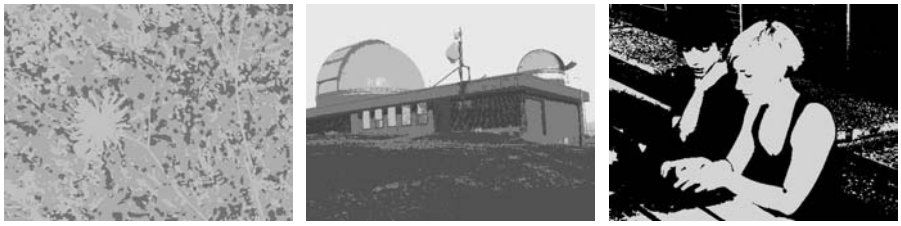
Figure 5.14 illustrates the use of hand-selected prototypes. This method can be used in situations where the type of image is relatively stable in terms of the objects within, and the colors are thus fairly consistent: visual inspection, surveillance, and so on.

A better method would use the actual data in the image — the colors being used — to determine what the prototype colors should be. The *popularity algorithm* does just that. Based on the colors that actually occur in an image, the algorithm chooses the N most common as prototype colors. This means building a binned histogram of the frequency of occurrence of colors, and this is often based initially on a fixed grid imposed on the color axes. For example, use histogram bins that break each axis into 6 parts; this gives 6x6x6, or 216 bins. Each bin contains a range of 43 color values in one of the three channels. Then N of these are chosen as prototypes, and each pixel is assigned the nearest prototype color. N is selected based on the number of distinct object types that

can be expected to appear in the image. The results, as shown in Figure 5.15, are better than those from uniform quantization, but the images need further processing if coherent regions are to be identified.

The *median cut* algorithm divides each color axis at the median color, and can then repeat that process as often as wanted to give the desired number of prototype colors. This method, too, uses colors that actually occur in the image as a basis for building the prototypes.

Figure 5.13: (Top) Sample images for color quantization. Results of simple uniform quantization.



(a) (b) (c)

Figure 5.14: The use of pre-selected color prototypes in uniform quantization. (a) Flower segmented with yellow, green, red, and white. (b) Observatory segmented with blue, grey, green, and brown. (c) Students segmented with black and flesh tone.

Figure 5.15: Images of Figure 5.13 after the popularity algorithm has been applied.

The use of RGB works alright, but the nature of the colors in an image is really contained in the hue information. Extracting and using the hue from an image can be advantageous in color segmentation because the effect of intensity is reduced. The OpenCV system can convert an image into hue/saturation/intensity, or HSV, very easily through the `cvCvtColor` function

```
cvCvtColor( image1, image2, CV_BGR2HSV );
```

which converts `image1`, which is RGB, into `image2`, which is expressed in HSV.

All hue components can be extracted by the `cvSplit` function:

```
cvSplit(image, hue, sat, val, 0);
```

giving three images, one for each component of the pixels. The hue image can be used for segmentation as if it were a set of grey levels.

As an example, consider the ''students'' image of Figure 5.13c. Converting it into hue yields a complex grey-level image that needs further segmenting, as shown in Figure 5.16a. However, after applying the popularity algorithm to the hue image, recognizable regions appear, and if some smoothing is done to reduce variation within regions, a clearly segmented image can be created (Figure 5.16c). Smoothing over a greater area will result in larger homogeneous regions.

(a) (b) (c)

Figure 5.16: (a) Hue component of Figure 5.13c. (b) Popularity algorithm applied to hue. (c) Popularity applied to smoothed hue image.

There are a large number of philosophically diverse methods in the literature for classifying color pixels. Most recently, a fuzzy C-means algorithm (a clustering scheme) that uses an ''ant colony'' optimization scheme has yielded some excellent results [Yu, 2010]. This work also illustrates a relatively recent development in vision research - the use of standard image sets with ground truth. Specifically, Yu's group tested their method using the Berkeley Image Segmentation Dataset [Martin, 2001]. A second example would be the Prague texture database [Haindl, 2006]. The existence and use of data means that the

results can be compared directly against any others based on that data set. This is a great idea, because in the past results have been applied to whatever images happened to be lying about or, worse, to whatever ones the method happened to work best on.

5.9 Color Textures

Images of natural scenes are usually rich in texture *and* color. Combine this with the fact that digital cameras are ubiquitous and are being used by many of us to fill our hard drives with inexpensive color images, and the need to include color and color textures in a segmenting scheme becomes obvious. But how? Textures are complex, and color along is more difficult to deal with than grey levels.

A key way do deal with color textures is to separate the problems of color and texture. Look for color areas irrespective of texture, and use the techniques discussed so far to look for texture in the hue image or the intensity (grey-level) image, and then merge the two. At some level, all color texture segmentation schemes do this.

For example, a practical system used for inspecting potato chips [Mendoza, 2007] applies entropy (Equation 5.8), contrast (Equation 5.6), energy and homogeneity (Equation 5.7) features to various color channels, such as intensity, hue, and a* (a component of the CIELAB color coordinate system [McLaren, 1976]) in order to determine the quality class of a chip. Even the relatively well-known *blobworld* scheme [Belongie, 1998] distinguishes explicitly between color and texture features, and combines three of each in determining its segmentation.

The reason for separating color and texture is clear: Color is a property of a single pixel, whereas texture is a property of a geometrically related collection of pixels. Not only are the two things separable, but not separating them results in a complexity due to high dimensionality that can be quite difficult to deal with. Rarely is there an advantage to increasing the dimension of a problem (although support vector machines are one such exception and will be discussed Chapter 8, ''Classification'').

5.10 Website Files

`fast.exe`	Fast texture code, command line
`fast1.exe`	Fast texture code, OpenCV

test1.exe	Standard texture code, OpenCV
fast.c	Source for fast texture library
fast1.c	Source for fast OpenCV texture library
lib.c	Source for image library
lib.h	Image include file
popularity.c	Popularity algorithm for color quantization
pupularity-hue.c	Popularity on hue channel
text1.c	Standard OpenCV texture library
text.c	Standard command-line texture library
uniform.c	Uniform color quantization
t1.pgm	Test image, Figure 5.1a
t2.pgm	Test image, Figure 5.1b
t3.pgm	Test image, Figure 5.1c
t4.pgm	Test image, Figure 5.1d
t5.pgm	Test image, Figure 5.1e

5.11 References

Ashlock, D., D. Zheng, and J. L. Davidson. ''Genetic Algorithms for Automatic Texture Classification.'' In *Statistical and Stochastic Methods in Image Processing II*, Proceedings of SPIE 3167 (1997): 140–151.

Belongie, S., C. Carson, H. Greenspan and J. Malik. ''Color and Texture-Based Image Segmentation Using EM and Its Application to Content-based Image Retrieval.'' *ICCV* (1998): 675–682

Chabrier, S., C. Rosenberger, B. Emile and H. Laurent. ''Optimization-Based Image Segmentation by Genetic Algorithms.'' *EURASIP Journal on Image and Video Processing* (2008): 1–10.

Christoudias, C., B. Georgescu, and P. Meer. ''Synergism in Low Level Vision.'' *Proceedings of the 16th ICPR* 4 (August 2002): 150–155.

Cohen, P., C.T. LeDinh and V. Lacasse. ''Classification of Natural Textures by Means of Two Dimensional Masks.'' *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, no. 1 (1989): 125–128.

Deng, Y. and B. Manjunath. ''Unsupervised Segmentation of Color-Texture Regions in Images and Video.'' *IEEE PAMI* 23, no. 8 (August 2001): 800–810.

Derin, H. and W. S. Cole. ''Segmentation of Textured Images Using Gibbs Random Fields.'' *Computer Vision, Graphics, and Image Processing* 35 (1986): 72–98.

Faugeras, O.D., ed. *Fundamentals in Computer Vision*. Cambridge: Cambridge University Press, 1983.

Ferryanto, S. and A. Kolmogorov-Smirnov. ''Type Statistic for Detecting Structural Changes of Textures.'' *Pattern Recognition Letters* 16 (1995): 247–256.

Ganesan, L. and P. Bhattacharyya. ''A New Statistical Approach for Micro Texture Description.'' *Pattern Recognition Letters* 16 (1995): 471–478.

Gong, M. and Yang, Y. H. ''Genetic-based Multiresolution Color Image Segmentation.'' In *VI'01*, Vision Interface Conference (Ottawa, Ontario, Canada, June 2001): 141–148.

Haindl, M. and Mikes, S. ''Model-Based Texture Segmentation.'' *LNCS* 3212 (2004): 306–313.

Haindl, M. and Mikes, S. ''Unsupervised Texture Segmentation Using Multi-Spectral Modelling Approach.'' *ICPR* (2) 2006: 203–206.

Haralick, R. M. and L. G. Shapiro. *Computer and Robot Vision*. Reading: Addison-Wesley, 1992.

Hsiao, J. Y. and A. A. Sawchuk. ''Unsupervised Image Segmentation Using Feature Smoothing and Probabalistic Relaxation Techniques.'' *Computer Vision, Graphics, and Image Processing* 48 (1989): 1–21.

Jin, X. C., S. H. Ong, and Jaysooriah. ''A Practical Method for Estimating Fractal Dimension.'' *Pattern Recognition Letters* 16 (1995): 457–464.

Julesz, B. ''Textons, the Elements of Texture Perception, and their Interactions.'' *Nature* 290, no. 12 (1981): 91–97.

Laws, K. I. ''Rapid Texture Identification.'' *SPIE Image Processing for Missile Guidance* (1980): 376–380.

Peet, F. G., and T. S. Sahota. ''Surface Curvature as a Measure of Image Texture.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7.6 (1985): 734–738.

McLaren, K. ''The Development of the CIE 1976 (L*a*b*) Uniform Colour-Space and Colour-Difference Formula.'' *Journal of the Society of Dyers and Colourists* 92 (1976): 338–341.

Martin, D., C. Fowlkes, D. Tal, and J. Malik. ''A Database of Human Segmented Natural Images and Its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics.'' Paper presented at the 8th International Conference on Computer Vision, 2001: 416–423.

Mendoza, F., P. Dejmek, and J. Aguilera. ''Colour and Image Texture Analysis in Classification of Commercial Potato Chips.'' *Food Research International* 40 (2007): 1146–1154.

Mikeš, S. and M. Haindl. ''Prague Texture Segmentation Data Generator and Benchmark.'' *ERCIM News* 64 (2006): 67–68.

Pratt, W. K. *Digital Image Processing*. 2nd ed. New York: John Wiley & Sons, 1991.

Rosenberger, C. and K. Chehdi. ''Unsupervised Segmentation of Multi-Spectral Images.,'' Paper presented at the International Conference on Advanced Concepts for Intelligent Vision Systems, Ghent, Belgium, September 2003.

Russ, J. C. ''Surface Characterization: Fractal Dimensions, Hurst Coefficients, and Frequency Transforms.'' *Journal of Computer Assisted Microscopy* 2 (1990): 249–257.

Sarkar, N. and B. B. Chaudhuri. ''An Efficient Differential Box Counting Approach to Compute Fractal Dimension of Image.'' *IEEE Transactions on Systems, Man, and Cybernetics* 24 (1994): 115–120.

Strouthopoulos, C. and N. Papamarko. ''Multithresholding of Mixed-Type Documents.'' *Engineering Applications of Artificial Intelligence* 13, no. 3 (2000): 323–343.

Tsai, C. M. and H. H. Lee. ''Binarization of Color Document Images via Luminance and Saturation Color Features.'' *IEEE Trans. Image Process* IP-11 (April 2002): 434–451.

Unser, M. ''Sum and Difference Histograms for Texture Classification.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986): 118–125.

Yoshimura, M. and S. Oe. ''Evolutionary Segmentation of Texture Using Genetic Algorithms Towards Automatic Decision of Optimum Number of Segmentation Areas.'' *Pattern Recognition* 32, no. 12 (1999): 2041–2054.

Yu, Z., O. Au, R. Zou, W. Yu, and J. Tian. ''An Adaptive Unsupervised Approach Toward Pixel Clustering and Color Image Segmentation.'' *Pattern Recognition* 43 (2010): 1889–1906.

CHAPTER

6

Thinning

6.1 What Is a Skeleton?

Everyone working in the computer vision field knows what thinning is: it is what you do to produce the *skeleton* of an object, usually a bi-level object. Reasonably enough, one might then ask, ''What is a skeleton?'' We now venture into the realm of opinion because, as with texture, there is no generally agreed-upon definition for what a skeleton is. Worse yet, and unlike with texture, we may not know a skeleton when we see it. This is unfortunate, because the generation of a digital skeleton is often one of the first processing steps taken by a computer vision system when attempting to extract features from an object in an image. A skeleton is presumed to represent the shape of the object in a relatively small number of pixels, all of which are (in some sense) *structural* and therefore necessary. In line images the skeleton conveys all the information found in the original, wherein lies the value of the skeleton: The position, orientation, and length of the line segments of the skeleton are representative of those of the lines of which the image is composed. This simplifies the task of characterizing the components of the line image.

Thinning, therefore, can be defined as the act of identifying those pixels belonging to an object that are essential for communicating the object's shape: these are the skeletal pixels, and form a set. That no generally agreed-upon definition of a digital skeleton exists has been pointed out by many people [Davis, 1981; Haralick, 1991] without altering the situation. Of the literally hundreds of papers on the subject of thinning in print, the vast majority are concerned with the implementation of a variation on an existing thinning method, where

the novel aspects are related to the performance of the algorithm. Many of the more recent thinning algorithms were designed with an eye on the clock: The speed of the algorithm is improved, while often leaving the basic principles alone. The quality of the skeleton or the means by which it is found is rarely the subject of analysis.

This chapter examines a number of approaches to thinning, and we will always come back to the original issue of the definition without finding a solution. However, three things can be stated in advance and should be kept in mind:

1. Not all objects can or should be thinned. Thinning is useful for objects consisting of lines, whether they are straight or curved, and is not useful for objects having a shape that encloses a significant area. For example, a circle can be thinned since it is represented by a curved line; a disk cannot be meaningfully thinned.
2. What works as a skeleton in one situation may not work in all situations. Thinning is usually one step in preparing an image for further processing. The nature of the subsequent steps often dictates the properties needed of the skeleton.
3. Thinning is the act of identifying the skeleton, and is not defined by the algorithm used. In particular, thinning is not always an iterative process of stripping away the outer layers of pixels.

6.2 The Medial Axis Transform

Possibly the first definition of a skeleton is that of [Blum, 1967] in defining the *medial axis function* (MAF). The MAF treats all boundary pixels as point sources of a wave front. Each of these pixels excites its neighbors with a delay time proportional to distance, so that they, too, become part of the wave front. The wave passes through each point only once, and when two waves meet they cancel each other, producing a *corner*. The *medial axis* is the locus of the corners, and forms the skeleton (Blum says *line of symmetry*) of the object. The MAF uses both time and space information, and can be inverted to give back the original picture. It is possible to implement this directly, but it is difficult: What is needed is to convert the continuous transform to a discrete one. This involves various approximations involving the distance function on a discrete grid. This allows the MAF to be applied to a raster image, for which the medial axis is not defined.

One way to find the medial axis is to use the boundary of the object. For any point P in the object, locate the closest point on the boundary. If there is more than one boundary point at the minimum distance, then P is on the medial

axis. The set of all such points is the medial axis of the object. Unfortunately, this must be done at a very high resolution, or Euclidean distances will not be equal when they should be, and skeletal pixels will be missed.

An approximation to the medial axis on a sampled grid is more easily obtained in two steps. First, compute the distance from each object pixel to the nearest boundary pixel. This involves computing the distance to all boundary pixels. Next, the Laplacian of the distance image is calculated, and pixels having large values are thought to belong to the medial axis.

The way that distance is measured has an impact on the result, as shown in Figure 6.1. The medial axis was found for a T-shaped object using Euclidean distance, *4-distance*, and *8-distance*. 4-distance between pixels A and B is defined to be the minimum number of horizontal and vertical moves needed to get from A to B. 8-distance is the minimum number of pixel moves, in any of the standard eight directions, needed to get from A to B. There are clear differences in the medial axis depending on which way distance is calculated, but any of them could be used as a skeleton.

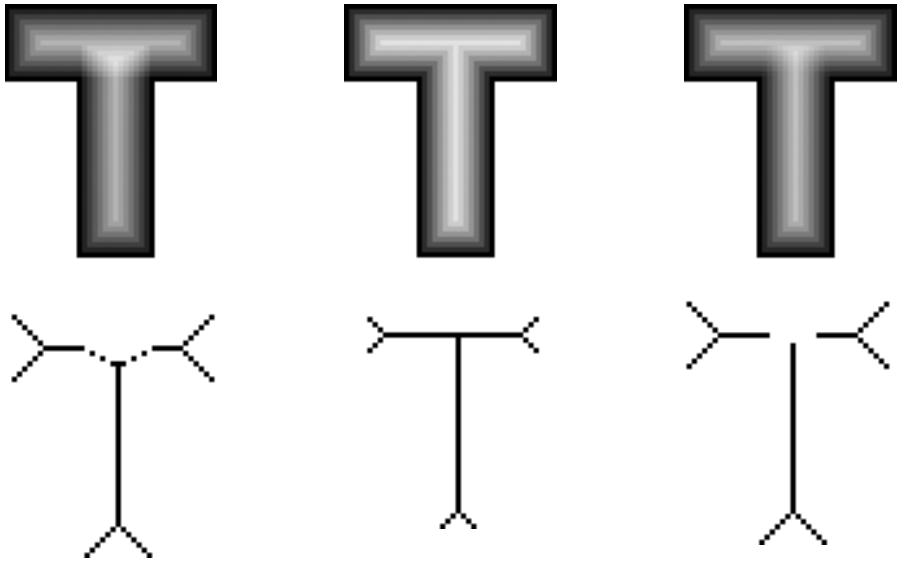
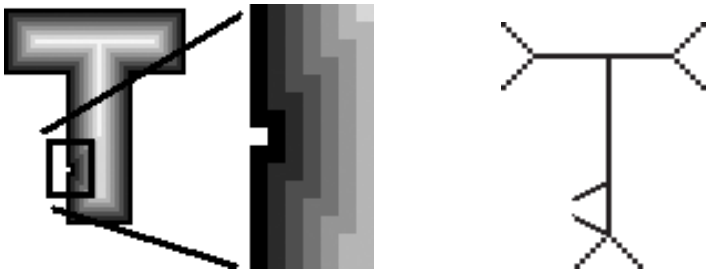


Figure 6.1: The effect of the distance function on the medial axis. (a) Medial axis (above) and skeleton (below) of the T-shaped object, using 4-distance. (b) Medial axis and skeleton computed using 8-distance. (c) Computed using Euclidean distance.

The skeleton of the *T* produced by the medial axis does not have the same shape as the *T*, nor does it need it. The main concern is whether the skeleton characterizes the basic shape of the object somehow. On the other hand, a simple example exposes a fundamental problem with the medial axis as a

skeleton. Most people would agree that the skeletons of two objects that are similar to each other should, in turn, be similar. Figure 6.2 shows an object that differs from Figure 6.1a in only a single pixel; the medial axes of these objects, on the other hand, differ substantially.



(a) (b)

Figure 6.2: A single pixel difference between two objects can create a large difference in their skeletons. (a) The T-shaped object, but with one less black pixel. (b) The skeleton of the new object, quite different from those in Figure 6.1.

Most vision researchers would agree that the way the MAF is applied to discrete, raster images often does not yield an ideal skeleton, and takes too long to compute. It does, however, form the basis of a great many thinning methods, and in that regard is a very important concept.

6.3 Iterative Morphological Methods

The majority of thinning algorithms are based on a repeated stripping away of layers of pixels until no more layers can be removed. A set of rules defines which pixels can be removed, and some sort of template-matching scheme frequently is used to implement these rules. Often, the rules are designed so that it is easy to tell when to stop: when no change occurs after two consecutive passes through the image.

The first such algorithm to be described [Stentiford, 1983] is typical of the genre. It uses 3×3 templates, where a match of the template in the image means to delete (set to white) the center pixel. The basic algorithm is:

1. Find a pixel location (i,j) where the pixels in the image I match those in template M1 (Figure 6.3a).
2. If the central pixel *is not an endpoint*, and has *connectivity number* $= 1$, then mark this pixel for later deletion.
3. Repeat steps 1 and 2 for all pixel locations matching the template M1.
4. Repeat steps 1–3 for the remaining templates in turn: M2, M3, and M4.

5. If any pixels have been marked for deletion, then delete them by setting them to white.
6. If any pixels were deleted in step 5, then repeat the entire process from step 1; otherwise, stop.

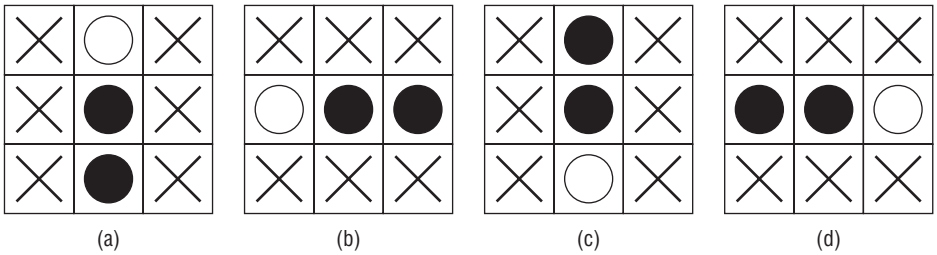


Figure 6.3: Templates for identifying pixels that can be deleted in the Stentiford thinning algorithm. (a) Template M1. (b) Template M2. (c) Template M3. (d) Template M4. The specified black and white pixels in the templates must correspond to pixels of an identical color in the image; the Xs indicate places where we don't care what color the image pixel is.

The image must be scanned for a template match in a particular order for each template. The purpose of template M1 is to find removable pixels along the top edge of an object, and we search for a match from left to right, then from top to bottom. M2 will match a pixel on the left side of an object; this template moves from the bottom to the top of the image, left to right. M3 will locate pixels along the bottom edge, and moves from right to left, bottom to top. Finally, to find pixels on the right side of an object, match template M4 in a top-to-bottom, right-to-left fashion. This specific order and direction for applying the templates ensures that the pixels will be removed in a symmetrical way, without any significant directional bias.

There are still two issues to be resolved, both from step 2. A pixel is an endpoint if it is connected to just one other pixel; that is, if a black pixel has only one black neighbor out of the eight possible neighbors. If endpoints were to be deleted, then any straight lines and open curves would be removed completely, rather like opening a zipper.

The concept of a connectivity number is somewhat more challenging. Because we are using only very small parts of an image, the role of that image segment in the overall picture is not clear. Sometimes a single pixel connects two much larger sections of an object, and it is intuitively obvious that such a pixel cannot be removed. To do so would create two objects where there was originally only one. A connectivity number is a measure of how many objects a particular pixel *might* connect.

One such connectivity measure, as shown in Figure 6.4, is [Yokoi, 1973]:

$$C_n = \sum_{k \in S} N_k - (N_k \cdot N_{k+1} \cdot N_{k+2}) \qquad \text{(EQ 6.1)}$$

Where N_k is the color value of one of the eight neighbors of the pixel involved, and $S = \{1, 3, 5, 7\}$. N_1 is the color value of pixel to the right of the central pixel, and they are numbered in counterclockwise order around the center. The value of N_k is one if the pixel is white (background) and zero if black (object). The center pixel is N_0, and $N_k = N_{k-8}$ if $k > 8$. Another way that connectivity can be computed is by visiting the neighbors in the order $N_1, N_2 \ldots N_8, N_1$. The number of color changes (black–white) counts the number of regions the central pixel connects.

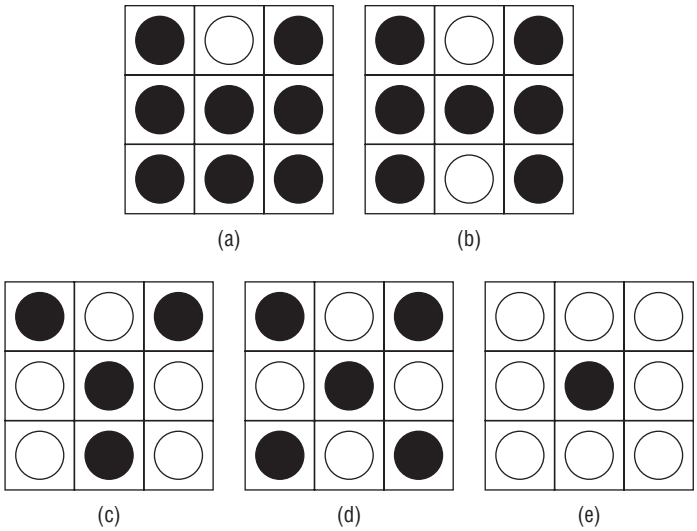


Figure 6.4: An illustration of the connectivity number. (a) The central pixel does not connect any regions and can be removed. Connectivity number = 1. (b) If the central pixel were to be deleted, then the left and right halves could become disconnected. Connectivity number = 2. (c) Connectivity = 3. (d) Connectivity = 4, the maximum. (e) Connectivity = 0.

Figure 6.5 shows one iteration (the first) of this thinning algorithm applied to the T-shaped object of Figure 6.1. One iteration includes one pass for each of the four templates. The black pixels are those marked for deletion, and it is clear from the figure exactly what each template accomplishes. Each complete iteration effectively erodes a layer of pixels from the outside of the object, but unlike standard morphological erosion, the deletion of a pixel is contingent upon meeting the endpoint and connectedness constraints.

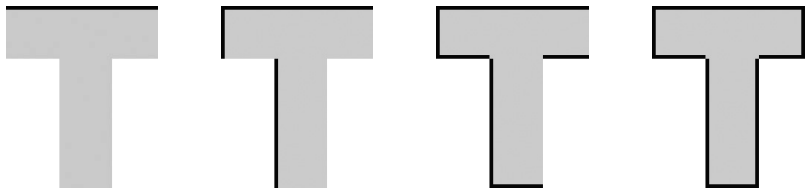


Figure 6.5: The four parts of each iteration of the Stentiford thinning method. (a) After applying template M1. (b) After M2. (c) After M3. (d) After M4. In each case, the black pixels represent those to be deleted in this iteration.

Complete thinning of this object requires 13 iterations (counting the final iteration, which does nothing except show that we are finished). Figure 6.6 shows the resulting image after each iteration. One iteration makes four passes through the image, which in this case is 60×60 pixels, or 3600 pixels. Thus, 187,000 pixels were examined in order to thin this simple image. It gets worse: Each template application looks at three pixels (the maximum is 561,600), and each time a template match occurs, another 18 pixels are looked at (the upper limit is 10,108,800 pixels, but will be a fraction of that in practice). Finally, there will be one extra pass of each iteration to delete the marked pixels (10,152,000). This is an expensive way to thin a small image, but is quite typical of template-based mark-and-delete algorithms.

A few classic problems with this thinning algorithm show up as *artifacts* in the skeleton. They are classic because they tend to appear in a great variety of algorithms of this type, and researchers in the area have learned to anticipate them. The first of these is called *necking*, in which a narrow point at the intersection of two lines is stretched into a small line segment (Figure 6.7a). Tails can be created where none exist because of excess thinning where two lines meet at an acute angle (Figure 6.7b). Finally, and perhaps most commonly, is the creation of extra line segments joining a real skeletal segment; this has been called a *spurious projection*, *hairs*, or *line fuzz* (Figure 6.7c).

Stentiford suggests a preprocessing stage to minimize these thinning artifacts. Since line fuzz is frequently caused by small irregularities in the object outline, a smoothing step is suggested before thinning to remove them. Basically, a pass is made over all pixels, deleting those having two or fewer black neighbors and having a connectivity number less than two.

For dealing with necking, he suggests a procedure called *acute angle emphasis*, in which pixels near the joint between two lines are set to white if they ''plug up'' an acute angle. This is done using the templates shown in Figure 6.8. A match to any template marks the central pixel for deletion, and causes another iteration of less severe acute angle emphasis using only the first three

templates of each type. If any pixels were deleted, one last pass using only the first templates of each type is performed.

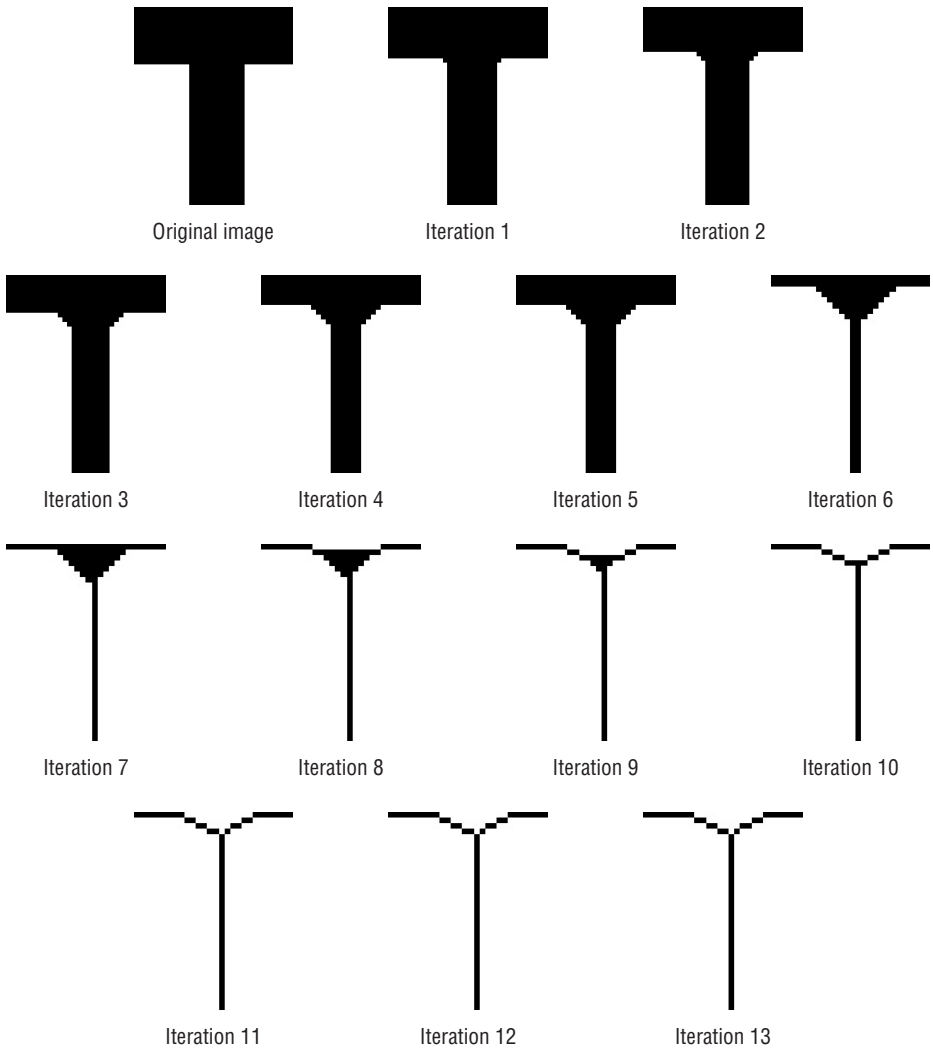


Figure 6.6: All iterations of the Stentiford thinning algorithm applied to the *T*. The last two iterations are the same, since one extra pass is needed to ensure that the skeleton is complete.

Smoothing is done first, followed by all passes of acute angle emphasis, followed finally by the thinning steps. Figure 6.9 shows the final skeletons of the characters from Figure 6.7 when the preprocessing steps are included.

As good as these skeletons appear to be, the method is still flawed. The use of three stages of acute angle emphasis will not be sufficient for very thick characters, and the templates do not match all situations that can cause

necking and tailing. Also, the smoothing step will not catch all irregularities that can cause line fuzz. Still, perfection should not be expected, and the method does pretty well, particularly as a preprocessing step for character recognition.

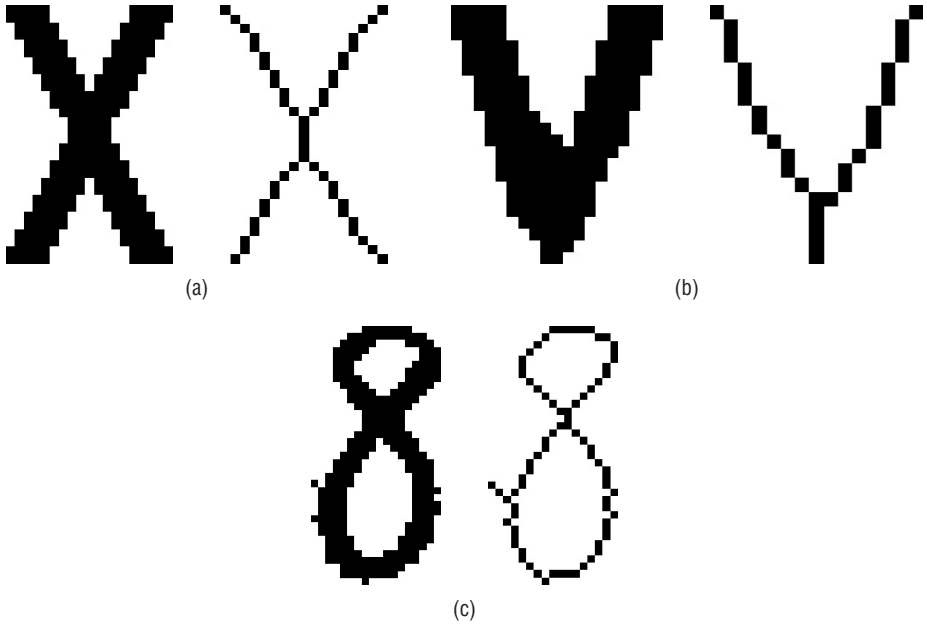


Figure 6.7: Classic thinning artifacts. (a) Necking. (b) Tailing. (c) Spurious projection (line fuzz).

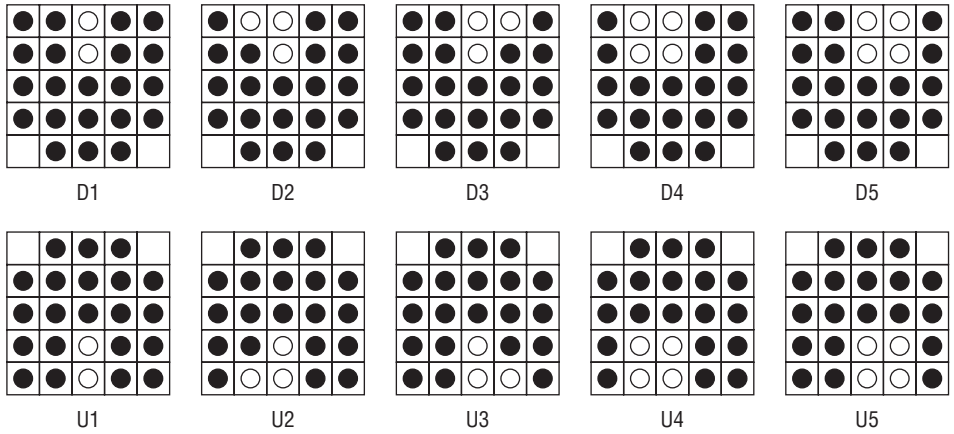


Figure 6.8: Templates used for the acute angle emphasis preprocessing step.

One thinning algorithm that seems to be in everybody's toolkit is the Zhang-Suen [Zhang, 1984] algorithm. It has been used as a basis of comparison

for thinning methods for many years, and is fast and simple to implement. It is a *parallel* method, meaning that the new value for any pixel can be computed using only the values known from the previous iteration. Therefore, if a computer having one CPU per pixel were available, it could determine the entire next iteration simultaneously. Since most of us don't have a computer of that size, let's consider only the version of the program that uses one CPU.

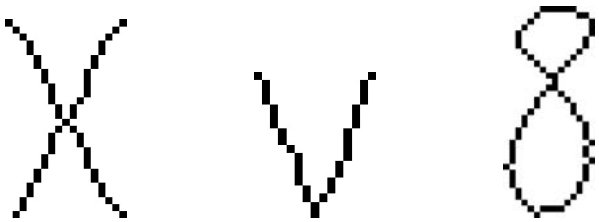


Figure 6.9: Final thinned characters, after both preprocessing steps and thinning.

The algorithm is broken up into two subiterations instead of, for example, the four subiterations of the Stentiford method. In one subiteration, a pixel $I(i,j)$ is deleted (or marked for deletion) if the following four conditions are all true:

1. Its connectivity number is one (1).
2. It has at least two black neighbors and not more than six.
3. At least one of $I(i, j+1)$, $I(i-1, j)$ and $I(i, j-1)$ are background (white).
4. At least one of $I(i-1, j)$, $I(i+1, j)$ and $I(i, j-1)$ are background.

At the end of this subiteration the marked pixels are deleted. The next subiteration is the same except for steps 3 and 4:

1. At least one of $I(i-1, j)$, $I(i, j+1)$ and $I(i+1, j)$ are background.
2. At least one of $I(i, j+1)$, $I(i+1, j)$ and $I(i, j-1)$ are background.

and again, any marked pixels are deleted. If at the end of either subiteration there are no pixels to be deleted, then the skeleton is complete, and the program stops.

Figure 6.10 shows the skeletons found by the Zhang-Suen algorithm applied to the four example images shown so far: the *T*, *X*, *V*, and *8*. The *T* skeleton is exceptionally good, and the *V* skeleton does not show any signs of tailing. The *X* skeleton does still show necking, and the *8* skeleton still has line fuzz. The preprocessing steps suggested by Stentiford may clear this up.

Before trying this, an improvement of the algorithm was suggested [Holt, 1987] that is faster and does not involve subiterations. First, the two

subiterations are written as logical expressions which use the 3×3 neighborhood about the pixel concerned. The first subiteration above can be written as:

$$v(C) \wedge (\sim \text{edge}(C) \vee (v(E) \wedge v(S) \wedge (v(N) \vee v(W)))) \quad \text{(EQ 6.2)}$$

which is the condition under which the center pixel C survives the first subiteration. The v function gives the value of the pixel (1 = true for an object pixel, 0 = false for background), and the *edge* function is true if C is on the edge of the object — this corresponds to having between two and six neighbors and connectivity number = 1. The letters E, S, N, and W correspond to pixels in a particular direction from the center pixel C; E means east (as in $I(i,j+1)$) S means south (as in $I(i+1,j)$) and so on.

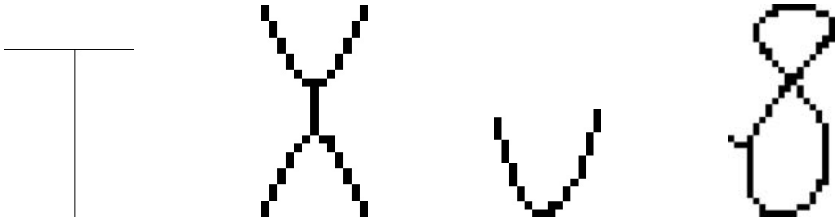


Figure 6.10: The skeletons produced by the standard Zhang-Suen thinning algorithm when applied to the test images of Figure 6.2 and 6.7.

The second subiteration would be written as:

$$v(C) \wedge (\sim \text{edge}(C) \vee (v(W) \wedge v(N) \wedge (v(S) \vee v(E)))) \quad \text{(EQ 6.3)}$$

Holt et al. combined the two expressions for survival (Eqs. 6.2 and 6.3) with a connectedness-preserving condition (needed for parallel execution) and came up with the following single expression for pixel survival:

$$\begin{aligned} v(C) \wedge (&\sim \text{edge}(C) \vee \\ &(edge(E) \wedge v(N) \wedge v(S)) \vee \\ &(edge(S) \wedge v(W) \wedge v(E)) \vee \\ &(edge(E) \wedge \text{edge}(SE) \wedge \text{edge}(S)) \end{aligned} \quad \text{(EQ 6.4)}$$

This expression is not a daunting as it appears; the v functions are simply pixel values, and the edge function is just about as complex as the connectivity function used in the Stentiford algorithm. The results from this are good, but not identical to the standard Zhang-Suen. However, there is still more to come.

Sometimes, when thinning is complete, there are still pixels that could be deleted. Principal among these are pixels that form a staircase; clearly half of the pixels in a staircase could be removed without affecting the shape or

connectedness of the overall object. Basically, the central pixel in one of the following windows can be deleted:

$$\begin{array}{ccc} 0 & 1 & x \\ 1 & 1 & x \\ x & x & 0 \end{array} \quad \begin{array}{ccc} x & 1 & 0 \\ x & 1 & 1 \\ 0 & x & x \end{array} \quad \begin{array}{ccc} 0 & x & x \\ x & 1 & 1 \\ x & 1 & 0 \end{array} \quad \begin{array}{ccc} x & x & 0 \\ 1 & 1 & x \\ 0 & 1 & x \end{array}$$

To avoid creating a new hole, we simply add a condition that one of the x values be 0. For windows having a northward bias (the first two above) the expression for survival of a pixel in the staircase-removal iteration is:

$$\begin{array}{l} v(C) \wedge \sim (v(N) \wedge \\ ((v(E) \wedge \sim v(NE) \wedge \sim v(SW) \ \wedge (\sim v(W) \vee \sim v(S)) \vee \\ (v(W) \wedge \sim v(NW) \wedge \sim v(SE) \ \wedge (\sim v(E) \vee \sim (S)))))) \end{array} \qquad \text{(EQ 6.5)}$$

The pass having a southward bias is the same, but with north and south exchanged. None of the example images shown so far possess any significant amount of staircasing, but the image introduced in Figure 6.11 does. The version thinned using staircase removal seems more smooth and symmetrical than the other skeletons. Figure 6.12 shows the result of applying this method to the four test images we have been using. The basic problems are still present; in fact, this method does not deal with tails as well as the standard Zhang-Suen method, and the T skeleton is not as good.

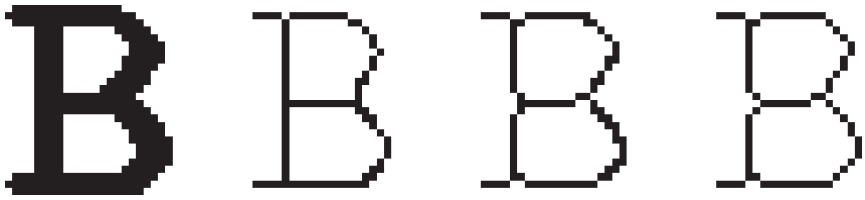


Figure 6.11: Variations on the Zhang-Suen thinning algorithm. (a) Original image (b) Thinned using the standard algorithm. (c) Thinned using Holt's variation. (d) Holt's variation plus staircase removal.

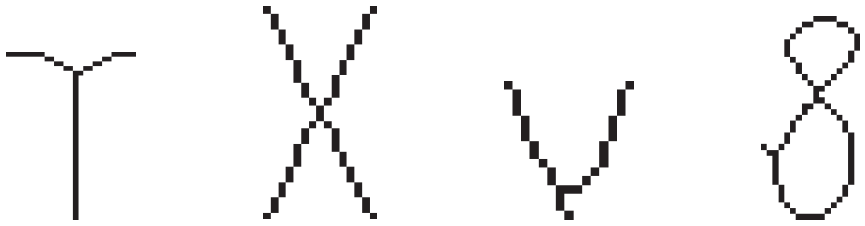


Figure 6.12: Results from Holt's algorithm with staircase removal applied to the standard test images.

If simple speed is what is of importance, then the Holt variation of Zhang-Suen is the better of the methods shown so far. On the other hand, if the quality of the skeleton is of prime importance, it is probable that a merging of the three methods is in order: Stentiford's preprocessing scheme feeding images into Zhang-Suen's basic algorithm, with Holt's staircase removal as a post-processor. The code for this sequence of operations appears in Section 6.8, since it includes all the techniques of importance that have been discussed to this point. It is available on the accompanying website as the program `zhangsuenbest.c`, and does appear to generate the best skeletons of all the methods shown so far; of course, this is a subjective measure. Figure 6.13 shows the best skeletons.

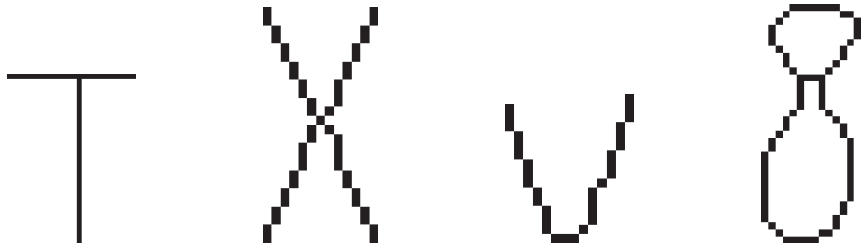


Figure 6.13: Skeletons obtained from using the Stentiford preprocessing steps combined with the Zhang-Suen thinning algorithm and Holt's staircase-elimination procedure.

6.4 The Use of Contours

The iterative mark-and-delete methods discussed so far have in common that they always delete pixels from the outer layer. These are on the boundary of the object, and form a contour having a distance of zero pixels from the background. A contour-based method locates the entire outer contour, or even all contours, and then deletes the pixels on the contour except those needed for connectivity. These methods tend not to use 3×3 templates for locating pixels to be removed.

The essential scheme used by a contour-based thinning algorithm is:

1. Locate the contour pixels.
2. Identify pixels on the contour that cannot be removed.
3. Remove all contour pixels but those in step 2.
4. If any pixels were removed in step 3, then repeat from step 1.

The external contour is usually traced in a manner identical to that used when finding the *chain code*. Starting at any black pixel having a horizontal or vertical neighbor in the background (a boundary pixel), the pixels on the contour are visited or traced in a counterclockwise fashion until the starting

pixel is seen again. The pixels are saved in a list when visited, giving a fast way to revisit them later. Then the contour pixels are marked somehow and the process is repeated, in order to find internal contours such as would occur around a hole in the object, until no starting pixels remain.

After the contour has been identified, the contour pixels that must not be removed can be located. One way to do this uses the concept of a *multiple pixel* [Pavlidis, 1982]. There are three types of multiple pixel, none of which can be safely removed from a contour.

- The first type (as shown in Figure 6.14a) appears more than once in the list of contour pixels. The reason for this is that there is no way to get from one part of the object to the other without passing through the multiple pixel, so deleting this pixel would separate the two parts. It is this type of pixel for which the phrase *multiple pixel* was named.
- The second type of multiple pixel has no neighbors in the interior of the object; that is, all its object neighbors belong to the contour (Figure 6.14b). This could only occur if the pixel involved *stuck out* from the boundary; perhaps it is the endpoint of a line, for example. Line endpoints clearly cannot be deleted either.
- The third type of multiple pixel has a neighbor that is on the contour but that is not its immediate successor or predecessor. This can occur when the contour turns back on itself, or when an internal and an external contour meet and parallel each other, as in Figure 6.14c. To delete such pixels would create the possibility that two-pixel-wide lines would simply be removed, which is unacceptable.

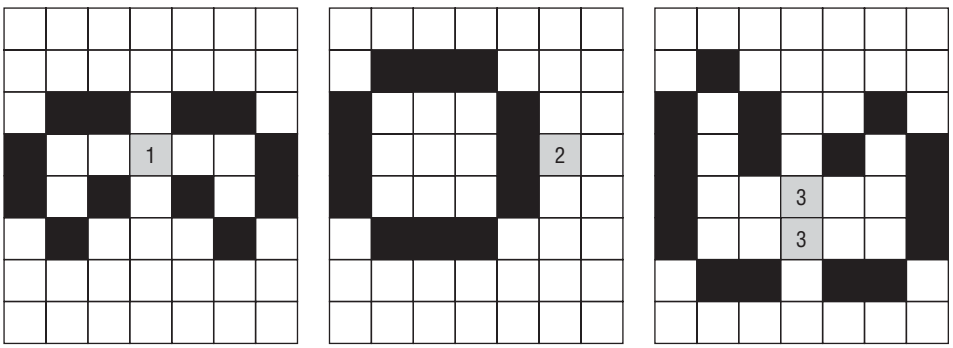


Figure 6.14: Multiple pixels. (a) The pixel marked 1 is a multiple pixel because it is visited twice in one complete traversal of the contour. (b) The pixel marked 2 is a multiple pixel because it has no neighbors that are not contour pixels (no internal neighbors). (c) The pixels marked 3 are multiple pixels because they are neighbors of each other, but do not occur immediately before or after each other in the list of contour pixels. They belong to different parts of the same contour in this case.

The three types of multiple pixels are easy to identify using a pass through the contour, marking the pixels and checking the three conditions. They are marked as nonremovable, and then all other contour pixels are set to the background level. Contours are marked, checked, and deleted until no further change takes place.

There are a few problems with the algorithm as presented so far, the most serious of which is that the skeletons obtained are two pixels thick in many places; this is due to multiple pixels of type 3 being neighbors, but not removable. The fix is to add another pass through the thinned image, removing all pixels on a right-angle corner [Zhang, 1984], as shown in Figure 6.15a. Since it was useful before, a pass of Holt's staircase removal was added as well.

Another problem is caused by the tracing of the contours. Some tracers (including Pavlidis's) will trace a tiny contour in locations where an external and internal contour pass close to each other (see Figure 6.15b). Rather than rewrite the tracer, the program was modified to ignore contours having four or fewer pixels. This may have ramifications if the image is supposed to contain a number of one-pixel holes.

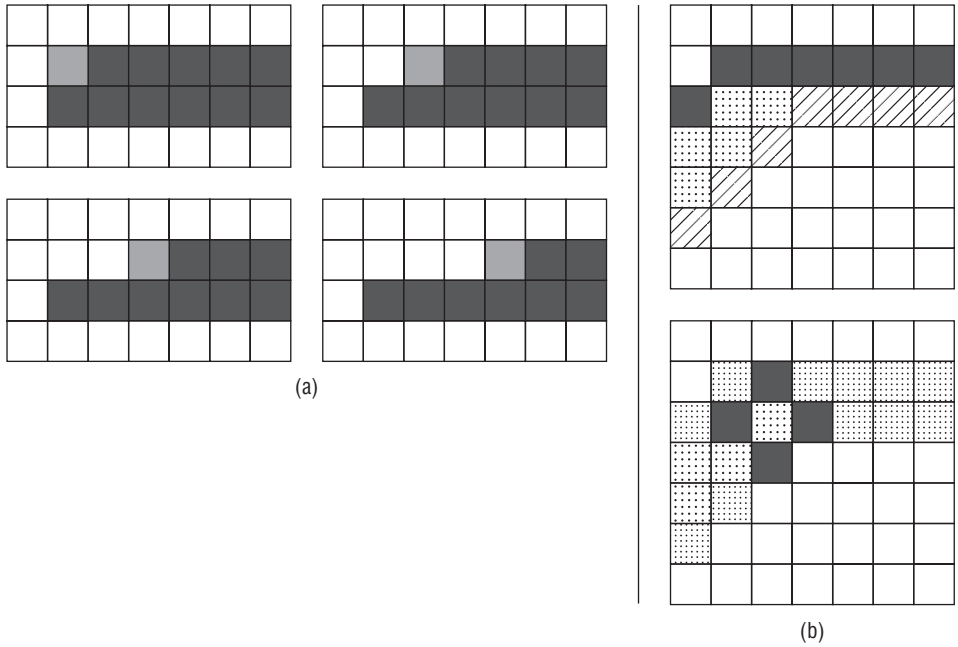


Figure 6.15: Problems with contour thinning. (a) Two-pixel-wide lines should be thinned further—in this case, by deleting those pixels on the corner of a right angle. (b) Some situations can result in a spurious contour; one cure is to forbid very small contours.

The resulting skeletons suffer from artifacts, especially tailing. The preprocessing methods do not help a great deal, but the smoothing procedure from

Stentiford does limit the fuzz, and so was added. The skeletons generated by this system for the four test images can be seen in Figure 6.16. If these do not appear to be better than those of Figure 6.13, remember that the advantage of the contour methods is supposed to be speed. For example, a more recent contour-based algorithm [Kwok, 1989] claims execution speeds that are 10–20 times faster than Zhang-Suen, but no quality comparisons are available.

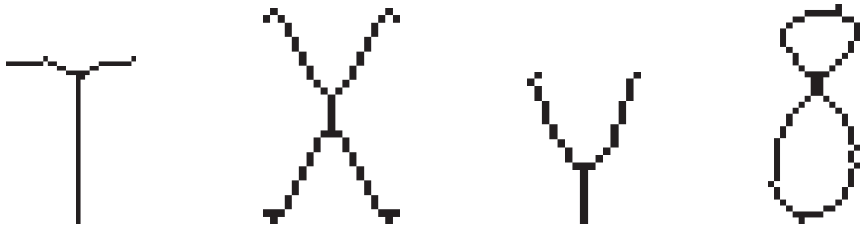


Figure 6.16: The result of using contour-based thinning. Each of the four test images has been thinned by repeatedly stripping away contours, while leaving the multiple pixels behind. Double lines were removed after the fact (Figure 6.15), and staircases were removed. The smoothing pass described by Stentiford was done before thinning was attempted.

6.4.1 Choi/Lam/Siu Algorithm

Some of the early work on thinning considered that skeletal pixels could be thought of as the centers of consecutive circles that were tangent to the outline (contour) of the object being thinned. Indeed, this is true; circles can be drawn inside of thick lines that touch both side of the line and contain only object pixels. If such a circle is moved along the line, adjusting the radius to fit, it can remain in contact with both sides, and the center of the circles forms a line or curve that is skeletal. The problem is that this really only works well for continuous circles. Digital objects have jagged boundaries, thick edges, and quantized features, so it is impossible to accomplish what has just been described.

[Choi, 2003] suggests that it is possible to characterize skeletal pixels by considering that they are the locus of a sequence of circle centers and by considering connectivity and geometry. With this in mind, note that skeletal pixels tend to be associated with two or more minimal distance boundary points, as exemplified by the points Q_1 and Q_2 in Figure 6.17a. These points can be thought to divide the object boundary into two parts, part A from Q clockwise to Q_2, and part B, clockwise from Q_2 to Q_1. From these definitions and from basic geometry can be derived an astonishing property: If a point exists along part A outside of the circle centered at P having a distance greater than a specified value d, and another lies on part B, also greater than distance d from P, then P is a skeletal pixel.

The thinning algorithm works by testing all object pixels to determine if they are skeletal, rather than by stripping away successive contours. For a

candidate pixel, P, determine the nearest contour pixel, Q. This is done in an especially clever way using the *signed sequential Euclidean distance* (SSED) transform — a distance map in which the X and Y directions to the nearest boundary are kept, not just the distance. Thus, each element in the distance map is a vector, v, and $P + v$ is the nearest boundary pixel for any object pixel, P.

The SSED transform is tricky to implement. The website provides an example, `choi.c`, but the original article [Ye, 1988] describes a faster albeit more obscure version. Figure 6.17b shows an example of this distance transform applied to a simple image.

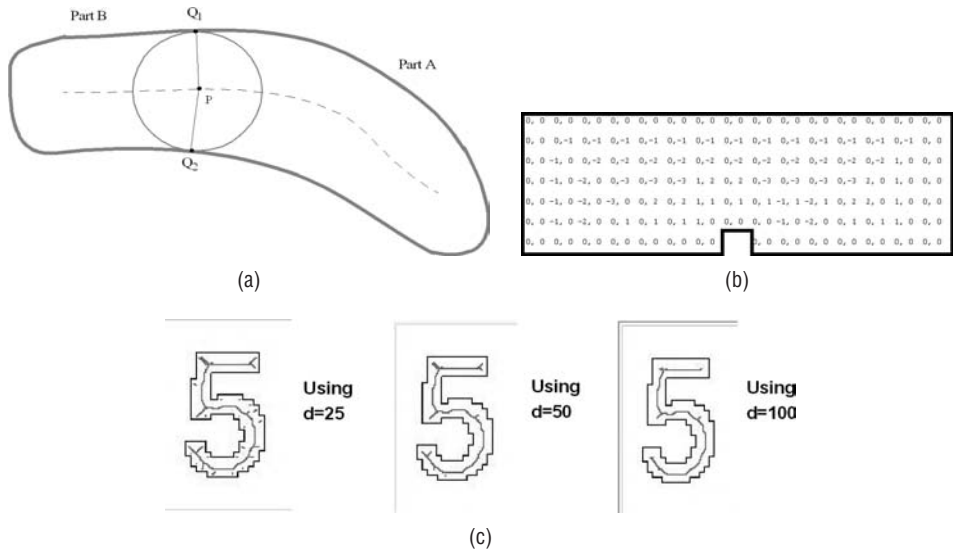


Figure 6.17: Using the SSED transform to thin a character. (a) Basic definitions surrounding the use of a disk and simple geometry in thinning. (b) The SSED image for a simple rectangle with one pixel cut out of the boundary. The numbers represent the x and y distance to the nearest boundary pixel. (c) The result of applying this method to a digit "5" for three values of the distance parameter d.

A thinning method based on these ideas is: Compute the SSED $= \boldsymbol{I}_d$ for the target image I.

1. For each object pixel $\boldsymbol{P} = \boldsymbol{(Px,Py)}$ in the image which is not on the boundary:
   a. Find the point $\boldsymbol{Q}$ on the boundary which is closest to $\boldsymbol{P}$. Do this by adding the coordinates of $\boldsymbol{P}$ to those of $\boldsymbol{I}_d[Px, Py]$. That is:

   $$\boldsymbol{Q}_x = P_x + I_{dx}[P_x, P_y]$$
   $$Q_y = P_y + I_{dy}[P_x, P_y]$$
   where $\boldsymbol{I}_d[x, y]$ is a vector $= (\boldsymbol{I}_{dx}[x, y], I_{dy}[x, y])$

b. Find the nearest boundary pixel to each of the eight neighbors of $\boldsymbol{P}$. Call these $\boldsymbol{Q}_i$.

2. For $i = 1 \ldots 8$:
   a. Check the distance value $|\boldsymbol{Qi} - \boldsymbol{Q}|^2$ for to ensure that it is greater than $\boldsymbol{d}$. If not, reject $\boldsymbol{Q}_i$. Resume from B. Here $|\boldsymbol{x}|$ represents the Euclidean norm, and so $|\boldsymbol{x}|^2$ is the norm squared. This means we avoid computing the square root.
   b. Find the vector difference $\boldsymbol{dQ} = \boldsymbol{Qi} - \boldsymbol{Q}$, and let $\mathbf{z}$ be the value of maximum coordinate in $\boldsymbol{dQ}$; that is the max of $\boldsymbol{dQ}_x$ and $\boldsymbol{dQ}_y$.
   c. If $|\boldsymbol{Q}_i|^2 - |Q|^2 \leq z$, mark $\boldsymbol{Q}_\mathrm{i}$ as skeletal.
3. Repeat from step 1.

This method was applied to a digit ''5'' and the results are shown in Figure 6.17c. The value selected for d is important to the method, and this is unfortunate in general, as it may be hard to determine what d should be. Small values of d result in multiple branches and stems. Larger values reduce this effect, but very large values result in a disconnected skeleton. It may be possible to determine a good value of d from the average width of the object.

The method produces generally disappointing skeletons but has some interesting features. The use of the SSED transform in this thinning algorithm suggests other places where this type of distance map might be useful. The use of the parameter d can be seen as a curse, in that an estimate is needed in advance, but also a blessing, in that it could be used to ''tune'' the skeletons as desired.

6.5 Treating the Object as a Polygon

Instead of being treated as a raster object, the boundary of a region can be made into a polygon. The contour-based methods begin this way, since a chain code is really a representation of a polygon, but then fall into old habits and discard successive contours (polygons) until the skeleton remains. An interesting approach [Martinez-Perez, 1987] uses geometric properties of the polygon that represents the object boundary to locate the skeleton.

The first step is to obtain the boundary of the object represented as a polygon. This could be done with a chain code, creating many small polygon edges: one edge per pair of pixels. It would be better to convert the boundary into vector form, where each edge in the polygon was stored as its starting and ending pixel coordinates. The resulting polygon should be stored in counterclockwise order so that moving from one vertex to the other is a simple matter. It should be pointed out that vectorizing the boundary is not an easy thing to do.

Figure 6.18 shows a contrived test image as a set of vectors in polygon order. Now the nodes are traversed, and one of two things is done. If the angle made

by the node (and its previous and next points) is less than 180 degrees, then the angular bisector is constructed from the node to the point where the bisector meets the opposite face on the polygon; in Figure 6.18, this was done in the case of node 2. Otherwise, the line normal to both the incoming and outgoing edges is drawn to the point where it intersects the polygon; this has been done for nodes numbered 4 and 9, for example.

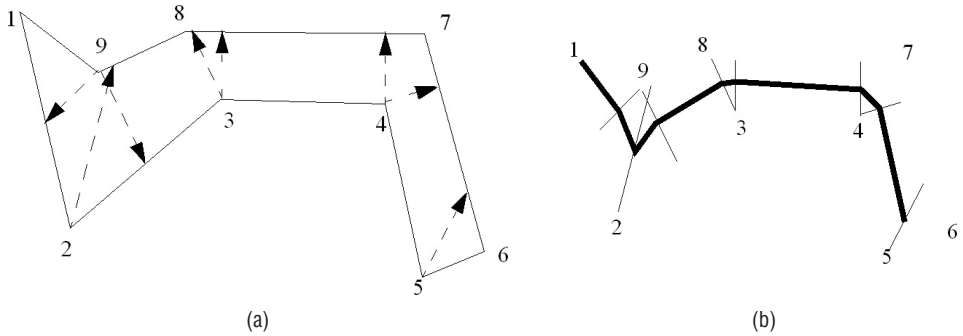


Figure 6.18: Treating the object as a polygon. (a) Some of the projecting lines computed for a hypothetical object. (b) The centers of the lines are points on the skeleton, which has been traced here as the thick black line.

Now, starting at vertex 1, the path through the midpoints of the line segments constructed in the previous step forms the skeleton. The tracing process is also not easy to implement, but on the face of it the skeleton should be a good one. There are some idiosyncracies (such as the possible formation of a loop in a thick corner, and the fact that the first skeletal segment starts in a corner, rather than at the middle of an edge).

This algorithm is intended for thinning objects that are not very thick to begin with. Characters would be included in the set of such objects, as would graphs and many maps. It is included here as an alternative strategy that shows potential, and one that has not been explored to the extent that it should.

6.5.1 Triangulation Methods

The previous method broke a polygon, representing an arbitrary shape to be thinned, into other, simpler, polygons that were easier to deal with. In computer graphics, *triangulation* is the act of taking an arbitrary polygon and converting it into triangles, which are faster to draw and shade, and generally easier to manipulate in graphics applications. The triangles do not overlap and fully tile the original polygon. Why is this interesting? Because these triangles can help find a skeleton of the original polygon. As all raster objects are essentially polygons, the adjacent pixels in the boundary can be connected, creating line segments; therefore, a polygon triangulation can be a key step in thinning because, as before, the midpoints of the triangles comprise a skeleton.

Many triangulation-based schemes exist (e.g., [Morrison, 2006; Melhi, 2001; Zou, 2001; Ogniewicz, 1995]). The details vary, but they are all based on the same basic scheme:

1. Identify points on the object outline to be used in triangulation.
2. Construct a triangulation of the outline.
3. Collect the midpoints of the triangle segments and connect them with line segments, creating a skeleton.

Of these three steps, the first step most determines the success of the procedure. Using every boundary pixel can work, but the triangulation step is unstable and the triangles are very small. Zou uses line segment endpoints on one side of the contour and a contour point on the other side, resulting in a narrow triangle. Morrison seems to construct contours from the boundary and start at the beginning of the contour. It is also possible to collect co-linear boundary pixels into sets, each corresponding to a straight line segment, and then to use the segment endpoints as a basis for triangulation. Once the starting point is found, the triangulation algorithms are well defined, although not all triangulation algorithms can deal with holes in the object. A good place to start in an implementation is the many websites about discrete geometry. The triangulation program at `www.cs.unc.edu/~dm/CODE/GEM/chapter.html`, described in *Graphics Gems*, is accessible

The starting point is important, too. Starting at the end of a linear stroke can present difficulties, and finding a point nearer the center is more work. Continuous contours can be extracted from the digital boundaries, and this makes the choice easier. Figure 6.19 shows a generic outline of a triangulation-based procedure.

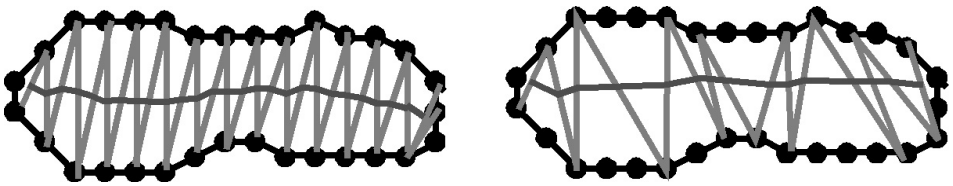


Figure 6.19: (a) A discrete outline of a naive triangulation. The skeleton is the line joining the centers of the triangle sides. (b) Using boundary pixel sets that correspond to straight line segments results in bigger triangles and fewer of them, but a similar skeleton.

6.6 Force-Based Thinning

Up to this point, the elements implicit to definitions of skeleton include:

- Skeletal pixels are in some sense as far from the object boundaries as possible.

- A skeletal pixel is connected to at least one other, unless the skeleton consists of exactly one pixel.
- A line crossing the object boundary as a perpendicular will cross the skeleton exactly once before crossing another boundary, unless it is (a) too close to a point where lines meet, or (b) too close to the end of a line.

As an example, a simple object and its human-computed skeleton are shown in Figure 6.20a, where grey represents a boundary pixel and a black pixel is a skeletal pixel. The skeleton above satisfies all the discussed properties, and while a six-year-old human could draw it, there are very few (if any) thinning algorithms that could. In most cases, humans perform thinning by computing a medial axis *in a preferred direction*. The center pixel found by slicing the object perpendicular to the stroke is chosen as skeletal wherever possible. This produces Figure 6.20b, which is purely computational.

There is also a perceptual aspect, which involves closing the gaps in the skeleton and extending the lines to the ends. This aspect can perhaps only be approximated on a computer. The direction in which to slice the object is that direction which is perpendicular to the stroke, and this may not be perpendicular to the boundary at all points. Nonlocal information is needed to perform this operation properly. In computer vision applications the skeleton of an object is extracted, and used to locate strokes. What is being proposed here is to reverse this process: Strokes are located and used to generate the skeletons.

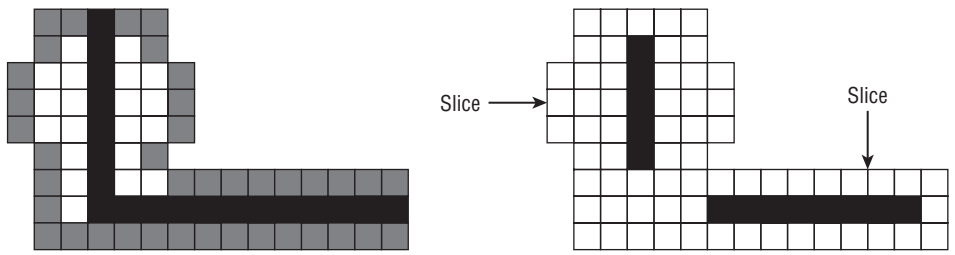


Figure 6.20: (a) A simple figure to be thinned. The human-generated skeleton is composed of the black pixels. (b) Slicing the figure in a direction normal to the boundary gives the bulk of the skeleton.

6.6.1 Definitions

A *digital band* can be defined as a set of connected pixels with the following properties:

- All pixels lie within perpendicular distance d of a discrete curve C, which does not have any loops (i.e., is simple). The minimum distance between C and any boundary pixel is $d/2$.

- The value of d is much smaller than the length of the curve C.
- The direction associated with each boundary pixel is approximately the same as that of the nearest point on C.

This definition would include most digital lines and curves, either thick or thin, as digital bands. A *digital band segment* is a subset of a digital band obtained by slicing the band at two places in a direction perpendicular to C at those places. This relaxes property two above so that the length of the curve C over the segment must be simply greater than $2d$.

A *stub* is a digital band segment where there are constraints placed on the changes in direction undergone by C. In particular, over the segment: (1) the direction may be constant (linear stub), or (2) the direction may represent either a convex or concave curve (but not both), having an identifiable (if approximate) center and radius of curvature. Finally, the *skeleton of a stub* is the set of pixels obtained by using the center pixel of each slice across the stroke in a direction perpendicular to C. For example, in the case of a linear stroke, these pixels should comprise the principal axis.

Now the approach to skeletonization can be clarified. Given a line image to be thinned, it can be broken down into a set of stubs that have been concatenated so that their boundaries form a continuous digital curve. These each have a clearly defined skeleton, and the first draft of the overall skeleton (the skeletal sketch) is simply the collected skeletons of all the stubs.

The skeleton may be complete at this point, although it is unlikely. The problem is that it is not possible to accurately determine the stubs comprising the object — some stubs are too short for this given that the image is discrete. It is often possible to fit hundreds of different stub combinations to a given object.

6.6.2 Use of a Force Field

The goal here is to find a method for locating skeletal pixels in a digital band that will also be useful as an approximation for objects consisting of concatenated band segments. Our idea is to have all the background pixels that are adjacent to the boundary act as if they exerted a $1/r^2$ force on the object pixels. The skeletal pixels will lie in areas having the ridges of this force field, and these areas can be located by finding where the directions of the force vectors change significantly.

The algorithm first locates the background pixels having at least one object pixel as a neighbor and marks them. These will be assumed to exert a repulsive ''force'' on all object pixels: The nearer the object pixel is to the boundary, the greater is the force acting on it. This force field is mapped by subdividing the region into small squares and determining the force acting on the vertices of the squares. The skeleton lies within those squares where the forces acting

on the corners act in opposite directions. Those squares containing skeletal areas are further subdivided, and the location of the skeletal area is recursively refined as far as necessary or possible.

The change in the direction of the force is found by computing the dot product of each pair of force vectors on corners of the square regions:

$$\begin{aligned} d_1 &= \hat{f}_1 \cdot \hat{f}_2 \\ d_2 &= \hat{f}_2 \cdot \hat{f}_3 \\ d_3 &= \hat{f}_1 \cdot \hat{f}_4 \end{aligned} \qquad \text{(EQ 6.6)}$$

If any one of d_1, d_2, or d_3 is negative, then the region involved contains some skeletal area.

To compute the force vector at each pixel location is time-consuming. For each object pixel, a straight line is drawn to all marked pixels on the object outline. Lines passing through the background are discarded, as illustrated in Figure 6.21, and for each of the remaining lines a vector with length $1/r^2$ and direction from the outline pixel to the object pixel is added to the force vector at that pixel. A graphical illustration of the force calculation is given in Figure 6.22.

This is done for all object pixels; then recursive subdivision can be used to refine the positions of the skeletal areas. From any endpoints of the skeleton found in the previous stage, we consider growing this skeletal line until it hits another skeleton or an edge. If it hits itself, the loop grown thereby is deleted.

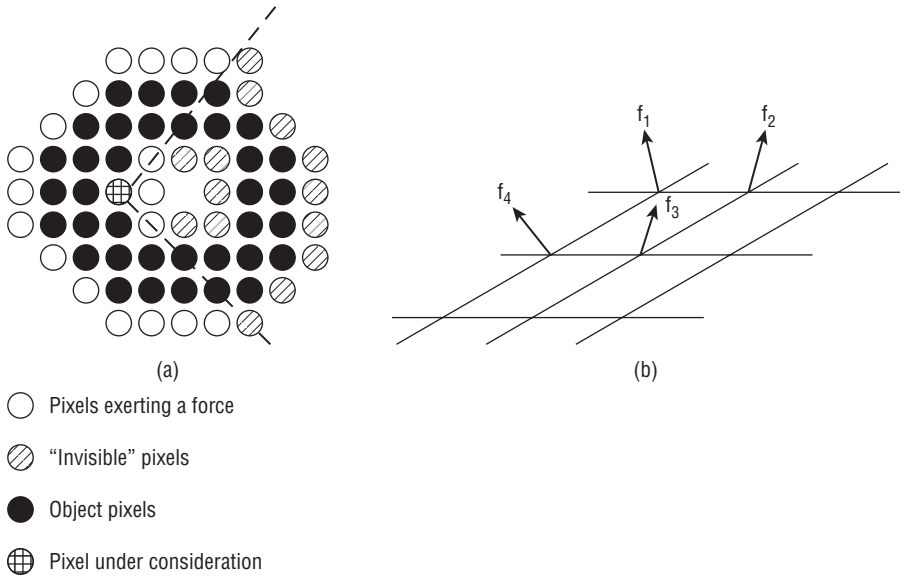


Figure 6.21: (a) When computing the force at a pixel, only the "visible" pixels are considered. The object insulates the others from having an influence. (b) The calculation of the dot product determines whether the force becomes zero somewhere in the pixel being tested.

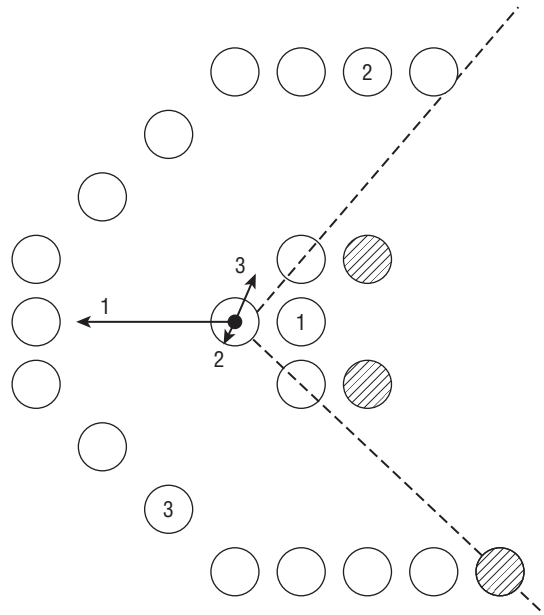


Figure 6.22: The force at a given pixel is the vector sum of the forces to all visible pixels. Only boundary pixels exert a force, and only object pixels have the force computed.

The details of the growing process are relatively simple. First, a queue to hold the points to be grown is defined. All the endpoints of the current stubs are placed into the queue as potential starting points for the growth process. Then points are removed from the queue one at a time and tested to see if growth is possible; if so, it is added to the skeleton and the new skeletal point is added to the queue if it, too, is a potential starting point.

To grow from a point P, the point must satisfy two conditions. P must have exactly one or two 8-connected neighbors that are skeletal pixels, and if it has two such neighbors, then these must be adjacent to each other. The preferred direction of growth is through these neighbors towards P and beyond to the next pixel. There will be three candidate pixels, and the one of these having the smallest force magnitude is *grown into*: It is added to the skeleton and placed on the queue for further growth steps. The growing process will stop when the growth front hits an edge or other part of the skeleton.

At a subpixel level, the growth process first attempts to find new skeletal pixels at double the previous resolution. Using the stub endpoints the regions to be refined are identified, and forces are computed for each pixel at the new resolution; the resolution doubles each time. Then the dot products are computed as before, looking for zero crossings. When located, a zero crossing becomes a skeletal pixel at the current resolution and also marks all containing pixels at lower resolutions as skeletal. The refinement can be continued at

higher resolutions until no change is seen; then the growth process continues at the original resolution in the original way (minimal force path).

This certainly approximates the set of skeletal pixels S for a digital band. For example, assume an infinitely long, straight band along the x axis, having width $2w$. Then the boundaries of the band are the lines $y = w$ and $y = -w$. Then the force acting on the point (x,y) would be:

$$F(x,y) = \int_{-\infty}^{\infty} \frac{L_1}{|L_1|^3} dl_x + \int_{-\infty}^{\infty} \frac{L_2}{|L_2|^3} dl_x \qquad \text{(EQ 6.7)}$$

where $L_1 = (x - l, y - w), L_2 = (x - l, w + y)$, and l is the length along the boundary. This becomes:

$$F(x,y) = \left(0, \frac{4y}{(w+y)(y-w)}\right) \qquad \text{(EQ 6.8)}$$

Now, any of the dot products referred to previously can be written as:

$$d_i = \left(\frac{16y(y+dy)}{(w+y)(y-w)(w+y+dy)(y+dy-w)}\right) \qquad \text{(EQ 6.9)}$$

All that is needed is to know in what circumstances this expression is negative. Since $-w + dy < y < w - dy$ it is known that $y - w$ and $y + dy - w$ are negative and that $w + y$ and $w + y + dy$ are positive, the sign of the dot product is the sign of $y(y + dy)$. Solving this quadratic reveals that it is negative only between 0 and $-dy$. Thus,

$$C(x,y,dx,dy) = \begin{cases} 1 & if - dy < y < 0 \\ 0 & otherwise \end{cases} \qquad \text{(EQ 6.10)}$$

As dy approaches 0 this becomes:

$$C(x,y) = \begin{cases} 1 & y = 0 \\ 0 & otherwise \end{cases} \qquad \text{(EQ 6.11)}$$

which means that the x axis is the skeleton, as was suspected. This demonstration holds for infinitely long straight lines in any orientation and having any width.

The application of this method to real figures is based on three assumptions:

- What is true for infinitely long lines is approximately true for shorter (and curved) ones;
- A figure can be considered to be a collection of concatenated digital band segments.
- Intersections can be represented by multiple bands, one for each crossing line.

From the results so far, these assumptions appear to be at least approximately true.

Further work on this idea has developed over the past decade. Most recently, researchers have developed a high-speed algorithm for approximating the force fields and have extended the force-based thinning idea into three dimensions [Brunner, 2008].

6.6.3 Subpixel Skeletons

The force-based thinning method has been implemented and tested on a number of images, both artificial and scanned. The results in all cases are either promising or excellent. The subpixel accurate skeletons provide substantially more information about the geometry of the object. There will often be areas in the object where the forces are not actually zero, but are small or known to be changing. By splitting each pixel into four pixels, a more accurate force can be calculated for each such pixel, and the region where the force is zero can be estimated. If this fails, each subpixel can be further split into four, and so on (Figure 6.23).

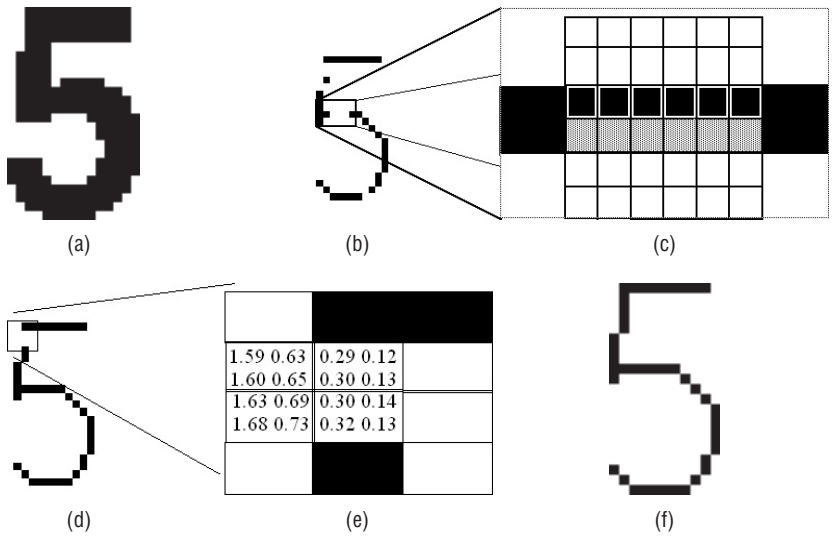


Figure 6.23: Subpixel skeletons. (a) The original image. (b) The level-1 skeleton, showing zero crossings. (c) A subpixel section of a gap in the level-0 skeleton. (d) The level-1 skeleton. (e) Subpixel force magnitudes in a gap in the level-1 skeleton. (f) The final skeleton with all gaps filled in.

This is expensive, since for each subpixel the visible pixels must be determined, and then the force accumulated. One way to speed this up is to compute the forces based on the lines formed by the boundary pixel instead of using

each boundary pixel individually. Then each line would exert a single force on each pixel, rather than many forces. In addition, the visibility calculation can be simplified by using a simple distance threshold. Line segments further than d units away would not contribute a force.

Figure 6.24 shows the forces computed for a hand-printed 8, and gives the skeleton as determined by the force-based method. For comparison purposes the skeleton found by the Zhang-Suen algorithm is shown also.

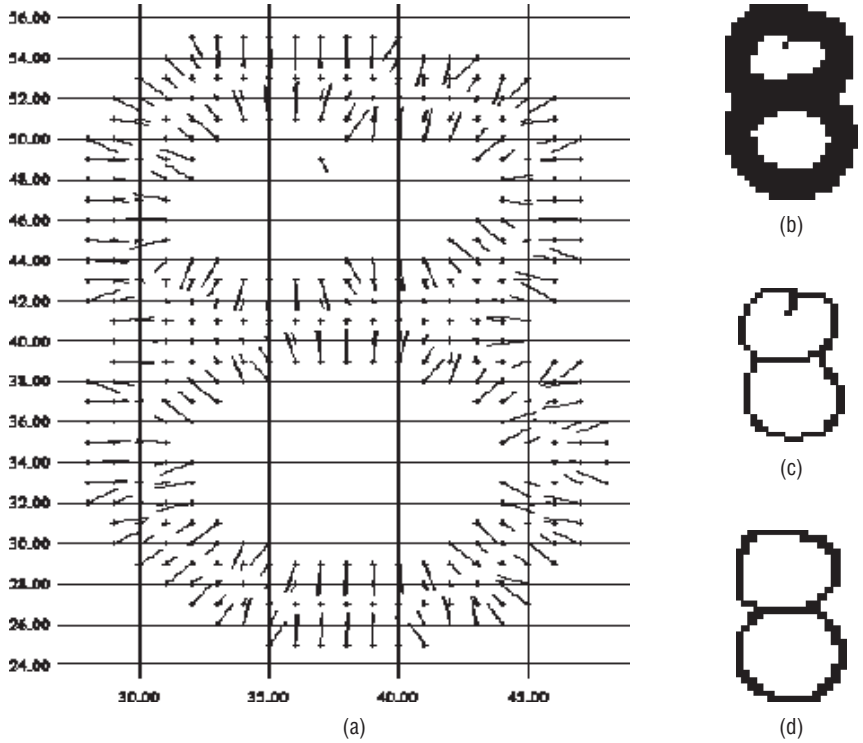


Figure 6.24: Force-based thinning applied to a handprinted 8. (a) The forces at each pixel, where force is proportional to the length of the line drawn from the pixels, and direction is normal to them. (b) The original figure. (c) Thinned using Zhang-Suen. (d) Force-based skeleton.

6.7 Source Code for Zhang-Suen/Stentiford/Holt Combined Algorithm

```
/* Zhang & Suen's thinning algorithm */
#define MAX
#include "lib.h"
#include <math.h>

#define TRUE 1
```

```
#define FALSE 0
#define NORTH 1
#define SOUTH 3
#define REMOVE_STAIR 1
void thnz (IMAGE im);
int nays8 (IMAGE im, int r, int c);
int Connectivity (IMAGE im, int r, int c);
void Delete (IMAGE im, IMAGE tmp);
void check (int v1, int v2, int v3);
int edge (IMAGE im, int r, int c);
void stair (IMAGE im, IMAGE tmp, int dir);
int Yokoi (IMAGE im, int r, int c);
void pre_smooth (IMAGE im);
void match_du (IMAGE im, int r, int c, int k);
void aae (IMAGE image);
int snays (IMAGE im, int r, int c);

int t00, t01, t11, t01s;

void main (int argc, char *argv[])
{
   IMAGE data, im;
   int i, j;
   if (argc < 3)
   {
     printf (`Usage: thnbest <input file> <output file> \n´); exit (0);
   }
   data = Input_PBM (argv[1]);
   if (data == NULL)
   {
          printf (`Bad input file '%s'\n´, argv[1]); exit (1);
   }
/* Embed input into a slightly larger image */
   im = newimage (data->info->nr+2, data ->info->nc+2);
   for (i=0; i<data->info->nr; i++)
     for (j=0; j<data->info->nc; j++)
       if (data->data[i] [j]) im->data[i+1] [j+1] = 1;
        else im->data[1+1] [j+1] = 0;

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

/* Pre_process */
   pre_smooth (im);
   aae (im);
```

```
/* Thin */
   thnz (im);

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

   Output_PBM (data, argv[2]);
}

/* Zhang-Suen with Holt's staircase removal */
void thnz (IMAGE im)
{
   int i, j, k, again=1;
   IMAGE tmp;
   tmp = newimage (im->info->nr, im->info->nc);

/* BLACK = 1, WHITE = 0. */
   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] = 0;
         else im->data[i][j] = 1;
       tmp->data[i] [j] = 0;
     }

/* Mark and delete */
   while (again)
     {
         again = 0;
/* Second sub-iteration */
         for (i=1; i<im->info->nr-1; i++);
          for (j=1; j<im->info->nc-1; j++)
         {
              if (im->data[i] [j] != 1) continue;
              k = nays8(im, i, j);
              if ((k >= 2 && k <= 6) && Connectivity(im, i, j) == 1)
              {
               if (im->data[i] [j+1] *im->data[i-1] [j]
                           *im->data[i] [j-1] == 0 &&
                      im->data[i-j][j] *im->data[i+1] [j]
                      *im->data[i] [j-1] == 0)
               {
                   tmp->data[i] [j] = 1;
                   again = 1;
               }
              }
          }



         Delete (im, tmp);
```

```
         if (again == 0) break;



/* First sub-iteration */
         for (i=1; i<im->info->nr-1; i++)
          for (j=1; j<im->info->nc-1; j++)
          {
              if (im->data[i] [j] != 1) continue;
              k = nays8(im, i, j);
              if ((k >= 2 && k <= 6) && Connectivity(im, i, j)==1)
              {
              if (im->data[i-1] [j] *im->data[i] [j+1]
                          *im->data[i+1] [j]==0 &&
                  im->data[i] [j+1] *im->data[i+1] [j] *im->data[i] [j-1] == 0)
              {
                  tmp->data[i] [j] = 1;
                  again = 1;
              }
              }
          }
       Delete (im, tmp);
   }

/* Post_process */
stair (im, tmp, NORTH);
        Delete (im, tmp);
        stair (im, tmp, SOUTH);
        Delete (im, tmp);

/* Restore levels */
   for (i=1; i<im->info->nr-1; i++)
     for (j=1; j<im->info->nc-1; j++)
          if (im->data[i] [j] > 0) im->data[i] [j] = 0;
          else im->data[i] [j] = 255;

   freeimage (tmp);
}

/* Delete any pixel in IM corresponding to a 1 in TMP*/
void Delete (IMAGE im, IMAGE tmp)
{
     int i, j;

/* Delete pixels that are marked */
   for (i=1; i<im->info->nr-1; i++)
     for (j=1; j<im->info->nc-1; j++)
       if (tmp->data[i] [j])
       {
          im->data[i] [j] = 0;
          tmp->data[i] [j] = 0;
        }
```

```
/* Number of neighboring 1 pixels*/
int nays8 (IMAGE im, int r, int c)
{
     int i, j, k=0;

   for (i=r-1; i<=r+1; i++)
     for (j=c-1; j<=c+1; j++)
       if (i !=r || c !=j)
        if (im->data[i] [j] >= 1) k++;
   return k;
}
/* Number of neighboring 0 pixels*/
int snays (IMAGE im, int r, int c)
{
      int i, j, k=0;
   for (i=r-1; i<=r+1; i++)
     for (j=c-1; j<=c+1; j++)
       if (i !=r || c !=j)
        if (im->data[i] [j] == 0) k++;
   return k;
}

/* Connectivity by counting black-white transitions on the boundary */
       int Connectivity (IMAGE im, int r, int c)
{
     int i, N=0;
     if (im->data[r] [c+1] >= 1 && im->data[r-1] [c+1] == 0) N++;
     if (im->data[r-1] [c+1] >= 1 && im->data[r-1] [c] == 0) N++;
     if (im->data[r-1] [c] >= 1 && im->data[r-1] [c-1] == 0) N++;
     if (im->data[r-1] [c-1] >= 1 && im->data[r] [c-1] == 0) N++;
     if (im->data[r] [c-1] >= 1 && im->data[r+1] [c-1] == 0) N++;
     if (im->data[r+1] [c-1] >= 1 && im->data[r+1] [c] == 0) N++;
     if (im->data[r+1] [c] >= 1 && im->data[r+1] [c+1] == 0) N++;
     if (im->data[r+1] [c+1] >= 1 && im->data[r] [c+1] == 0) N++;

     return N;
}

/* Stentiford's boundary smoothing method*/
void pre_smooth (IMAGE im)
{
     int, i, j;
     for (i=0; i<im->info->nr; i++)
       for (j=0; j<im->info->nc; j++)
         if (im->data[i] [j] = = 0)
          if (snays (im, i, j) <= 2 && Yokoi (im, i, j)<2)
            im->data[i] [j] = 2;

     for (i=0; i<im->info->nr; i++)
       for (j=0; j<im->info->nc; j++)
         if (im->data[i] [j] = = 2) im->data[i] [j] = 1;
}
```

```
/* Stentiford's Acute Angle Emphasis*/
void aae (IMAGE im)
{
     int i, j, again = 0, k;

     again = 0;
     for (k=5; k> = 1; k-=2)
     {
       for (i=2; i<im->info->nr-2; i++)
         for (j=2; j<im->info->nc-2; j++)
           if (im->data[i] [j] == 0)
                 match_du (im, i, j, k);

       for (i = 2; i<im->info->nr-2; i++)
         for (j=2; j<im->info->nc-2; j++)
           if (im->data[i] [j] = = 2)
           {
                  again = 1;
                  im->data[i] [j] = 1;
           }
       if (again = = 0) break;
     }
}

/* Template matches for acute angle emphasis*/
void match_du (IMAGE im, int r, int c, int k)
{

/*D1 */
     if (im->data[r-2] [c-2] == 0 && im->data[r-2] [c-1] == 0 &&
         im->data[r-2] [c] == 1 && im->data[r-2] [c+1] == 0 &&
         im->data[r-2] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 1 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 0 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r+2] [c-1] == 0 &&
         im->data[r+2][c] == 0 && im->data[r+2] [c+1] == 0)
     {
           im->data[r] [c] = 2;
           return;
     }

/* D2*/
     if (k >= 2)
     if (im->data[r-2] [c-2] == 0 && im->data[r-2] [c-1] == 1 &&
         im->data[r-2] [c] == 1 && im->data[r-2] [c+1] == 0 &&
```

```
         im->data[r-2] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 1 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0
         im->data[r+1] [c] == 0 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r+2] [c-1] == 0 &&
         im->data[r+2] [c] == 0 && im->data[r+2] [c+1] == 0)
{
           im->data[r] [c] = 2;
           return;
}

/* D3 */
     if (k>=3)
     if (im->data[r-2] [c-2] == 0 && im->data[r-2] [c-1] == 0 &&
         im->data[r-2] [c] == 1 && im->data[r-2] [c+1] == 1 &&
         im->data[r-2] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 1 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         i->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1][c] == 0 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r+2] [c-1] == 0 &&
         im->data[r+2] [c] == 0 && im->data[r+2] [c+1] == 0
}
{
           im->data[r] [c] = 2;
           return;
}

/* D4 */
     if (k>=4)
     if (im->data[r-2] [c-2] == 0 && im->data[r-2] [c-1] == 1 &&
         im->data[r-2] [c] == 1 && im->data[r-2] [c+1] == 0 &&
         im->data[r-2] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 1 &&
         im->data[r-1] [c] == 1 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 0 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
```

```
          im->data[r+2] [c-1] == 0 &&
          im->data[r+2] [c] == 0 && im->data[r+2] [c+1] == 0)
{
            im->data[r] [c] = 2;
            return;
}



/* D5 */
     if (k>=5)
     if (im->data[r-2] [c-2] == 0 && im->data[r-2] [c-1] == 0 &&
         im->data[r-2] [c] == 1 && im->data[r-2] [c+1] == 1 &&
         im->data[r-2] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 1 && im->data[r-1] [c+1] == 1 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 0 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r+2] [c-1] == 0 &&
         im->data[r+2] [c] == 0 && im->data[r+2] [c-1] == 0 )
     {
           im->data[r] [c] = 2;
           return;
     }

/* U1 */
     if (im->data[r+2] [c-2] == 0 && im->data[r+2] [c-1] == 0 &&
         im->data[r+2] [c] == 1 && im->data[r+2] [c+1] == 0 &&
         im->data[r+2] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 1 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0)
     {
           im->data[r] [c] = 2;
           return;
     }

/* U2 */
     if (k>=2)
     if (im->data[r+2] [c-2] == 0 && im->data[r+2] [c-1] == 1 &&
```

```
         im->data[r+2] [c] == 1 && im->data[r+2] [c+1] == 0 &&
         im->data[r+2] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 1 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r-2] [c-1] == 0 &&
         im->data[r-2] [c] == 0 && im->data[r-2] [c+1] == 0)
     {
           im->data[r] [c] = 2;
           return;
     }

/* U3 */
     if (k>=3)
     if (im->data[r+2] [c-2] == 0 && im->data[r+2] [c-1] == 0 &&
         im->data[r+2] [c] == 1 && im->data[r+2] [c+1] == 1 &&
         im->data[r+2] [c+2] == 0 &&
         im->data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
         im->data[r+1] [c] == 1 && im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0 &&
         im->data[r-1] [c+2] == 0 &&
         im->data[r-2] [c-1] == 0 &&
         im->data[r-2] [c] == 0 && im->data[r-2] [c+1] == 0)
     {
           im->data[r] [c] = 2;
           return;
     }

/* U4 */
     if (k>=4)
     if (im->data[r+2] [c-2] == 0 && im->data[r+2] [c-1] == 1 &&
         im->data[r+2] [c] == 1 && im->data[r+2] [c+1] == 0 &&
         im->data[r+2] [c+2] == 0 && im->data[r+1] [c-2] == 0 &&
         im->data[r+1] [c-1] == 1 &&
         im->data[r+1] [c] == 1 &&
         im->data[r+1] [c+1] == 0 &&
         im->data[r+1] [c+2] == 0 &&
         im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
         im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
         im->data[r] [c+2] == 0 &&
         im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
         im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0 &&
```

```
          im->data[r-1] [c+2] == 0 &&
          im->data[r-2] [c-1] == 0 &&
          im->data[r-2] [c] == 0 && im->data[r-2] [c+1] == 0
     {
           im->data[r] [c] = 2;
           return;
     }

/* U5 */
     if (k>=5)
     if (im->data[r+2] [c-2] == 0 && im->data[r+2] [c-1] == 0 &&
          im->data[r+2] [c] == 1 && im->data[r+2] [c+1] == 1 &&
          im->data[r+2] [c+2] == 0 &&
          im+>data[r+1] [c-2] == 0 && im->data[r+1] [c-1] == 0 &&
          im->data[r+1] [c] == 1 && im->data[r+1] [c+1] == 1 &&
          im->data[r+1] [c+2] == 0 &&
          im->data[r] [c-2] == 0 && im->data[r] [c-1] == 0 &&
          im->data[r] [c] == 0 && im->data[r] [c+1] == 0 &&
          im->data[r] [c+2] == 0 &&
          im->data[r-1] [c-2] == 0 && im->data[r-1] [c-1] == 0 &&
          im->data[r-1] [c] == 0 && im->data[r-1] [c+1] == 0 &&
          im->data[r-1] [c+2] == 0 &&
          im->data[r-2] [c-1] == 0 &&
          im->data[r-2] [c] == 0 && im->data[r-2] [c+1] == 0)
}
     {
           im->data[r] [c] = 2;
           return;
     }
}
/* Yokoi's connectivity measure*/
int Yokoi (IMAGE im, int r, int c)
{
     int N[9];
     int i, j, k, i1, i2;
     N[0] = im->data[r] [c] != 0;
     N[1] = im->data[r] [c+1] != 0;
     N[2] = im->data[r-1] [c+1] != 0;
     N[3] = im->data[r-1] [c] != 0;
     N[4] = im->data[r-1] [c-1] != 0;
     N[5] = im->data[r] [c-1] != 0;
     N[6] = im->data[r+1] [c-1] != 0;
     N[7] = im->data[r+1] [c] != 0;
     N[8] = im->data[r+1] [c+1] != 0;
     k = 0
     for (i=1; i<=7; i+=2)
     {
       i1 = i+1; if (i1 > 8) i1 -= 8;
       i2 = 1+2; if (i2 > 8) i2 -= 8;
       k += (N[i] - N[i] *N[i1] *N[i2]);
     }
```

```
     return k;
}

/* Holt's staircase removal stuff*/
void check (int v1, int v2, int v3)
{
     if (!v2 && (!v1 || (v3)) t00 = TRUE;
     if ( v2 && ( v1 || v3)) t11 = TRUE;
     if ( (!v1 && v2) || (!v2 && v3) )
     {
         t01s = t01;
         t01 = TRUE;
     }
}
int edge (IMAGE im, int r, int c)
{
     if (im->data[r] [c] == 0) return 0:
     t00 = t01 = t01s = t11 = FALSE;

/* CHECK (vNW, vN, vNE) */
     check (im->data[r-1] [c-1], im->data[r-1] [c], im
                ->data[r-1] [c+1]);

/* CHECK (vNE, vE, vSE) */
check (im->data [r-1][c+1], im->data[r] [c+1],
                 im->data[r+1] [c+1]);

/* CHECK (vSE, vS, vSW) */
     check (im->data [r+1] [c+1], im->data [r+1] [c],
                im->data [r+1] [c-1]);

/* CHECK (vSW, vW, vNW) */
     check (im->data [r+1] [c-1], im->data [r] [c-1],
                im->data [r-1] [c-1]);

     return t00 && t11 && !t01s;
}
void stair (IMAGE im, IMAGE tmp, int dir)
{
   int i, j;
   int N, S, E, W, NE, NW, SE, SW, C;

   if (dir == NORTH)
   for (i=1; i<im->info->nr-1; i++)
     for (j=1; j<im->info->nc-1; j++)
     {
      NW = im->data[i-1] [j-1]; N = im->data[i-1] [j]; NE = im->data[i-1] [j+1];
      W = im->data[i] [j-1]; c = im->data[i] [j]; E =
im->data[i] [j+1];
       SW = im->data[i+1][j-1]; S = im->data[i+1] [j]; SE = im->data[i+1] [j+1];
       if (dir == NORTH)
```

```
      {
       if (C && ! (N &&
                  ((E && !NE && !SW && (!W || !S)) ||
                   (W && !NW && !SE && (!E || !S)) )) )
          tmp->data[i] [j] = 0;/* Survives */
      .else
        tmp->data[i] [j] = 1;
      }else if (dir == SOUTH)
      {
       if (C && ! (S &&
                   ( (E && !SE && !NW && (!W || !N)) ||
                    (W && !SW && !NE && (!E || !N)) )) )
        tmp->data[i] [j] = 0;/* Survives */
       else
        tmp->data[i] [j] = 1;
      }
     }
}
```

6.8 Website Files

`medialaxis.exe`	Blum's medial axis transform
`stentiford.exe`	Stentiford's thinning method
`choi.c`	Choi et al. thinning
`contour1.c`	Pavlidis contour thinning
`contour2.c`	Pavlidis contour thinning, with pre- and post-processing
`medialaxis.c`	Blum medial axis
`stentiford.c`	Stentiford thinning
`zhangsuen.c`	Basic Zhang-Suen
`zhangsuenbest.c`	Source code, Zhang-Suen with pre- and post-processing
`5.pbm`	Test image, digit *5*
`5r.pbm`	Test image, digit *5*, level reversed from `5.pbm`
`B.pbm`	Test image, letter *B*
`H.pbm`	Test image, letter *H*
`T.pbm`	Test image, letter *T*
`V.pbm`	Test image, letter *V*
`X.pbm`	Test image, letter *X*

6.9 References

Arcelli, C. ''Pattern Thinning by Contour Tracing.'' *Computer Graphics and Image Processing* 17 (1981): 130–144.

Blum, H. ''A Transformation for Extracting New Descriptors of Shape.'' *Symposium Models for Speech and Visual Form.* Weiant Whaten-Dunn, ed. Cambridge: MIT Press, 1967.

Bookstein, F. L. ''The Line Skeleton.'' *Computer Graphics and Image Processing* 11 (1979): 123–137.

Brunner, D. and G. Brunnett. ''Fast Force Field Approximation and Its Application to Skeletonization of Discrete 3D Objects.'' G. Drettakis and R. Scopigno, ed. *EUROGRAPHICS 2008* 27, no. 2 (2008).

Chen, Y. and W. Hsu. ''A Modified Fast Parallel Algorithm for Thinning Digital Patterns.'' *Pattern Recognition Letters* 7 (1988): 99–106.

Chen, Y. and W. Hsu. ''An Interpretive Model of Line Continuation in Human Visual Perception.'' *Pattern Recognition* 22, no. 5 (1988): 619–639.

Choi, W., Lam, K. and W. Siu. ''Extraction of the Euclidean Skeleton Based on a Connectivity Criterion.'' *Pattern Recognition* 36 (2003): 721–729.

Davis, E. R. and A. P. N. Plummer. ''Thinning Algorithms: A Critique and a New Methodology.'' *Pattern Recognition* 14 (1981): 53–63.

Guo, Z. and R. W. Hall. ''Parallel Thinning with Two-Subiteration Algorithms.'' *Communications of the ACM* 32, no. 3 (1989): 359–373.

Haralick, R. M. ''Performance Characterization in Image Analysis: Thinning, a Case in Point.'' *ICDAR 91*. Paper presented at the First International Conference on Document Analysis and Recognition, Saint-Malo, France, September 30–October 2, 1991.

Hilditch, C. J. ''Linear Skeletons from Square Cupboards.'' *Machine Intelligence IV*, B. Meltzer and D. Mitchie, ed. Edinburgh: University Press, 1969.

Holt, C. M., Stewart, A., Clint, M. and R. H. Perrott. ''An Improved Parallel Thinning Algorithm.'' *Communications of the ACM* 30 (1987): 156–160.

Jang, B. K. and R. T. Chin. ''Analysis of Thinning Algorithms Using Mathematical Morphology.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-12* (1990): 541–551.

Kwok, P. C. K. ''A Thinning Algorithm by Contour Generation.'' *Communications of the ACM* 31 (1988): 1314–1324.

Latecki, L., Li, Q., Bai, X., and W. Liu. ''Skeletonization Using SSM of the Distance Transform.'' Proceedings of the International Conference on Image Processing, ICIP 2007, San Antonio, Texas, September 16-19, 2007.

Manocha, M. ''Fast Polygon Triangulation based on Seidel's Algorithm.'' *Graphics Gems V.* A. Paeth, ed. New York: Academic Press, 1995.

Martinez-Perez, M., Javier Jiménez, J. and J. Navalon. ''A Thinning Algorithm Based on Contours.'' *Computer Vision, Graphics, and Image Processing* 39 (1987): 186–201.

Melhi, M., Ipson, S., and W. Booth. ''A Novel Triangulation Procedure for Thinning Hand-Written Text.'' *Pattern Recognition Lett*. 22 (2001): 1059–1071.

Montanari, U. ''A Method For Obtaining Skeletons Using a Quasi-Euclidian Distance.'' *Journal of the ACM* 15 (1968): 600–624.

Montanari, U. ''Continuous Skeletons from Digitized Images.'' *Journal of the ACM* 16. (1969): 534–549.

Morrison, P. and J. Zou. ''Skeletonization Based on Error Reduction.'' *Pattern Recognition* 39 (2006): 1099–1109.

Ogniewicz, R., and O. Kubler. ''Hierarchic Voronoi Skeletons.'' *Pattern Recognition* 28 (1995): 343–359.

Pal, S. K. ''Fuzzy Skeletonization of an Image.'' *Pattern Recognition Letters* 10 (1989): 17–23.

Parker, J. R. and C. Jennings. ''Defining the Digital Skeleton.'' Paper presented at SPIE Conference on Vision Geometry, Boston, MA, 1992.

Pavlidis, T. *Algorithms for Graphics and Image Processing*. Rockville, Maryland: Computer Science Press, 1982: p. 416.

Piper, J. ''Efficient Implementation of Skeletonization Using Interval Coding.'' *Pattern Recognition Letters* 3 (1985): 389–397.

Sinha, R. M. K. ''A Width-Independent Algorithm for Character Skeleton Estimation.'' *Computer Vision, Graphics, and Image Processing* 40 (1987): 388–397.

Sossa, J. H. ''An Improved Parallel Algorithm for Thinning Digital Patterns.'' *Pattern Recognition Letters* (1989): 77–80.

Stefanelli, R. ''A Comment on an Investigation into the Skeletonization Approach of Hilditch.'' *Pattern Recognition* 19 (1986): 13–14.

Stentiford, F. W. M. and R. G. Mortimer. ''Some New Heuristics for Thinning Binary Handprinted Characters for OCR.'' *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13, no. 1 (January/February 1983): 81–84.

Suzuki, S. and K. Abe. ''Sequential Thinning of Binary Pictures Using Distance Transformation.'' Paper presented at Eighth International Conference on Pattern Recognition, Paris, 1986.

Suzuki, S. ''Binary Picture Thinning by an Iterative Parallel Two-Subcycle Operation.'' *Pattern Recognition* 20 (1987): 297–307.

Xia, Y. ''Skeletonization VIA the Realization of the Fire Front's Propagation and Extinction in Digital Binary Shapes.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11* (1989): 1076–1086.

Ye, Q.Z. ''The Signed Euclidean Distance Transform and Its Applications.'' Paper presented at proceedings of the Ninth International Conference on Pattern Recognition, Rome, 1988, 495–499.

Yokoi, S., Toriwaki, J. and T. Fukumura. ''Topological Properties in Digitized Binary Pictures.'' *Systems Computer Controls* 4 (1973): 32–39.

Zhang, S. and K. S. Fu. ''A Thinning Algorithm for Discrete Binary Images.'' Paper presented at proceedings of the International Conference on Computers and Applications, Beijing, China, 1984, 879–886.

Zou, J., Chang, H., and H. Yan. ''Shape Skeletonization by Identifying Discrete Local Symmetries.'' *Pattern Recognition* 34 (2001): 1895–1905.

CHAPTER
7

Image Restoration

7.1 Image Degradations—The Real World

Anyone who has ever taken a photograph will understand that capturing an image exactly as it appears in the real world is very difficult, if not impossible. There is noise to contend with, which in the case of photography is caused by the graininess of the emulsion, or the resolution and quantization of the image sensor motion blur, focus problems, depth-of-field issues, and the imperfect nature of even the best lens system. The result of all these *degradations* is that the image (photograph) is an approximation of the scene.

Often the image is good enough for the purpose for which it was produced. On the other hand, there are some instances where the correction of an image by computer is the only way to obtain a usable picture. The original problems with the Hubble Space Telescope are a case in point; the optics produced images that did not approach the potential of the telescope, and a repair mission was not immediately possible. Computers were used to repair some of the distortion caused by the optics and give images that were of high quality.

Image restoration is the art and science of improving the quality of an image based on some absolute measure. It usually involves some means of undoing a distortion that has been introduced, such as motion blur or film graininess. This can't be done in any perfect way, but vast improvements are possible in some circumstances.

The techniques of image restoration are very mathematical in nature, and this may distress some people who are interested in the subject. The purpose of this section is to provide insight, and so a very practical approach is taken.

The mathematics will be skimmed over, and readers interested in the details will find references at the end of this chapter to explore further. However, not all the math can be eliminated.

The example of the Hubble Space Telescope is very relevant, since it is an ideal way to introduce a technique for characterizing the distortion inherent in an image. A star, when viewed through a telescope, should be seen as a perfect point of light. Ideally, all the light energy of the star would be focused on a single pixel. In practice this is not so, because the distortions of the atmosphere and the telescope optics will yield a slightly blurred image in which the central pixel is brightest, and a small region around it is less bright, but brighter than the background. The distortions that have been inflicted on the point image of the star are reflected in the shape and intensity distribution of the star's image. All stars (for a reasonable optical system) will have the same distortions imposed upon them; indeed, all points on the image have been replaced by these small blobs, and the sum of all the blobs is the sampled image.

The effect that an image acquisition system has on a perfect point source is called the *point spread function* (PSF). The sample image has been produced by convolving the PSF with the perfect image, so that the same blur exists at all points. Figure 7.1 shows a diagrammatic view of how distortion and noise have been applied to the original image to give the sampled, observed image. To obtain the perfect image given the sampled one is the goal of restoration, and it is not generally possible. We therefore wish to improve the image as much as possible, and the PSF tells us what has been done to the image. The ideal solution is to *deconvolve* the image and the PSF, but this can only be done approximately and at some significant expense.

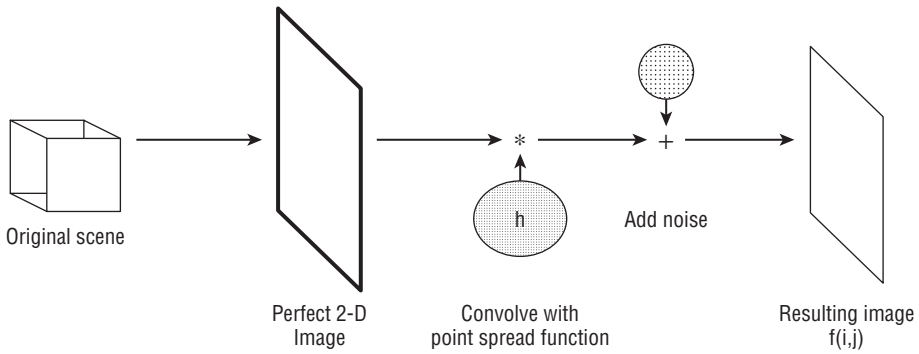


Figure 7.1: One model of how a perfect image becomes distorted by imperfect (real) acquisition systems.

This discussion assumes that the PSF is the same at all points of the image, in which case the system is said to be spatially invariant. This is the situation generally assumed in the literature, but the Space Telescope was not spatially invariant. In cases like this the solution is to assume that the PSF is almost

constant over a small region of the image, and to restore the image in pieces using very similar techniques to those that will be discussed.

The first thing to do is to see if a blurred image can be created artificially, by convolving a known PSF with a created image. Then methods of reducing the blur can be applied to these known images, and the results can be assessed objectively. Since it is important to know the PSF, methods of estimating it from a distorted image must also be discussed, as will certain special cases (such as motion blur) for which specific restoration schemes have been devised.

A key tool in the analysis and restoration of images is the *Fourier transform*. This is a mathematical tool devised in the mid-twentieth century and based on the Fourier series, which was itself devised more than 200 years ago. Its goal is to determine how much of each possible frequency occurs in a specific signal. It was first used to analyze sound waves and like signals that were specifically composed of the sum of many sine wave-like signals, but the use has been expanded to include other kinds of signals and other kinds of frequencies. Because the Fourier transform is so crucial to further work on image restoration, it needs to be examined in more detail.

7.2 The Frequency Domain

A convolution can be carried out directly on an image by moving the convolution matrix (image) so that it is centered on each pixel of the image in turn, then multiplying the corresponding elements and summing the products. This was described in Equation 2.13, for example. This is a time-consuming process for large images, and one that can be speeded up by using the Fourier transform.

A *transform* is simply a mapping from one set of coordinates to another. For example, a *rotation* is a transform; the rotated coordinate system is different from the original, but each coordinate in the original image has a corresponding coordinate in the rotated image. The *Hough transform* is another example, in which pixel coordinates (i,j) are converted into coordinates (m,b) representing the slope and intercept of the straight lines that pass through the pixel.

The Fourier transform converts spatial coordinates into frequencies. Any curve or surface can be expressed as the sum of some number (perhaps infinitely many) of sine and cosine curves. In the *Fourier domain* (called the *frequency domain* as well) the image is represented as the parameters of these sine and cosine functions. The Fourier transform is the mathematical mechanism for moving into and out of the frequency domain.

The frequency domain is so named because the two parameters of a sine curve are the amplitude and the frequency. The fact that an image can be converted into a frequency domain representation implies that the image can contain high-frequency or low-frequency information; this is true. If the grey level of some portion of the image changes slowly across the columns, then

it would be represented in the frequency domain as a sine or cosine function having a low frequency. Something that changes quickly, such as an edge, will have high-frequency components.

It is therefore possible to build filters that will remove or enhance certain frequencies in the image, and this will sometimes have a restorative effect. Indeed, noise consists of mainly high-frequency information, and so filtering out of the very high frequencies should have a noise reduction effect. It unfortunately also has an edge-reduction effect.

There are many other reasons to use a Fourier transform. A convolution can be carried out directly on an image by moving the convolution matrix (image) so that it is centered on each pixel of the image in turn, and then multiplying the corresponding elements and summing the products. This was described in Equation 2.13, for example. It is a time-consuming process for large images, and one that can be speeded up greatly by using the Fourier transform.

7.2.1 The Fourier Transform

The Fourier transform breaks up an image (or, in one dimension, a signal) into a set of sine and cosine components. It is important to keep these components separate, and so a vector of the form (*cosine*, *sine*) is used at each point in the frequency domain image; that is, the values of the pixels in the frequency domain image are two component vectors. A convenient way to represent these is as *complex* numbers.

Each complex number consists of a real part and an imaginary part, which can be thought of as a vector. A typical complex number could be written as:

$$z = (x, jy) = x + jy \qquad \text{(EQ 7.1)}$$

where j is the imaginary number $\sqrt{-1}$. The exponential of an imaginary number can be shown to be the sum of a sine and cosine, which is exactly what we want:

$$e^{j\theta} = \cos\theta + j\sin\theta \qquad \text{(EQ 7.2)}$$

This polar form is what will be used from here on, but it is important to remember that it is really a shorthand for the sum of the sine and cosine parts. In one dimension, the Fourier transform of a continuous function $f(x)$ is:

$$F(w) = \int_{t=0}^{\infty} f(t)\, e^{-jwt} dt \qquad \text{(EQ 7.3)}$$

If the function has been sampled so that it is now discrete, the integral becomes a sum over all the sampled points:

$$F(w) = \sum_{k=0}^{N-1} f(k)\, e^{\frac{2\pi jwk}{N}} \qquad \text{(EQ 7.4)}$$

This will be called the discrete Fourier transform (DFT), and is what is really calculated for sampled data like images when a Fourier transform is computed. If the function $f(k)$ is a sample sine curve, then the Fourier transform $F(w)$ should yield a single point showing the parameters of the curve. Figure 7.2a shows just such a sampled sine curve, which has the form

$$f(k) = 2\,\mathrm{Sin}\left(\frac{2\pi k}{128}\right) \qquad \text{(EQ 7.5)}$$

Figure 7.2b shows the Fourier transform of the curve. Note that it has a single peak at the point $w = 8$, which happens to correspond to the frequency of the original sine curve: eight cycles per 1024 pixels, or one cycle in 128 pixels. Figure 7.2c and d show a pair of sine curves and their Fourier transform, which has two peaks (one per sine curve).

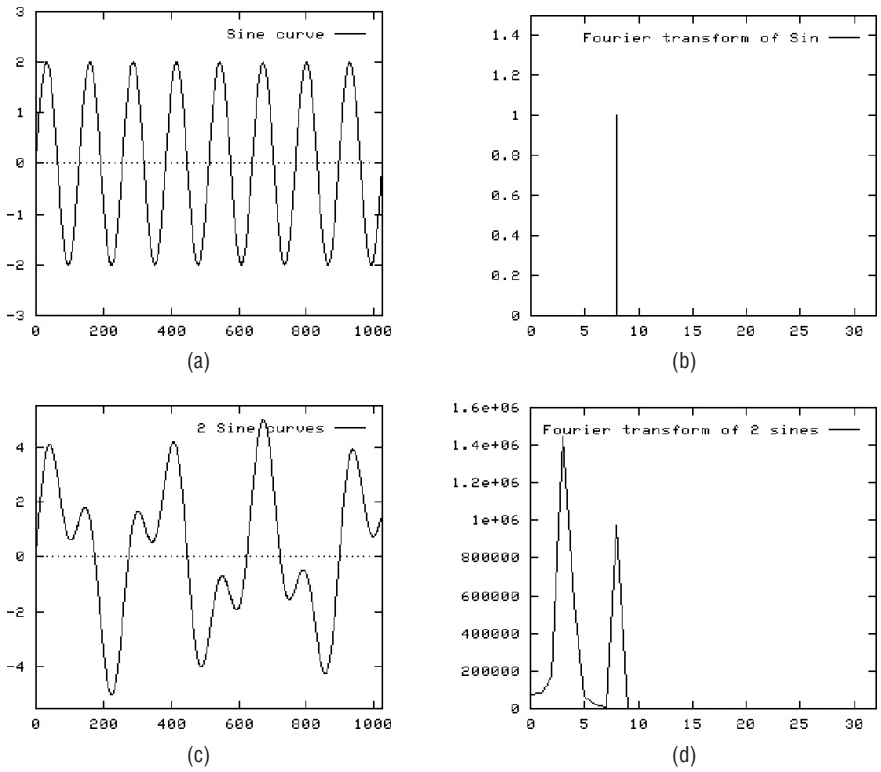


Figure 7.2: Simple one-dimensional signals and their Fourier transforms. (a) Sine function with a period of 128 pixels. (b) Fourier transform, showing a peak at 8 for a signal of duration 1024, giving eight periods/duration. (c) Sum of two sine curves: period = 128 + period = 300. (d) Fourier transform, showing two peaks, one per sine function.

The Fourier transforms shown in Figure 7.2 were computed by the C procedure `slow.c`, which uses Equation 7.4 to do the calculation. This does

work fine, but is very slow, and it would not likely be used in any real system. Because of the very useful nature of the Fourier transform, an enormous effort has been extended to make it computationally fast. The essential C code from `slow.c` is shown in Figure 7.3, this can be compared with the code for the fast Fourier transform (FFT), which will be discussed in the next section. The complex numbers are implemented as structures, each having a real and imaginary component.

```
void slowft (float *x, complex *y,
int n)
{
     COMPLEX tmp, z1, z2, z3, z4;
     int m, k;

     cmplx(0.0,atan(1.0)/n* -8.0,&tmp);
     for (m = 0; m<=n/2; m++)
     {
       y[m].real=x[0]; y[m].imag=0.0;
       for (k=1; k<=n-1; k++)
       {
/* Exp (tmp*k*m) */
           cmplx ((float)k, 0.0, &z2);
           cmult (tmp, z2, &z3);
           cmplx ((float)m, 0.0, &z2);
           cmult (z2, z3, &z4);
           cexp (z4, &z2);
/* *x[k] */
           cmplx (x[k], 0.0, &z3);
           cmult (z2, z3, &z4);
/* + y[m] */
           csum (y[m], z4, &z2);
           y[m].real = z2.real;
           y[m].imag = z2.imag;
       }
     }
}

viod cmplx(float rp, float ip, COMPLEX
*z)
{
     z->real = rp;
     z->imag = ip;
}
```

```
void cexp (COMPLEX z1, complex *res)

{
     COMPLEX x, y;

  x.real = exp((double)z1.real);
  x.imag = 0.0;
  y.real=(float)cos((double)z1.imag);
  y.imag=(float)sin((double)z1.imag);
  cmult (x, y, res);
}

void cmult (COMPLEX z1, COMPLEX z2,
            COMPLEX *res)
{
  res->real = z1.real*z2.real-
           z1.imag*z2.imag;
  res->imag=z1.real*z2.imag +
           z1.imag*z2.real;
}

void csum (COMPLEX z1, COMPLEX z2,
COMPLEX *res)
{
    res->real = z1.real + z2.real;
    res->imag = z1.imag + z2.imag;
}

float cnorm (COMPLEX z)
{
  return z.real*z.real +z.imag*z.imag;
}
```

Figure 7.3: Source code for the obvious implementation of the Fourier transform.

7.2.2 The Fast Fourier Transform

Make no mistake, the FFT is simply a faster way to compute the Fourier transform, and is not a new or different transform in its own right [Cooley, 1965]. The optimizations needed to speed up the calculation are partly standard programming tricks (such as computing some of the values in advance outside

of the loop) and partly mathematical techniques. There are a number of very good references on the FFT [Bracewell, 1965; Brigham, 1974], but these deal rather rigorously with the subject. Here, the code in Figure 7.3 will be successively improved until it implements the basic FFT method.

The first optimization involves moving the exponential calculation (cexp) to a position outside of the inner loop. This is done by precomputing all the N possible products:

$$F(w) = \sum_{k=0}^{n-1} f(k)\ e^{(-2\pi j/n)(wk)} f(k) = \sum_{k=0}^{n-1} pre[wk \bmod n]\ f(k) \qquad \text{(EQ 7.6)}$$

This reduces the strength of the operation within the loop from a complex exponentiation to a complex product. For the transform computed in Figure 7.2a–b, for example, the program (Figure 7.3) requires about 5 times the CPU time as does the program `slow2.c` (Figure 7.4), which uses precomputed exponentials.

```
void slowft(float *x, COMPLEX *y, int n)
{
  COMPLEX tmp,z1, z2, z3, z4, pre[1024];
  int m, k, i, p;

/* Constant factor -2 pi */
  cmplx (0.0, atan(1.0)/n * -8.0, &z1);
  cexp (z1, &tmp);

/* Precompute most of the exponential */
  cmplx (1.0, 0.0, &z1);/*z1=1.0; */
  for (i=0; i<n; i++)
  {
    cmplx(z1.real, z1.imag, &(pre[i]));
    cmult (z1, tmp, &z3);
    cmplx (z3.real, z3.imag, &z1);
  }
```

```
/* Compute all Y values */

  for (m = 0; m<=n; m++)
  {
      cmplx (x[0], 0.0, &(y[m]));
      for (k=1; k<=n-1; k++)
      {
         p = (k*m % n);
         cmplx (x[k], 0.0, &z3);
         cmult (z3, pre[p], &z4);
         csum (y[m], z4 &z2);
         y[m].real = z2.real;
         y[m].imag = z2.imag;
      }
     }
}
```

Figure 7.4: The program slow2, obtained from slow by precomputing the complex exponential entries.

The next step involves the mathematical observation that the even-numbered elements can be computed separately from the odd ones. In cases where n is even, this will reduce the number of multiplications by half. The even coefficients are found using:

$$\begin{aligned} F(2a) &= \sum_{k=0}^{n/2-1} e^{(-2\pi j/n)(ak)}\ Sum\,[k] \\ Sum[k] &= \frac{(f(k)+f(k+m))}{2} \end{aligned} \qquad \text{(EQ 7.7)}$$

where a runs from 0 to $n/2 - 1$. Similarly, the odd elements are found by:

$$F(2a+1) = \sum_{k=0}^{n/2-1} e^{(-2\pi j/n)(ak)}\ Diff\ [k]$$
$$Diff\ [k] = \frac{(f(k) - f(k+m))e^{(-2\pi j/n)k}}{2} \qquad \text{(EQ 7.8)}$$

The `Sum` and `Diff` arrays can be calculated in advance. The odd and even parts of the Fourier transform are computed separately, and then merged into a common matrix F. The Fourier transform of Figure 7.2 requires 0.81 seconds using this enhancement (program `slow3.c`, Figure 7.5). In the code shown in the figure, the function `evenodd` finds the `Sum` and `Diff` arrays, and the previous version of the FTT (`slow2.c`) is called to compute the actual transform.

```
void evenodd (float *x, COMPLEX *y, int n)
{
  int m, i;
  COMPLEX Sum[1024],Diff[1024], z1, z2;
  COMPLEX Even[512], Odd[512], tmp;
  float xs, ys;

  m = n/2;
  cmplx (0.0, atan (1.0)/n * -8.0, &z1);
  cexp (z1, &tmp);
  cmplx (1.0, 0.0, &z1);

  for (i=0; i<m; i++)
  {
    xs = (x[i] + x[i+m])/2.0;
    ys = (x[i] - x[i+m])/2.0;
    cmplx (xs, 0.0, &(Sum[i]));
    cmplx (ys, 0.0, &z2);
    cmult (z1, z2, &(Diff[i]));
    cmult (z1, tmp, &z2);
    cmplx(z2.real,z2.imag, &z1);
  }

  slowft (x, Even, m);
  slowft (x, Odd,  m);

  for (i=0; i=<m; i++)
  {
     y[i<<1] = Even[i];
     y[i<<1 + 1] = Odd[i];
  }
}
```

Figure 7.5: The program slow3, in which the odd and even transform elements are computed separately.

Now comes an obvious step, but one that restricts the input data set even further. It seems obvious that if the number of elements is a *power of two* rather than simply being even, then the number of even elements is still even and can profit from a repeated application of Equation 7.7. Ditto, of course, for the odd elements, down to the point where there is only one element, which is a very simple case. This means that the calculation now reduces to the calculations of Equations. 7.7 and 7.8, followed by a recursive transform calculation of the odd and even halves, and then the merging of the two halves.

With this step, the basic FFT algorithm is in place. There are more optimizations that are often associated with a good FFT procedure, but the essentials are present in the program `slow4.c` (Figure 7.6); this program requires only 0.07 seconds to complete the Fourier transform we have been using as an example.

```
void slowft(float *x,COMPLEX *y, int n)
{
     COMPLEX xx[1024];
     int i;

     for (i=0; i<n; i++)
       cmplx (x[i], 0.0, &(xx[i]));
     evenodd (xx, y, n);
}

void evenodd(COMPLEX *x,
                 COMPLEX *Y,int n)
{
  int m, i;
  COMPLEX *Sum, *Diff;
  COMPLEX *Even, *Odd;
  COMPLEX z1, z2, z3, z4, tmp, two;
  float xs, ys;

  if (n<1 printf ("what???? \n");

/* The simple case, where N=1 */
  cmplx (2.0, 0.0, &two);
  if (n == 1)
  {
    cmplx(x[0].real,x[0].imag,&(y[0]));
    return;
  }
/* Otherwise, N is even */
  m = n/2;
  cmplx (0.0,atan (1.0)/n* -8.0, &z1);
  cexp (z1, &temp);
  cmplx (1.0, 0.0, &z1);
/* Allocate temporary space */
  Sum = (COMPLEX *)
     malloc(sizeof (struct cpx)*m);
  Diff = (COMPLEX *)
     malloc (sizeof (struct cpx) *m);
Even = (COMPLEX *)
   malloc (sizeof (struct cpx) * m);
Odd = (COMPLEX *)
   malloc (sizeof (struct cpx) *m);
if (Sum==0 || Diff==0 ||
    Even==0 || Odd==0)
{
  printf ("Panic-memory.\n");
  exit(1);
}

for (i=0; i<m; i++)
{
  csum (x[i], x[i+m], &z3);
  cdiv (z3, two, &(Sum[i]));

  cdif (x[i], x[i+m], &z3);
  cmult (z3, z1, &z4);
  cdiv (z4, two, &(Diff[i]));

  cmult (z1, tmp, &z2);
  cmplx (z2.real, z2.imag, &z1);
}

evenodd (Sum, Even, m);
Evenodd (Diff, Odd, m);

for (i=0; i<m; i++)
{
     y[i*2].real = Even[i].real;
     y[i*2].imag = Even[i].imag;
     y[i*2 + 1].real = Odd[i].real;
     y[i*2 + 1].imag = Odd[i].imag;
}

free(Sum); free(Diff);
free(Even); free(Odd);
}
```

Figure 7.6: The program slow4, which is the essential code for the FFT.

OpenCV offers a convenient set of functions for computing the Fourier transform of images and signals. The accompanying website contains a Fourier transform library named `fftlib.c` that can be used in the remainder of this chapter instead of the OpenCV functions, and for which the source code is available. The basic FFT procedure in this library is called, simply, `fft`. The initialization procedure `fftinit` must be called first, passing the size of the data array to be transformed. There are many useful functions in this library, which will be described as the need arises.

7.2.3 The Inverse Fourier Transform

The inverse Fourier transform will undo the transformation; when given the Fourier transform of a set of data, the inverse transform will reconstruct the original data. The log function and the exp function have a similar relationship, where one undoes the other.

The formula for the discrete inverse Fourier transform is:

$$f(k) = \frac{1}{N} \sum_{w=0}^{N-1} F(w) e^{2\pi jwk/N} \qquad \text{(EQ 7.9)}$$

which differs from the forward transform in the sign of the exponent. Now might be the time to point out that the constant factor $1/n$ is somewhat flexible. Some people apply it to the forward transform, some split it between the forward and inverse transforms, and some apply it only to the inverse transform. In fact, the programs `slow1.c` – `slow4.c` produce somewhat different numerical results because the multiplicative constant was ignored utterly.

Of course, the fast algorithm can be applied to the inverse transform, so that a set of data can be transformed just as easily in either direction. In the `fftlib.c` routines, the inverse one-dimensional FFT function is the same as the forward FFT function; there is a parameter that specifies a forward or inverse direction.

7.2.4 Two-Dimensional Fourier Transforms

So far, the application of the Fourier transform to images has been vague, since the transforms seen so far apply to one-dimensional data only. The extension to two dimensions is simple; mathematically, we have:

$$F(u,v) = \frac{1}{\sqrt{nm}} \sum_{i=0}^{n-1} \sum_{k=0}^{m-1} e^{-2\pi j(ui+vk)/nm} f(i,k) \qquad \text{(EQ 7.10)}$$

The inverse transform is the same, but with the sign in the exponent reversed. Note that the constant multiplier is split between the forward and inverse transforms, which is not always done.

The Fourier transform of an image f is calculated by first computing the Fourier transform of each row, giving an image f'. Then the Fourier transform of each column of f' is computed, giving F, the transform of the image. This allows us to use the one-dimensional FFT methods already discussed to compute the two-dimensional transform. Figure 7.7 gives a sample 2D FFT routine based on this, which was in fact taken from `fftlib.c`.

```
void fft2d ( COMPLEX_IMAGE image,
             float direction )
{
     float temp[1024]; /* For columns */
     int i, j; /* Iteration counters */
     int d, nu;

     nu = vlog2(FFTN);
     if (direction == FORWARD)
      d = 0;
     else d = 1;

/* Transform Rows */
     for( i = 0; i < FFTN; i++ )
        fft (image[i], direction);

/* Transform Columns */
     for( i = 0; i < FFTN; i++ )
     {
           for( j = 0; j < FFTN; j++ )
           {
            temp[j] = image[j][i];
            temp[j+FFTN] =image[j][i+FFTN];
           }
           fft (temp, direction);
           for( j = 0; j < FFTN; j++ )
           {
            image[j][i] = temp[j];
            image[j][i+FFTN] = temp[j+FFTN];
           }
     }
}
```

Figure 7.7: The basic two-dimensional FFT. The row transformation is performed in place but the columns must be copied into consecutive locations in a temporary array. The direction parameter controls whether a forward or inverse transform is done.

Equation 7.10 describes the general case. For the purposes of image restoration, it will be assumed that the images are square $N \times N$ arrays, where N is a power of two. If this is not initially true, then the images can be enlarged by adding rows and columns containing all zero pixels until it is true.

In addition to the use of the Fourier transform to allow the filtering of certain frequencies, there are other useful properties that become quite important when discussing two-dimensional data. Most important from the point of view of image processing is that a convolution is much easier to do in the frequency domain than in the spatial domain. In fact, a convolution is simply an element-by-element product of the Fourier transforms of the two images

being convolved. Specifically, the frequency domain convolution of image a with image b is performed as follows:

1. Check the image sizes to ensure that they are the same. If not, add zeros to the smaller image until it is the same size as the larger one.
2. Compute A, the Fourier transform of the image a; also compute B, the Fourier transform of the image b.
3. Compute the new image C as the product of the corresponding pixels in A and B; that is, $C(i,j) = A(i,j) \times B(i,j)$.
4. Compute the image c, the inverse Fourier transform of C. This is the result of the convolution.

In spite of the complexity of the Fourier transform computation, this process is actually faster than the straightforward method of Equation 2.13 when the smaller of the two images is larger than about 16×16 pixels.

7.2.5 Fourier Transforms in OpenCV

The Fourier transform function in OpenCV is quite flexible, allowing single or multiple dimension transforms to be computed by the same routine. The basic call is:

```
cvDFT(a, b, flag, height);
```

where `a` is the input data; `b` is the output Fourier transform; `flag` is an integer whose value indicates a set of possible requests; and `height` is the number of data rows to be processed. If `a` and `b` point to the same data array, the calculation is done *in place*, and the input data is overwritten by the resulting transform.

The `flag` parameter has a few commonly used values:

- `CV_DXT_FORWARD` — Do a forward transform
- `CV_DXT_INVERSE` — Do an inverse transform
- `CV_DXT_SCALE` — Scale the result (divide by the number of elements)
- `CV_DXT_ROWS` — Treat the rows as distinct data vectors and calculate the transform of all of them (as opposed to computing a two-dimensional transform)

These can be combined using an OR operator. A commonly used example would be

```
(CV_DXT_INVERSE | CV_DXT_SCALE)
```

which would result in a scaled inverse transform.

The type of the data arrays merits some discussion. The `cvDFT` function accepts neither an image as a parameter nor a floating-point array. The input and output parameters are of type `CvMat *`, which are pointers to OpenCV matrix objects, or vectors in the one-dimensional case. This is an odd type, designed to implement a matrix of arbitrary nature. The type of the components and the matrix dimensions and size are fixed when created, but can be float or integer, complex, or double. Accessing the components is not as simple as for an array type, but can be in the same manner as are pixels in an `IplImage` type, by using an access function such as `cvSet1D` or `cvSet2D`.

As a simple example, here is an annotated program that calculates a one-dimensional Fourier transform of the sine wave of Equation 7.5. It prints the values of the transform, and then calculates the reverse transform and prints the difference between the original data values and the result of the forward and inverse transformation. In most cases, this difference is zero to six digits. This example shows how to use a `CvMat` data object and how to find the Fourier transform of a one-dimensional data set.

1. Declare the needed variables:

```
int i,j;
CvScalar s, s1;
double p;
CvMat* a;
```

2. Create a matrix 1024 elements in size. The constant `CV_32FC2` passed to `cvCreateMat` means a 32-bit floating-point complex data type for components:

```
a = cvCreateMat( 1024, 1, CV_32FC2 );
```

3. Initialize the array to the sine function $a_k = 2 \sin(2 * \pi * k/128)$, with the imaginary part 0. The function `cvSet1D` is used to set each value, just as `cvSet2D` is used in the case of setting an image pixel:

```
for (i=0; i<1024; i++)
{
    s.val[0] = 2.0 * sin (2.0*PI*i/128);
    s.val[1] = 0.0;
    cvSet1D(a, i, s);
}
```

4. Do the Fourier transform:

```
cvDFT(a,a,CV_DXT_FORWARD,0);
```

5. Print it, and note the max value at i=8:

```
for (i=0; i<256; i++)
{
     s  = cvGet1D (a, i); s1 = cvGet1D (a, i+256);
     printf ("%d %f   %d %f    ", i,s.val[0], i*2,s1.val[0]);
     s  = cvGet1D (a, i+512); s1 = cvGet1D (a, i+768);
     printf ("%d %f   %d %f\n", i, s.val[0], i*2, s1.val[0]);
}
scanf ("%d", &j);
```

6. Now back transform, with scaling:

```
cvDFT(a,a,CV_DXT_INVERSE_SCALE,0);
```

7. Finally, examine each element of the inverse transform and find the difference between it and the original data:

```
for (i=0; i<1024; i++)
{
     s = cvGet1D(a, i);
     p = 2.0 * sin (2.0*PI*i/128);
     printf ("%d  %lf  %lf  %lf\n", i,p,s.val[0],p-s.val[0]);
}
```

   The result of the forward transform followed by an inverse transform should be the original data, so the differences should be near 0. This program can be found on the website as `fft1d.c`.

7.2.6 Creating Artificial Blur

Image blur is accomplished by convolving an image with a PSF that represents the precise nature of the blur. A convolution can be performed in the frequency domain by computing the Fourier transform of the two images concerned, multiplying those transforms together pixel by pixel, and computing the inverse transform of the result. Thus, it should be a simple matter to introduce blur into some perfect images to obtain blurred versions that can be used for experimentation. Restoration methods can be tested on these known images, and the quality of the result can be determined by simply comparing the original against the restored image.

```
void fft2d ( COMPLEX_IMAGE image, float direction )
{
     float temp[1024];  /* For coloumns */
     int i, j;  /* Iteration counters */
     int d, nu;

     nu = vlog2(FFTN);
```

```
      if (direction == FORWARD)
       d = 0;
       else d = 1;

/* Transform Rows */
     for( i = 0; i < FFTN; i++ )
         fft (image[i], direction);

/* Transform Columns */
     for( i = 0; i < FFTN; i++ )
     {
        for( j = 0; j < FFTN; j++ )
        {
             temp[j] = image[j][i];
             temp[j+FFTN] =image[j][i+FFTN];
        }

        fft (temp, direction);
        for( j = 0; j < FFTN; j++ )
        {
             image[j][i] = temp[j];
             image[j][i+FFTN] = temp[j+FFTN];
        }
     }
}
```

Figure 7.8 shows a simple image that can be used for experimentation: It is a 128×128 image that simply contains the words *The Fourier Transform*. The PSF to be used is also shown. This particular PSF should blur the image equally in all directions, and is circular with a diameter of five pixels. The blurred version of the image (Figure 7.8c) is unreadable.

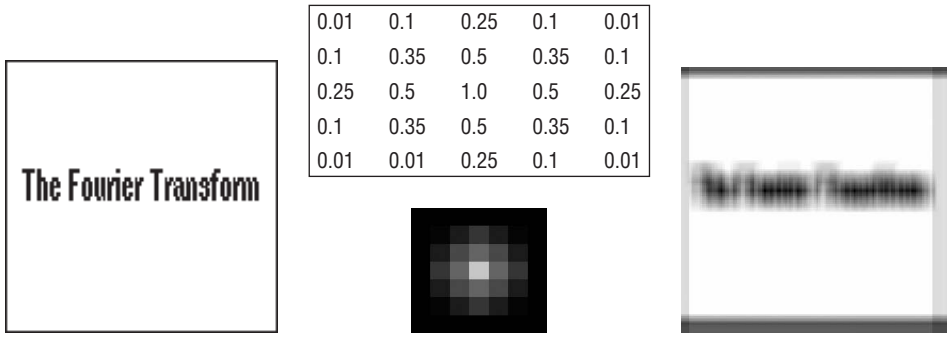


Figure 7.8: Blurring an image using a frequency domain convolution. (a) The original image. It was obtained using a screen capture, and is without noise. (b) The PSF used to generate the blur. (c) The result of convolving the image with the PSF.

It is important to note that the pixel values in the PSF image are all less than or equal to 1.0, since a PSF should not add energy (light, pixel levels) to the image; a PSF only spreads out the existing energy. If the pixel values in the PSF image were in the usual range of 0–255, the distortions introduced may be beyond the scope of image restoration to fix.

The blurred image was created using the `blur.c` program that is provided on the website. While discussing the various aspects of image restoration, we will be building a collection of software that actually performs the operations described. Each program will perform a specific task related to image restoration, and will illustrate a few of the basic functions needed. While each program is a stand-alone routine that accomplishes a specific task, the idea is to provide a collection of basic procedures that can be mixed and matched according to the specific needs of a task.

As the first program in the collection, `blur.c` introduces the largest number of new modules. All it really does is read in two image files, a source image and a PSF image, and convolve them as described previously. The main program describes this process in detail.

First, we have the basic declarations and `include` files:

```
#include <cv.h>
#include <highgui.h>
#include <stdlib.h>

int main( int argc, char** argv )
{
  IplImage* img = NULL;
  IplImage *psf, *psfa;
  IplImage *grayImg, *z, *zz;
  int i,j;
  CvScalar s, s1;
  double p, q;
```

Now the input image is read in. Change the path name to that of any file to be processed (or ask for a file name from input):

```
img=cvLoadImage(˜h:\\aipcv\\ch6\\face.pgm˝, CV_LOAD_IMAGE_UNCHANGED);
if (img == 0)
{
  printf (˜Input file not open.\n˝);
  return 0;
}
```

The Fourier transform can be calculated only for a grey-level image, so the input might need to be converted from color to grey:

```
grayImg=cvCreateImage(cvSize(img->width,img->height),img->depth, 1);
grayImg->origin = img->origin;
if (img->nChannels > 1)
  cvCvtColor(img, grayImg, CV_BGR2GRAY);
```

```
else
  grayImg = img;
```

Now compute the two-dimensional Fourier transform of the grey image. There is no OpenCV function to do exactly this, so one has to be built. `fftImage` computes the Fourier transform of the image provided and returns a pointer to an image representing it. This image will contain pixels that are complex numbers, and so has two floating-point channels.

```
z = fftImage (grayImg);
```

The frequency domain image can be displayed, although it is not necessary. Because a Fourier transform image has complex pixels and a huge range of levels, a special display routine has been provided: `displayFDimage`.

```
displayFDimage (z, "Transform");
```

Now read the PSF image:

```
psf = cvLoadImage("h:\\aipcv\\ch6\\psf1.pgm", CV_LOAD_IMAGE_UNCHANGED);
```

To compute a convolution between the PSF and the input image, the two must be the same size. The `align` function copies the PSF image, centered, into an image of zero-valued complex pixels that has the specified size — in most cases, the size of the image to be convolved with.

```
psfa = align (psf, img->height, img->width);
```

Now transform and display the PSF:

```
zz = fftImage (psfa);
displayFDimage (zz, "FFT of PSF");
```

Now the convolution is computed in the frequency domain. Extract the corresponding complex valued pixels (that is, pixels having the same *i,j* coordinates) from the two images and multiply them together. Multiplying two complex numbers is the same as multiplying two binomials.

```
for (i=0; i<z->height; i++)
{
    for (j=0; j<z->width; j++)
    {
        s = cvGet2D (z, i, j);
        s1 = cvGet2D (zz, i, j);
        p = s.val[0]*s1.val[0] - s.val[1]*s1.val[1];
        q = s.val[0]*s1.val[1] + s.val[1]*s1.val[0];
        s.val[0] = p; s.val[1] = q;
        cvSet2D (z, i, j, s);
    }
}
```

The image z is now the Fourier transform of the blurred image, or the convolution of the two images we started with. Compute the inverse transform of this image using `fftImageInv`, and then display the result.

```
    zz = fftImageInv (z);
  displayFDimage (zz, "Result");
  return 0;
}
```

The modules used by the main program include three that are especially valuable and will be used in most other restoration programs that follow: `fftImage`, `align`, and `displayFDimage`. It is necessary to know what they do, but only the `fftImage` function needs to be looked at in detail, as it is the basis of all the restoration methods.

The function `fftImage` has to do a lot of things before `cvDFT` is called to do the actual work. First, the input image is assumed to be grey level, but will be one channel. A Fourier transform operates on complex data, so a complex copy of the input image needs to be built. The code creates a double precision complex (`IPL_DEPTH_64F`) image, `complexPart`, out of the real and imaginary parts of the input. In fact, the input is the real part, and the imaginary part is always zero.

```
IplImage *fftImage (IplImage *img)
{
    IplImage *realpart, *imgpart, *complexpart, *ret;
    CvMat *ft;
    int sizeM, sizeN;
    CvMat tmp;

    realpart = cvCreateImage( cvGetSize(img), IPL_DEPTH_64F, 1);
    imgpart = cvCreateImage( cvGetSize(img), IPL_DEPTH_64F, 1);
    complexpart = cvCreateImage( cvGetSize(img), IPL_DEPTH_64F, 2);
    cvScale(img, realpart, 1.0, 0.0); // copy grey input image to realpart
    cvZero(imgpart); // Set imaginary part to 0
    cvMerge(realpart, imgpart, NULL, NULL, complexpart);//real,imag=complex
```

A DFT algorithm cannot always be applied to an arbitrary set of data. Many algorithms require that the image have rows and columns that are a power of two in size. Some require a square image. OpenCV provides a function that returns the best size to use for an array of N points — `cvGetOptimalDFTSize`:

```
sizeM = cvGetOptimalDFTSize( img->height - 1 );
sizeN = cvGetOptimalDFTSize( img->width - 1 );
```

The DFT function needs a matrix, not an image. Create a matrix of the correct size:

```
ft = cvCreateMat( sizeM, sizeN, CV_64FC2 );
```

The default origin for a Fourier transform is the upper-left corner. For images, it is better to move the origin to the image center. This simply means changing the sign of some of the components, and is done by the `origin_center` function:

```
origin_center (complexpart);
```

Now copy the origin centered, complex image into the matrix by extracting a pointer to a width × height submatrix `tmp` into the matrix `ft` and copying the pixels from the image into that submatrix:

```
cvGetSubRect( ft, &tmp, cvRect(0,0, img->width, img->height));
cvCopy( complexpart, &tmp, NULL );
```

The matrix `ft` is the same size or bigger than the input data image. If it is bigger, the part of the image to the right of the data image (the rightmost columns extra to the input image) must be set to zero. Get a pointer to that subimage and clear those values in the matrix:

```
cvGetSubRect( ft, &tmp, cvRect(img->width,0, ft->cols - img->width,
   img->height));
if ((ft->cols - img->width) > 0) cvZero( &tmp );
```

Finally, the Fourier transform can be computed. There may be pixels between the `img->height` row of the matrix and the last row that have not been initialized. They could be set to zero as was done with the rightmost columns, but the fourth parameter of `cvDFT` is the number of rows to process. Any data beyond that point will be ignored.

```
cvDFT( ft, ft, CV_DXT_FORWARD, complexpart->height );
```

If needed, use `cvMatToImage` (not a standard OpenCV function, but part of the library provided with this book), to convert a matrix of numbers into an image:

```
ret = cvMatToImage (ft);
```

Don't forget to free the allocated storage. Quite a lot was allocated and should be returned if more processing is going to be done:

```
    cvReleaseMat( &ft );
    cvReleaseImage( &realpart );
    cvReleaseImage( &imgpart );
    cvReleaseImage( &complexpart );
    return ret;
  }
```

7.3 The Inverse Filter

It was a useful exercise to show in detail how an image can be blurred, since the reverse is about to be attempted. The blurring was accomplished by convolving the image with the PSF. Although no noise was added, in the real case it would be (Figure 7.1), giving the following process:

$$F = I \times H + \eta \qquad \text{(EQ 7.11)}$$

where F is the Fourier transform of the blurred image, I is the Fourier transform of the perfect image, and H is the Fourier transform of the PSF. The η term is the noise, which can be characterized statistically but never known perfectly. The multiplication in the frequency domain corresponds to a convolution of the two images i and h.

The goal of image restoration is to reproduce the original image i as well as possible, given f and h. Algebraically, it seems to make sense to divide by H and ignore η in equation 7.11, giving:

$$\frac{F}{H} = I \qquad \text{(EQ 7.12)}$$

The mathematics is really vastly more involved than this, but the end result is the same: This is the inverse filter, which is the least-squares restoration of f.

In detail: given an input image f and a PSF in the form of an image h:

1. Compute the 2D FFT of the image f; call this F.
2. Compute the 2D FFT of the PSF image h; call this H.
3. Compute the new image $G(i,j) = F(i,j)/H(i,j)$
4. Compute the inverse FFT of G, giving the restored image g.

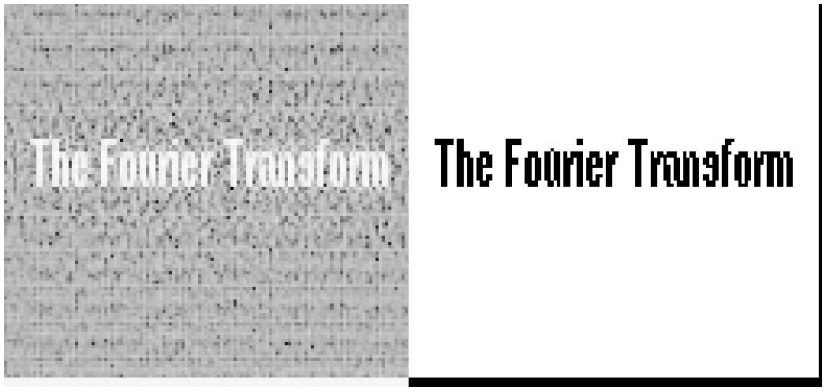
This can't be done naively or serious problems will result. The image H is certain to contain regions where the values are zero, and dividing by zero usually has disastrous results. In fact, even if H becomes too small, the result is that noise will dominate the restored image.

One solution is to check the norm of H at each pixel, and if it is below a specified threshold value, the division is not performed. Instead, the values from F can be left undisturbed, the pixel could be set to zero, or a default value can be used.

For example, Figure 7.9 shows the result of applying the inverse filter to the artificial image of Figure 7.8c. The filter demo program `blurandInverse.c` reads an image and a file containing a PSF and blurs the image, saving the result. Then it applies the inverse filter, displaying and saving the result of that, too.

The implementation of the inverse filter uses a new twist on the Fourier transform: the *origin-centered* Fourier transform. When the Fourier transform of

an image is calculated, it is done in such a way that the origin (and the brightest spot) is the pixel (0,0). This point is at the geometric center of an origin-centered transform, making the image symmetrical about its own center. It is a very easy thing to do: simply multiply every pixel (i, j) in the image by $(-1)^{i+j}$ before computing the FFT. This may be more a matter of personal preference than of necessity, but all the restoration modules use origin-centered transforms.



(a) (b)

Figure 7.9: Inverse filter restoration of Figure 7.8c. (a) The image as it appears after the restoration. (b) The same image after thresholding to remove the worst of the clutter. Note that the dot above the i is visible.

The inverse filter is very susceptible to noise, to the point where the inverse restoration of a very noisy image could be subjectively worse than the original. Adjustments to the threshold could be somewhat useful in these cases, and noise filtering might be useful, but a more sophisticated method is probably a better idea.

The inverse filter also depends on a good estimate of the PSF, of course, and estimating the PSF can be difficult. For example, blurring the image and saving it as a file can result in scaling effects being introduced into the image so that the same PSF that caused the blur does not effectively undo it in an inverse filter. This may be one reason why code commonly available on the Internet for demonstrating the inverse filter both blurs and unblurs the image without saving it, as was done in the `blurandInverse.c` program. Saving images as TIFF files is a good idea, because it can store pixels as real numbers. JPEG files actually introduce noise, through the built-in compression scheme.

7.4 The Wiener Filter

The Wiener filter [Helstrom, 1967] and its variants are designed to work in cases where the noise has become significant. This filter in its complete form requires that we know a good deal about the signal and the properties of the

noise that infects it. Without going into great detail, an approximation of this filter can be expressed as

$$I \approx \left[\frac{1}{H} \frac{\|H\|^2}{\|H\|^2 + K} \right] F \qquad \text{(EQ 7.13)}$$

where $\|H\|^2$ is the norm of the complex PSF image H, and K is a constant. One suggestion [Gonzalez, 1992] for a value of K is $2\sigma^2$, where σ^2 is the variance of the noise.

Figure 7.10 shows the Wiener filter restoration of the test image of Figure 7.8c. Because this image has no noise associated with it, any advantage of the Wiener filter is not clearly shown; indeed, in the absence of noise the Wiener filter is reduced to the inverse filter. Thus, the face image first seen in Chapter 4 (Figure 4.4) was blurred and then restored using both the inverse and Wiener filters. These results also appear in Figure 7.10. The image restored using the inverse filter shows a grid pattern, which is absent in the Wiener restored image.

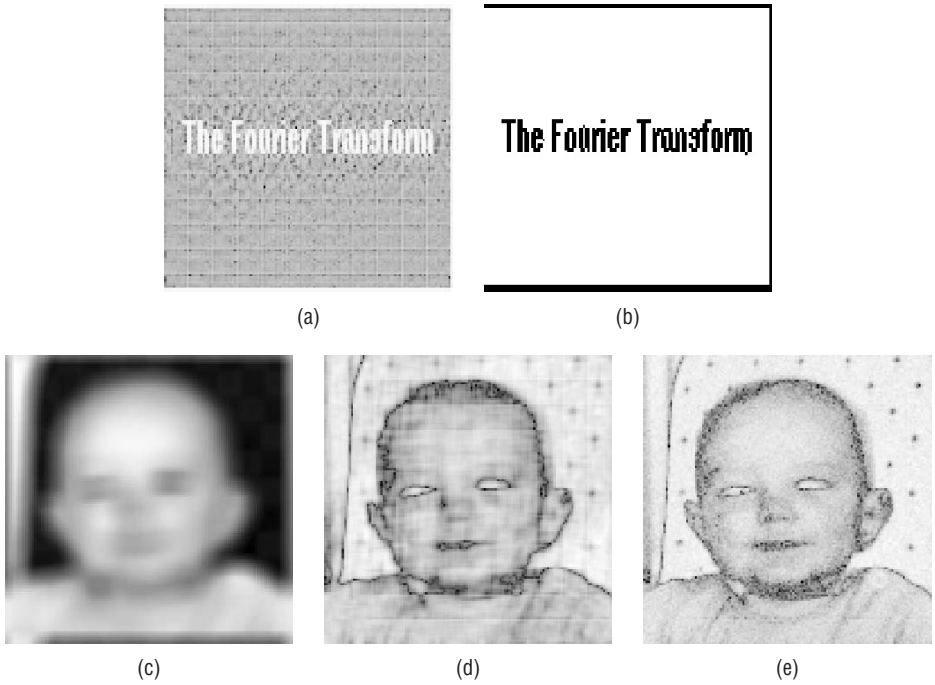


Figure 7.10: Wiener file restoration. (a) Restored test image. (b) The same image thresholded. (c) Blurred version of the face image. (d) Inverse filter restoration. (e) Wiener filter restoration. Note that the pattern in the background can be seen clearly.

7.5 Structured Noise

In some cases the distortion imposed on an image has a regular pattern associated with it. For example, when an electric motor is operated close to a television or other video device, it is common to see a pattern of lines on the screen. This is caused by the fact that the motor generates a signal that interferes with the television, and the interference has a frequency associated with it that is related to the speed of the motor.

There is a wide variety of causes for structured noise, and it can be quite a problem. However, because the noise is periodic, the Fourier transform can be used to determine where the peak frequencies are. The noise will correspond to one of these, and it can be virtually eliminated by clearing those regions in the frequency domain image that correspond to the noise frequencies, and then back-transforming it into a regular image.

As a simple example, Figure 7.11a shows the face image with high-frequency sinusoidal interference imposed on it. The Fourier transform of this image (Figure 7.11b) shows a number of bright spots (*spikes*). A matching pair of spikes appears in the upper-left and lower-right quarters of the image, and these correspond to the periodic signal causing the pattern of diagonal lines. To correct this, edit the Fourier domain image and set the two spikes to zero; then apply an inverse FFT to obtain the space domain image. The result, after some contrast improvement, appears in Figure 7.11d.

There are a number of questions still to be addressed. First among these concerns exactly which spikes to remove, and unfortunately there is no good answer. Experience with the appearance of Fourier domain images will help, and for this purpose the `fftlib.c` procedures will be very useful. In particular, it will be quickly learned that the peak in the center of the image contains much of the interesting information in the image, and must not be removed. Periodic signals cause symmetrical spikes on each side of the central peak, and though there can be many of these, they can be removed in pairs to see what happens. When the correct spikes are found, the image will be improved by their removal. Depending on the type of noise there could be more than one pair of spikes to be removed.

The restoration in this figure was performed by the program `snr.c` (structured noise removal). This program computes the Fourier transform of the input image, and then displays it on the screen. The user sets portions of the frequency domain image to zero (black) by clicking the mouse on the upper-left corner of a rectangular area to be darkened, dragging the mouse to the lower-right of the region, and then releasing mouse button. To undo, type a carriage return so that the selected portion will go black on the screen, and

the process can be repeated. When all regions are set to black, type **x** and the program will back transform the modified spectrum and display the result.

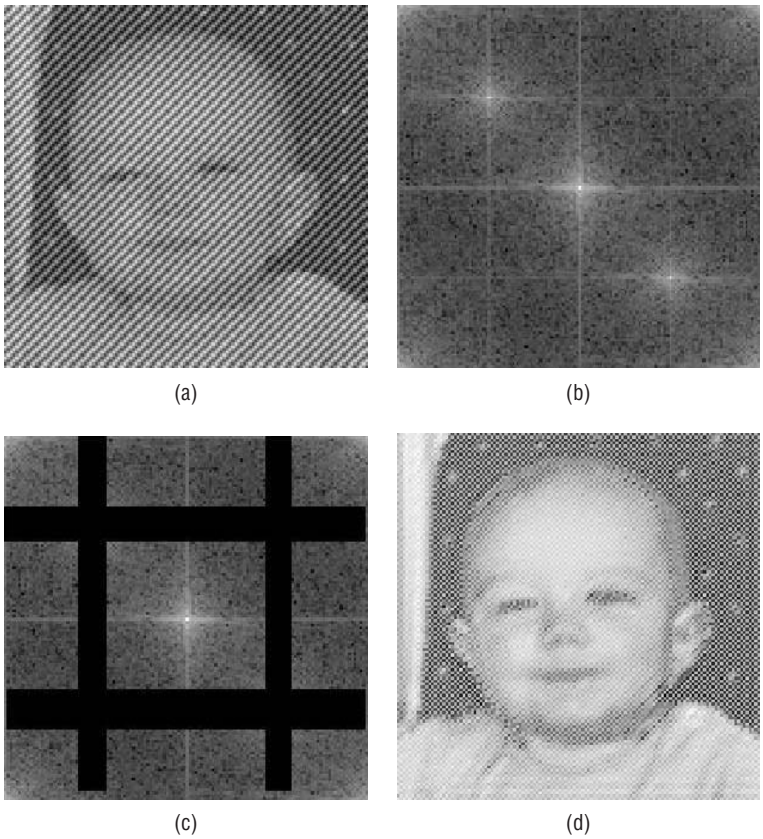


Figure 7.11: Example of structured noise removal. (a) Face image with an imposed sinusoidal pattern. (b) Fourier transform of the noisy image, showing the two spikes responsible for the pattern. (c) The regions of the fourier image to be removed. (d) Restored image; the spikes were set to zero and then the inverse Fourier transform was computed.

This results in the pixel at (32,32) and the one at (96,96) in the Fourier transform of the corrupt face image being set to zero.

Just as one person's weed is another's flower, what constitutes noise is in the eye of the beholder. One fairly common issue in document analysis is the existence of grid lines in the data. As a specific example, many types of pen recorder plot lines in ink on a sheet of graph paper. The grid lines interfere with the extraction of the plotted data lines, making them difficult to extract properly.

In this instance, knowledge of the behavior of the Fourier transform helps a lot. The transform of a horizontal line appears like a vertical line in the

frequency domain image, and a vertical line transform as a horizontal one. Therefore, it seems like a variation of the `snr.c` procedure would have a chance of removing the grid lines. After creating the Fourier transform, remove those pixels in the center rows and columns of the frequency domain image, taking care not to remove pixels too close to the center peak. This is called a *notch filter*.

Figure 7.12a shows an image that was obtained by scanning an original paper document. It is simply a handwritten note on graph paper. The goal is to remove the grid lines as completely as possible. The Fourier transform of the image (Figure 7.12b) is obtained, and the pixels within the specified notch regions are set to zero (Figure 7.12c) before performing the inverse Fourier transform. The result (Figure 7.12d) shows an astonishing lack of grid lines, although it does display some artifacts of the process (ringing).

Using this method, it would be possible to remove patterns in either direction — for example, lines on notepaper.

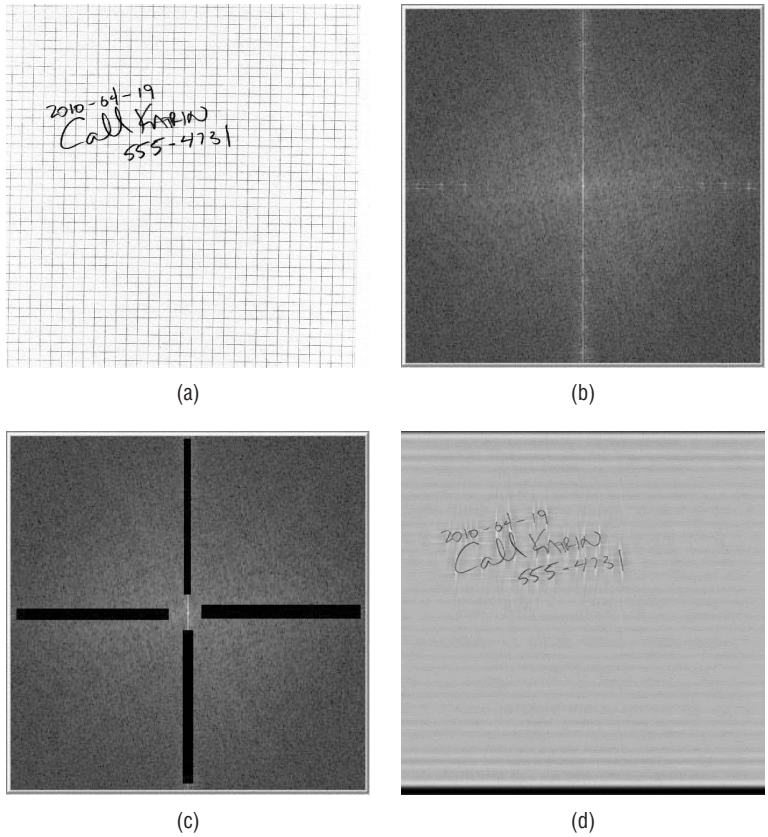


Figure 7.12: Removal of grid lines using a notch filter. (a) Original scanned image. (b) Fourier transform of scanned image. (c) Notches along the center lines, removing both the vertical and horizontal lines. (d) The restored image.

7.6 Motion Blur—A Special Case

If an image has been blurred due to the motion of either the camera or the object, the PSF will be an extended blob with the long axis indicating the direction of motion. While it is possible to use an inverse or Wiener filter restoration in these cases, there is a special solution that should not be as susceptible to noise.

If the motion can be assumed to be uniform and in the x (horizontal) direction, a very nice expression [Gonzalez, 1992; Sondhi, 1972] can be used to remove most of the blur without resorting to a Fourier transform: =0pt

$$I(y,x) \approx \bar{f} - \frac{1}{K}\sum_{k=0}^{K-1}\sum_{j=0}^{k} f\prime(y, x - ma + (k-j)a) + \sum_{j=0}^{m} f\prime(y, x - ja) \qquad \text{(EQ 7.14)}$$

where

- $\bar{f}$ is the mean value of the blurred image f.
- a is the distance of the blur.
- K is the number of times that the distance a occurs in the image; that is, the number of columns $= Ka$. K is approximated as an integer, where necessary.
- m is the integer part of x/a, for any specified horizontal position x.
- f' (i,j) is the derivative of f at the point (i, j). This can be approximated by a difference, as was done in Chapter 2.

As an example, Figure 7.13 shows this method applied to a blurred version of the sky image (Figure 4.2). This image is blurred by about 20 pixels in the horizontal direction. The restored version suffers from some artifacts but is a noticeable improvement over the blurred image. The program `motion.c` performed the restoration. When called, it requests the name of the input file (to be restored) and a *speed* value. The speed value reflects the number of pixels that pass a point during the time the image was acquired, and is mostly a guess. The program restores the image for a range of speed values centered at the one specified: if the value of 20 is entered, then the image is restored for values 10–30 in increments of 1. After each restoration, the image is displayed, allowing the user to select the best one.

The speed value can be estimated from the image, or arrived at by trial and error. To estimate the amount of blur, look for an edge that is as close to perpendicular to the direction of motion as possible. If the amount of motion is significant, it should be possible to determine the original and final position of the edge; that is, the location of the edge when the motion started (or the

shutter opened) and its position when it stopped (the shutter closed). This is illustrated in Figure 7.13d. From the expanded view of the edge, we can see that motion appears to encompass about 19 pixels. The result of a motion correction of 19 pixels (Figure 7.13e), 20 pixels (7.13c), and 21 pixels (7.13f) shows that the best result was achieved with a value of 20.

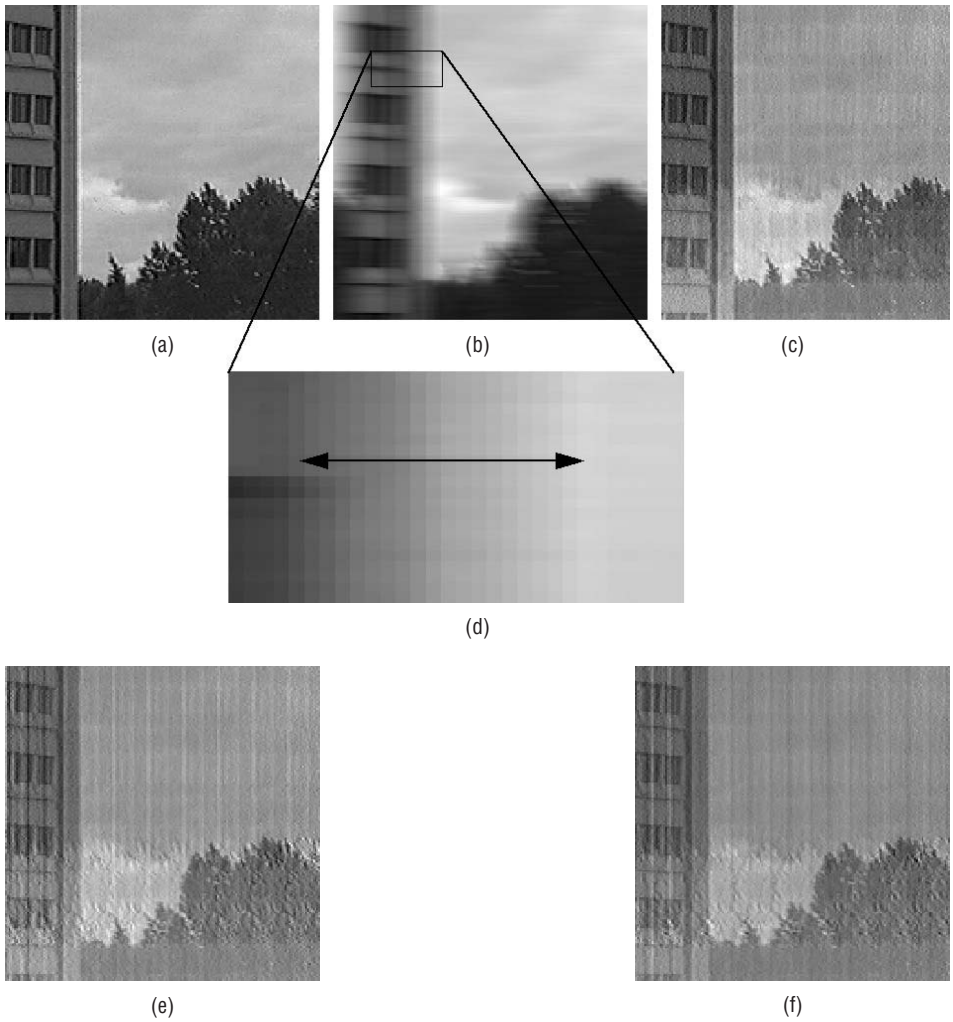


Figure 7.13: Motion blur removal. (a) The original sky image. (b) The same image blurred by about 20 pixels horizontally. (c) Restored, using the motion program. (d) Estimation of the amount of motion by examining the vertical edge. The approximate start and end of the edge can be seen in the blurred image. (e) The blurred image restored, assuming a motion of 19 pixels. (f) Result of the restoration assuming 21 pixels.

7.7 The Homomorphic Filter—Illumination

Homomorphic filtering is a technique in which an image is transformed to a new space or coordinate system in which the desired operation is simpler

to perform. Specifically, the problem to be addressed is one of improving the quality of an image that has been acquired under conditions of poor illumination. As mentioned in Chapter 4, illumination can have a very important influence on the appearance of the image, and on what it can be used for. The ideal situation would be to generate the original set of objects in an image without regard to the impinging illumination. This can't be done exactly or in all cases, but some steps can be made to improve the situation.

An image is an array of measured light intensities, and is a function of the amount of light reflected off the objects in the scene. The intensity is the product of the reflectance of the object R and the intensity of the illumination I:

$$f(i,j) = I(i,j)R(i,j) \quad \text{(EQ 7.15)}$$

The methods used so far for restoration involve using the Fourier transform, which will not separate I from R; it would if they were summed, of course. Fortunately, a sum can be created: Simply take the log of both sides:

$$\log f(i,j) = \log I(i,j) + \log R(i,j) \quad \text{(EQ 7.16)}$$

Now all that is needed is a way to separate these two components. A simple observation will help: Illumination tends to vary slowly, or relatively so, across an image. The reflectance, on the other hand, is characterized by sharp changes, especially at boundaries (edges). This means that if the low-frequency components (illumination) could be decreased, while increasing the high-frequency (reflectance) components, the problem would be solved. The Fourier transform is exactly what is needed to do this.

7.7.1 Frequency Filters in General

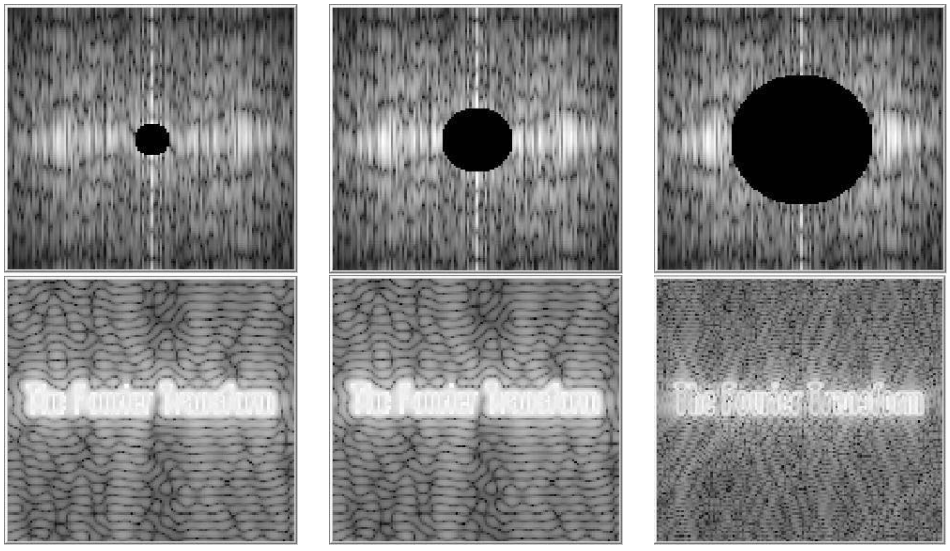
The use of the term ''frequency'' to describe parts of an image seems odd, but the use of a Fourier transform means that the image must be viewed as a signal, and so must possess something that corresponds to frequencies. A *spatial frequency* is a measure of how a structure in an image repeats over a distance. Sound can be thought of as the sum of various audio components, each of which has a specific frequency; the Fourier transform determines how much of each frequency (e.g., sine waves) comprises the sound. An image can be thought of as the sum of spatially varying grey or color components, each having a specific frequency. It is two dimensional, and the components sum across the image, reinforcing brightness in some paces and cancelling out in others, creating the light and dark regions that are seen.

High-frequency spatial components are small and correspond to edges, pixels, and small regions. Low-frequency spatial components give overall structure, consisting of objects and background features. Thus, using the Fourier transform, it should be possible to filter out or enhance various

spatial frequencies, should this be useful. In the Fourier transform of an origin-centered image, the low-frequency information corresponds to the region near the center of the image, and the frequency increases with distance from the center. So, to remove low-frequency information, delete data near the center of the Fourier-transformed image before back transforming.

The use of the Fourier transform as a filter to pass or block certain *spatial frequencies* will be illustrated by example. Consider the example image first seen in Figure 7.8a, consisting of the words *The Fourier Transform*. The origin-centered Fourier transform of this image will have a peak at the center; let's see what happens when pixels near the center of the transform are set to zero.

Figure 7.14 a–c show the results of clearing pixels near the center peak of the transform before back-transforming; pixels within a radius of 8, 16, and 32 pixels of the center (respectively) were affected. The effect on the image is curious: The more of the center region that is cleared, the more the image seems to consist of isolated lines and spots. The reason is that the outlying regions of the Fourier transformed image corresponds to high-frequency information, and removing high-frequency information affects details. On the other hand, passing only the central region and setting to zero the remainder of the Fourier transform allows the low-frequency information to be retained, while removing the high frequencies. This is seen in Figure 7.15. Only the basic outline or position of the objects remain, and details are progressively lost. This is called a *low-pass* filter, with the former filter being a *high-pass* filter.



(a) (b) (c)

Figure 7.14: The high-pass filter. (a) Pixels within a radius of 8 of the center of the Fourier transform were set to zero before back transforming. (b) Radius = 16. (c) Radius = 32.

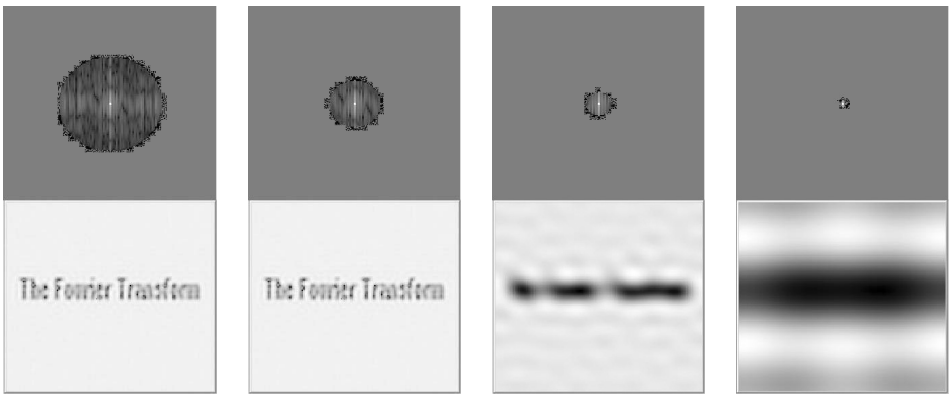


Figure 7.15: The low-pass filter. (a) Pixels outside of a circle of radius 32 of the center of the Fourier transform were set to zero, allowing only the center pixels to be used in the inverse transform. (b) Radius = 16. (c) Radius = 8. (d) Radius = 2.

That a particular spatial frequency will correspond to points in the Fourier transform that are a fixed distance from the center is true for origin-centered transforms only. A *frequency domain filter* is an image having values that correspond to the desired frequencies in the output. For example, the frequency domain filter that was used to give Figure 7.14c has zeros in a disk of radius 32 at the center of the image, and ones everywhere else. Values less than one will suppress the corresponding frequency in the output, and values greater than one will enhance that frequency.

A band, containing any particular set of spatial frequencies, can be either allowed to remain (*band-pass* filter) or be blocked (*band-stop* filter) to varying degrees. This would correspond to a ring of one or zero pixels in the frequency domain filter. In addition, a set of frequencies can be enhanced by increasing their relative values in the Fourier transform image, instead of simply passing or blocking them. This can be done at the same time as other frequencies are reduced or blocked altogether.

This happens to be what we want to do for the homomorphic filter. The high frequencies should be emphasized, so they will be increased. The low frequencies correspond to illumination, and so will be decreased. This will be called a *high-emphasis* filter, and its shape is shown in Figure 7.16a. Other such filters could be used, and the shape of the filter can be changed to meet specific needs [Stearns, 1988].

7.7.2 Isolating Illumination Effects

Now the homomorphic filter can be completed. The stages in processing are as follows:

1. Take the log of all pixels in the image.
2. Compute the Fourier transform of the image obtained in 1.

3. Apply the high-emphasis filter by multiplying the elements in the filter mask by those in the Fourier transform image.
4. Compute the inverse Fourier transform.
5. Compute the exponential of all pixels; this reverses the logarithm of step 1. Stretch the contrast, if needed.

Figure 7.16 shows an application of the homomorphic filter to the face image having an imposed sinusoidal illumination gradient (originally seen in Figure 4.4). The result is much clearer in the formerly dark areas, and the overall contrast is better. The suppression of the bands is not complete, but appears to be sufficient.

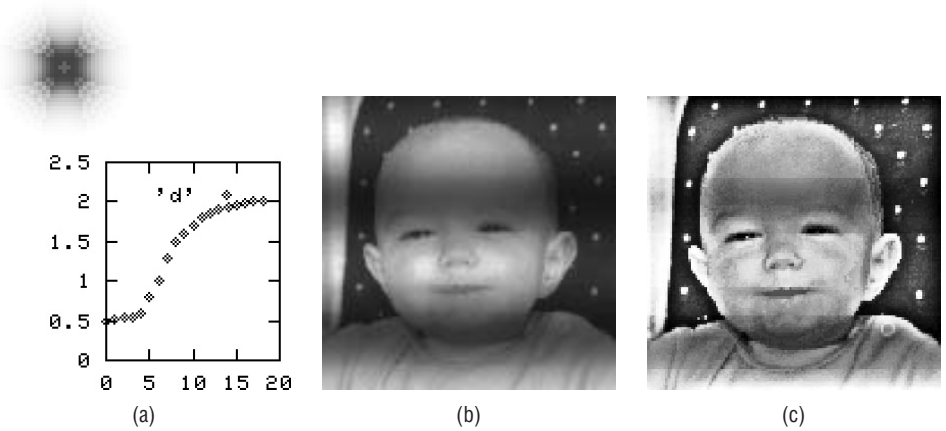


Figure 7.16: The homomorphic filter. (a) The shape of the high-emphasis filter, which is multiplied element-by-element with the Fourier transform of the log or the original image. The graph shows a cross-section through the center of the filter. (b) Test image: this is the face image with sinusoidal illumination. (c) Result of homomorphic filtering. Detail is much clearer, particularly in the areas that were dark.

7.8 Website Files

`fftlib.c`	A library for computing Fourier transforms; needs `fftlib.h`
`fftlib.h`	`Include` file for `fftlib.c`
`highemphasis.exe`	High-emphasis filter
`highpass.exe`	High-pass filter
`lowpass.exe`	Low-pass filter
`hiemphasis.c`	High-emphasis filter
`homomorphic.exe`	Homomorphic filter
`motion.exe`	Motion blur removal

snr.exe	Structured noise removal
blur.c	Blur an image using a PSF
blurAndInverse.c	Inverse filter demo
fft1d.c	One-dimensional Fourier transform, OpenCV
hiemphasis.c	High-emphasis filter
highpass.c	High-pass filter
homomorphic.c	Homomorphic filter
lowpass.c	Low-pass filter
motion.c	Motion blur removal
slow1.C	Version of the Fourier transform
slow2.C	Version of the Fourier transform
slow3.C	Version of the Fourier transform
slow4.C	Figure 7.6
snr.c	Structured noise removal
blur2010.pgm	Blurred FACE image
face.pgm	FACE image
facepsf.pgm	Point spread function for FACE blur
faces.pgm	FACE with sine illumination
four.pgm	Image of the word FOURIER
fourbl.pgm	Blurred FOURIER image
fsn.pgm	Structured noise image of FACE
grid2010.pgm	Figure 7.12a
psf1.pgm	Point spread function
psf1a.pgm	Point spread function
psf2.pgm	Point spread function
psf2a.pgm	Point spread function
skym.pgm	Motion-blurred image

7.9 References

Andrews, H. C. and B. R. Hunt. *Digital Image Restoration*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

Andrews, H. C. ''Digital Image Restoration: A Survey.'' *Computer* 7, no. 5 (1974): 36–45.

Bracewell, R. *The Fourier Transform and Its Applications*. New York: McGraw-Hill, 1965.

Brigham, E. O. *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

Cooley, J. W. and J. W. Tukey. ''An Algorithm for the Machine Calculation of Complex Fourier Series.'' *Mathematical Computation* 19 (1984): 297–301.

Geman, S. and D. Geman. ''Stochastic Relaxation, Gibbs Distributions, and Bayesian Restoration of Images.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6, no. 6 (1984): 721–741.

Gonzalez, R. C., and R. E. Woods. *Digital Image Processing*. Reading, MA: Adison-Wesley, 1992.

Gull, S. F. and G. J. Daniell. ''Image Reconstruction from Incomplete and Noisy Data.'' *Nature* 292 (1978): 686–690.

Hall, E. L. *Computer Image Processing and Recognition*. New York: Academic Press, 1979.

Helstrom, C. W. ''Image Restoration by the Method of Least Squares.'' *Journal of the Optical Society of America* 57, no. 3 (1967): 297–303.

Hunt, B. R. ''The Application of Constrained Least Squares Estimation to Image Restoration by Digital Computer.'' *IEEE Transactions on Computers* C-22, no. 9 (1973): 805–812.

Lee, H. C. ''Review of Image-Blur Models in a Photographic System Using the Principles of Optics.'' *Optical Engineering* 29 (1990).

Macnaghten, A. M. and C. A. R. Hoare. ''Fast Fourier Transform Free from Tears.'' *The Computer Journal* 20, no. 1 (1975): 78–83.

MacAdam, D. P. ''Digital Image Restoration by Constrained Deconvolution.'' *Journal of the Optical Society of America* 20, no. 12 (1970): 1617–1627.

Pavlidis, T. *Algorithms for Graphics and Image Processing*. Rockville, MD: Computer Science Press, 1982.

Pratt, W. K. and F. Davarian. ''Fast Computational Techniques for Pseudoinverse and Wiener Restoration.'' *IEEE Transactions on Computers* C-26, no. 6 (1977): 636–641.

Pratt, W. K. ''Generalized Wiener Filter Computation Techniques.'' *IEEE Transactions on Computers* C-21, no. 7 (1972): 636–641.

Sezan, M. I. and A. M. Tekalp. ''Survey of Recent Developments in Digital Image Restoration.'' *Optical Engineering* 29, no. 5 (1990): 393–404.

Singleton, R. C. ''On Computing the Fast Fourier Transform.'' *Communications of the ACM* 10, no. 10 (1967): 647–654.

Slepian, D. ''Restoration of Photographs Blurred by Image Motion.'' *Bell System Technical Journal* (1967): 2353–2362.

Stearns, S. D. and R. A. David. *Signal Processing Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Stockham, T. G. ''Image Processing in the Context of a Visual Model.'' *Proceedings of the IEEE* 60 (1972): 828–841.

Van Cittert, P. H., ''Zum Einfluß der Spaltbreite auf die Intensitätsverteilung in Spektrallinien II.'' *Zeitschrift für Physik* 69 (1931): 298–308.

CHAPTER

8

Classification

8.1 Objects, Patterns, and Statistics

Until now, the discussion has surrounded images and the operations that can be performed on them to enhance or otherwise modify them. The purpose of the modification is not really relevant in general; the result must simply meet some user-defined criteria of ''goodness.'' All this falls into the realm of *image processing* and is very common these days, especially when images are to be used by humans. Examples can be found in many forensic television programs such as *NCIS* and in motion pictures. A famous scene in *Blade Runner* depicts an investigator magnifying an image so that a reflection in a human eye yields digits that can be read. This is impossible with imaging technology available now, but it is an example of image processing. Another example is the processing of Hubble Space Telescope data that gives such wonderful color images.

Computer vision is more than that. It is the analysis of digital images so as to extract information automatically. The extracted information may be trivially simple, such as an answer to the question ''What color is this?'' or it may be much more complex, such as ''Whose face is this?''

Computer vision depends on having a good image with at least some known properties. Image processing is often used to enhance an image for further processing by vision algorithms, and sometimes there are known parameters of the camera and capture system that are used to permit more vision processing. Knowing the nature of the sensor and the lens, for example, can help to

determine absolute distances within images. However, at its heart, computer vision is about making measurements on images and/or determining what objects appear within those images.

Many people have difficulty understanding why this is a hard problem. After all, people recognize complex objects with apparent ease, and quickly. Why is this hard for computers? The answer is that computers use pixels to represent objects rather than some more natural representation that has more structure. Raster images are fundamentally two-dimensional and discrete, and are a poor way to represent an object. Figure 8.1 is an attempt to illustrate this.

255	255	255	26	26	26	21	21	21	4	4	4	5	5	5	5	5	5	3	3	3	7	7	7	16	16	16	14
14	14	5	5	5	12	12	12	11	11	11	7	7	7	8	8	8	9	9	9	23	23	23	10	10	10	29	29
29	16	16	16	11	11	11	8	8	8	15	15	15	9	9	9	11	11	11	17	17	17	53	53	53	228	228	228
255	255	255	19	19	19	10	10	10	10	10	10	4	4	4	4	4	4	12	12	12	10	10	10	11	11	11	11
11	11	18	18	18	15	15	15	12	12	12	14	14	14	14	14	14	21	21	21	27	27	27	64	64	64	165	165
165	135	135	135	56	56	56	15	15	15	15	15	15	29	29	29	20	20	20	21	21	21	42	42	42	137	137	137
255	255	255	227	227	227	1	1	1	5	5	5	4	4	4	4	4	4	11	11	11	6	6	6	14	14	14	29
29	29	110	110	110	65	65	65	188	188	188	166	166	166	74	74	74	112	112	112	221	221	221	205	205	205	238	238
238	236	236	236	213	213	213	68	68	68	18	18	18	19	19	19	14	14	14	15	15	15	8	8	8	19	19	19
255	255	255	0	0	0	6	6	6	8	8	8	2	2	2	9	9	9	9	9	9	124	124	124	207	207	207	217
217	217	232	232	232	228	228	228	234	234	234	243	243	243	246	246	246	236	236	236	243	243	243	243	243	243	241	241
241	244	244	244	243	243	243	234	234	234	39	39	39	15	15	15	11	11	11	16	16	16	9	9	9	10	10	10
254	254	254	18	18	18	5	5	5	3	3	3	0	0	0	78	78	78	187	187	187	212	212	212	218	218	218	231
231	231	231	231	231	237	237	237	235	235	235	236	236	236	241	241	241	236	236	236	241	241	241	244	244	244	240	240
240	238	238	238	239	239	239	237	237	237	188	188	188	36	36	36	26	26	26	14	14	14	9	9	9	17	17	17
254	254	254	0	0	0	4	4	4	8	8	8	24	24	24	110	110	110	186	186	186	207	207	207	219	219	219	221
221	221	229	229	229	232	232	232	233	233	233	233	233	233	231	231	231	238	238	238	239	239	239	235	235	235	240	240
240	236	236	236	231	231	231	232	232	232	186	186	186	33	33	33	33	33	33	2	2	2	31	31	31	5	5	5
22	22	22	7	7	7	5	5	5	4	4	4	42	42	42	71	71	71	184	184	184	211	211	211	211	211	211	225
225	225	232	232	232	232	232	232	235	235	235	236	236	236	236	236	236	239	239	239	241	241	241	241	241	241	234	234
234	233	233	233	227	227	227	226	226	226	220	220	220	25	25	25	17	17	17	16	16	16	20	20	20	10	10	10
0	0	0	4	4	4	9	9	9	9	9	9	39	39	39	78	78	78	139	139	139	202	202	202	217	217	217	215
215	215	232	232	232	234	234	234	228	228	228	233	233	233	232	232	232	238	238	238	241	241	241	243	243	243	234	234
234	234	234	234	227	227	227	223	223	223	105	105	105	51	51	51	25	25	25	18	18	18	16	16	16	20	20	20
37	37	37	13	13	13	4	4	4	17	17	17	49	49	49	87	87	87	131	131	131	208	208	208	222	222	222	222
222	222	226	226	226	231	231	231	232	232	232	238	238	238	234	234	234	238	238	238	240	240	240	240	240	240	232	232
232	227	227	227	226	226	226	204	204	204	200	200	200	22	22	22	19	19	19	16	16	16	17	17	17	11	11	11
246	246	246	15	15	15	9	9	9	0	0	0	38	38	38	104	104	104	153	153	153	225	225	225	238	238	238	233
233	233	233	233	233	233	233	233	233	233	233	231	231	231	236	236	236	245	245	245	242	242	242	237	237	237	235	235
235	227	227	227	217	217	217	219	219	219	171	171	171	14	14	14	17	17	17	5	5	5	31	31	31	18	18	18
0	0	0	8	8	8	5	5	5	12	12	12	50	50	50	97	97	97	174	174	174	154	154	154	225	225	225	235
235	235	236	236	236	233	233	233	213	213	213	220	220	220	229	229	229	233	233	233	232	232	232	239	239	239	235	235

Figure 8.1: The pixels in an image. What does the image represent?

The figure shows an array of numbers that represent an image. These are grey-level pixel values, and are really how the data is presented to the

computer and the vision algorithms. What is this image? What objects appear within it? From looking at the numbers, it is very hard to say. Figure 8.2 shows the image in a form that we can more easily process (greys) and it is plain from this that we have a picture of a face — Albert Einstein, in fact.

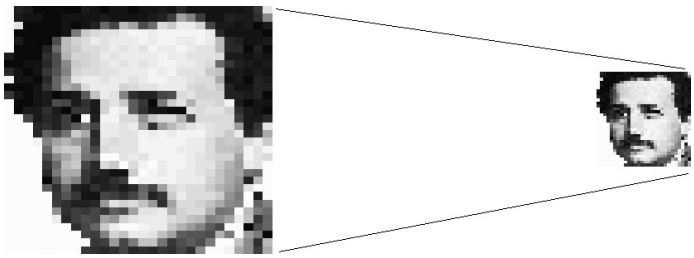


Figure 8.2: Rendering the data in Figure 8.1 as a grey-level image, we see that the 28x32 array of pixels is an image of Albert Einstein's face.

It is generally easier to decompose or parse a complex object into components than it is to synthesize low-level components into high-level complex objects or ideas. So, drawing objects into an image (i.e., rendering) is easier than the reverse, collecting pixels into related structures that represent objects. Computer vision is all about connecting the dots, literally: collecting pixels into logical structures that represent objects or portions of objects that can, in turn, be connected.

Some basic definitions are needed in order to clarify how computer vision will be carried out. Let's define an *object* as something, a two- or three-dimensional thing, that can appear in an image. *Object recognition* is the act of finding a collection of pixels in an image that represents an object being searched for, and the assigning of a label to the collection. The label is the name of the object. So, as a simple case, when a car is recognized in an image, an object recognition system will identify those pixels that are a part of the car and give it the label ''car.''

A *pattern* can be thought of as a combination of qualities (data) that form a characteristic arrangement. Patterns can occur in numbers, in pixels, sounds, or even behavior patterns. Patterns are important in computer vision because certain patterns of pixels represent specific objects in images. If those patterns can be detected, then it is an indication of the occurrence of that object in the picture. Sadly, characteristic patterns tend to be obscured by noise, scale, and orientation issues, lighting, and other practical matters. The usual method used within functioning vision systems is to collect simple patterns into low-level objects, which in turn are grouped into higher levels until the object can be built from the pieces. For instance, when trying to see faces in images, perhaps it is simpler to look for eyes or noses and try to build faces from these parts.

Because of the nature of the data (noisy and variable) and the difficulties in describing objects, vision algorithms are not perfect. Unlike, for example, a sorting algorithm, which must always yield a sorted collection of numbers upon demand, an object recognition algorithm usually works only sometimes. If a face recognizer can find 95% of the faces in an image, then 95% is the *success rate*, and it fails 5% of the time. Many things can cause failures, including shadows, motion blur, occlusion, noise, and scale problems. The developers of these systems generally know the success rates and often know the causes of failure, too. Thus, vision algorithms tend to be characterized by statistics, and frequently use statistics as an essential aspect of their function. Indeed, one of the most important methods of object recognition uses *statistical pattern recognition*, in which objects are characterized statistically using a set of measurements.

Now that some basic definitions have been presented, it is time for an essential aspect of computer vision to be made clear: *Vision systems are always looking for a specific set of objects*. This makes things a lot easier for the designer of the system. A vision system for counting cars on a freeway, for example, needs to be able to recognize vehicles, and perhaps people, roads, and static objects in the field of view. It does not have to recognize fish or camels, balloons, or chairs. If any of those objects appears in an image, the system will not identify them; indeed, these objects may interfere with the recognition of intended objects. The set of objects to be recognized is sometimes called the *domain* of the system or algorithm, and the behavior is to recognize an object or claim that none of the target objects appear within the image. Behavior outside of the domain, in other words, is not clearly defined.

8.1.1 Features and Regions

A crude but functional definition of a feature is *something that can be measured in an image*. A feature is therefore a number or a set of numbers derived from a digital image. The idea is that some objects belong to groups based on each of these measurements. Color is a simple feature, for example. It is a measurement (a determination of hue) and can, in fact, sometimes be used to recognize objects. For example, Figure 8.3 shows an image of a sample of carrots and tomatoes, carefully arranged to be isolated, but showing illumination effects and color differences. Vision problem: find the tomatoes.

Using color as a feature, we note that tomatoes are redder than carrots. Objects having a lot of red compared with other objects are more likely to be tomatoes than carrots (or peas). So, as a solution to the problem, it is possible to find isolated objects in the image, perhaps using an edge detector, and then count the red pixels within each object. In this example, it will work. The tomatoes can be isolated by thresholding hue at a value of 15/255. The result, as shown simply in Figure 8.4, is that all tomatoes can be found.

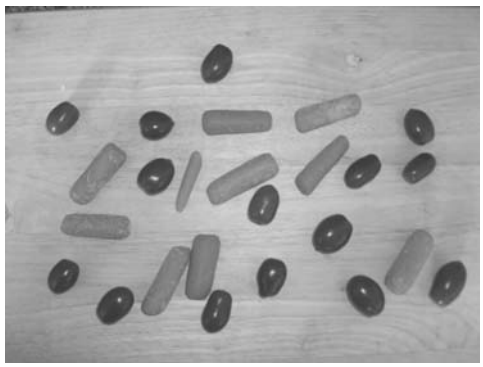


Figure 8.3: An image of tomatoes and carrots. Which are the tomatoes?

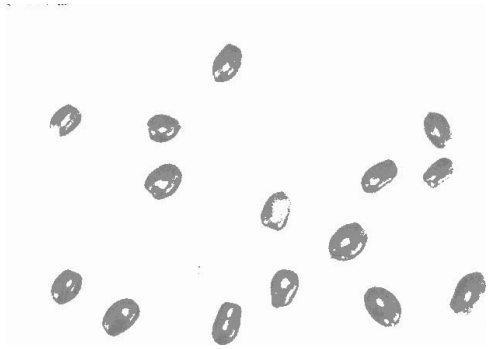


Figure 8.4: The tomatoes found by using hue as a feature.

Technically, this is still image processing and enhancement. A vision task would be to count the tomatoes, which can be done by counting the number of regions in the image of Figure 8.4. It does seem to show that hue can be used to solve the problem, though, and this is usually a part of the initial evaluation of a proposed solution.

Features are associated with image regions. An object within an image has a set of measurements (features) that can be used to characterize it. The initial task in object recognition vision problems is to isolate part of the image that might contain an object, measure its features, and determine which ones are useful in characterizing it.

The way that vision problems are approached usually has the following general form. A *target* is an object of the class being searched for; an example of what is to be recognized.

1. Find a way to isolate objects in the image that might be targets. This is often accomplished through edge detection, thresholding, region growing, or some other segmentation method.

2. Segmentation will create a simpler image in which targets are intermixed with other objects. Looking at the image, pick a feature that seems likely to characterize the targets.
3. Measure all potential targets using the proposed feature. Question: Is there a range of measured values that contain all the targets and no other objects? If so, this feature is the one to use.
4. If not all the targets can be found using this feature, record the percentage that can. Pick another feature and repeat from step 3.

It is possible to use more than one feature, so even if no one of them is perfect, we'll see that a set of merely acceptable ones can be just as good.

Let's use this scheme on the carrots/tomatoes problem. Isolation of the objects could be possible using edge detection or thresholding. Figure 5.1a shows a thresholded image, using Otsu's method (as in `thr_glh.c`). A surprising number of black regions are artifacts of the process, being neither carrots nor tomatoes. Most of these are tiny; some are too large. Area can be used to eliminate most of these, it would seem. Determining area requires a bit more processing.

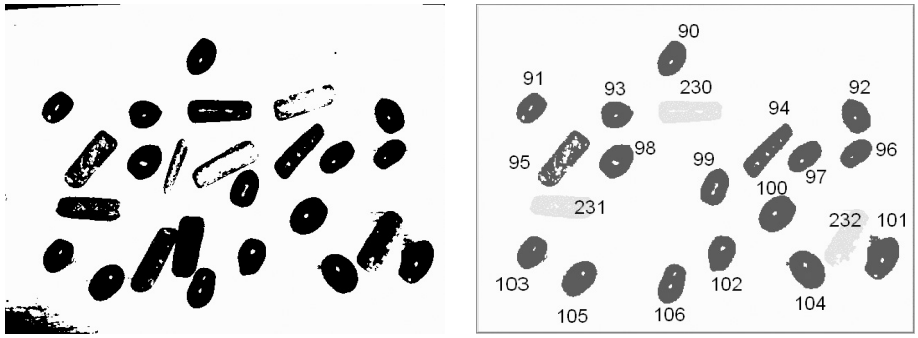


Figure 8.5: Marking potential targets in an image. (Left) A thresholded version of Figure 8.3. (Right) The regions that remain after selection using area. Each region is marked with a distinct grey level.

Each of the black areas in Figure 8.5a is separated from the others by regions of white pixels (background). A simple algorithm called a *flood fill* can identify isolated sets of connected black pixels; each of these could be a target (tomato). A basic recursive flood fill starts with a black pixel, and it recursively sets its black neighbors to another value (a *mark* value), and their black neighbors, and so on. In the present case, the target value (indicating an object, and to be replaced by a mark) is 0, and the mark value is anything other than 0 (black, object) and 255 (white, background). The result of a flood fill is that a region of

pixels that are connected to each other (a *connected region*) is given a particular grey level. The code for this is:

```
void flood (IMAGE img, int i, int j, int target, int replace)
{
       if (img->data[i][j] == target)
       {
             img->data[i][j] = 1;
             img->data[i][j] = replace;

             if (i+1 < img->info->nr) flood (img, i+1, j, target, replace);
             if (j-1 >= 0)            flood (img, i, j-1, target, replace);
             if (j+1 < img->info->nc) flood (img, i, j+1, target, replace);
             if (i-1 >= 0)            flood (img, i-1, j, target, replace);
       }
}
```

This particular implementation is slow and space consuming, but easy to code. Faster versions can no doubt be found on the Internet. The `flood` function requires that each pixel in the image be examined to see if it has the target value; if so, a region is grown around that *seed* pixel.

Winnowing the regions using area is simple. After a flood fill has marked a region with a new value M, the area of that region is simply the number of M-valued pixels in the entire image.

```
int area (IMAGE x, int c)
{
       int i,j,k=0;

       for (i=0; i<x->info->nr; i++)
             for (j=0; j<x->info->nc; j++)
                   if (x->data[i][j] == c) k++;
       return k;
}
```

If the area is too small (say, less than 900 pixels) or too big (perhaps 3100 pixels or bigger), then all the marked pixels are cleared (set to 255):

```
void clear (IMAGE x, int c)
{
       int i,j,k=0;

       for (i=0; i<x->info->nr; i++)
             for (j=0; j<x->info->nc; j++)
                   if (x->data[i][j] == c) x->data[i][j] = 255;
}
```

A final useful function would be `remark`, which will change the level of all pixels having one specific value to another one. The purpose is to uniquely mark a specified region. If there are a dozen regions on an image, it is useful

to know which ones are connected. Thus, we might mark the first connected region we find with the value 1, the second with 2, and so on.

```
void remark (IMAGE x, int oldg, int newg)
{
      int i,j,k=0;

      for (i=0; i<x->info->nr; i++)
            for (j=0; j<x->info->nc; j++)
                  if (x->data[i][j] == oldg)
                        x->data[i][j] = newg;
}
```

Now the first step — region identification — has the needed tools. The program `reg1.c` identifies the regions in the image that might be tomatoes and marks them with unique grey levels. Small regions are marked with values from 90 and higher, large regions are marked from 230 and higher. In the entire image, all pixels having a grey level of 90 belong to a single contiguous region, as do those having levels of 91, 92, and so on. Question: Does the region having level 90 correspond to a tomato?

So, we can return to the original issue: using color as a feature. Each pixel in the region-processed image corresponds to a color pixel in the original, and they both have the same coordinates. Pixels having a value of 90 in the region processed image can be examined in the original to see what their color is. Is there a similarity in hue within each region, and can that be used to classify the region as a tomato or a carrot? Yes.

8.1.2 Training and Testing

Let's look at the data. The image is scanned for pixels in the first region, which have the value 90. For each of these, fetch the corresponding pixel from the original color image and extract the RBG values. A mean for each color is computed, and associated with the region, and then the process is repeated for region 91, 92, and each other. Table 8.1 shows the results.

The values in the Area column are the number of pixels in each region. The Truth column is interesting and essential; it is the real classification of the object represented by the region. This was determined by a visual inspection of the image. A superficial examination of these data does not yield an obvious way to use them to determine perfectly which regions are carrots and which are tomatoes. If a chart is created showing the region class versus the region color, things become clearer.

Three scattergrams were created using the Microsoft Excel spreadsheet program. A *scattergram* shows data points or classes plotted against one or more features. Figure 8.6 shows the two classes, arbitrarily numbered 1 and 2, plotted against the value of each of the color components. This clarifies the situation rather well.

Table 8.1: Features for Classifying Vegetables

MARK	AREA	RED	GREEN	BLUE	TRUTH
90	1634	138	46	52	Tomato
91	1384	152	53	60	Tomato
92	1634	130	54	49	Tomato
230	2663	143	104	74	Carrot
93	1452	127	47	48	Tomato
94	2273	181	105	81	Carrot
95	2364	179	110	83	Carrot
96	1374	136	55	53	Tomato
97	1580	132	46	53	Tomato
98	1834	133	47	49	Tomato
99	1658	149	59	63	Tomato
231	2581	174	107	83	Carrot
100	2137	134	46	52	Tomato
232	2721	180	104	78	Carrot
101	2417	138	56	57	Tomato
102	1662	129	46	55	Tomato
103	1599	125	46	50	Tomato
104	2253	132	45	51	Tomato
105	1933	123	42	46	Tomato
106	1789	131	44	51	Tomato

The value of the green component is very useful for distinguishing the two classes, according to this figure. A vertical line can be drawn at green = 85 that has tomatoes on the left (smaller green value) and carrots on the right (larger green value). This means that for this image, tomato regions have a green component less than 85, and thus can be distinguished from carrots using a simple threshold. The same appears to be true for blue, although the gap between the two classes is smaller, and the threshold is about 70. The red component is most difficult to use as a feature, possibly because red and orange both contain significant red components. The tomatoes have red values between 110 and 170, and so require two thresholds to classify the objects. Figure 8.7 shows a classification of the regions according the blue component, as just described.

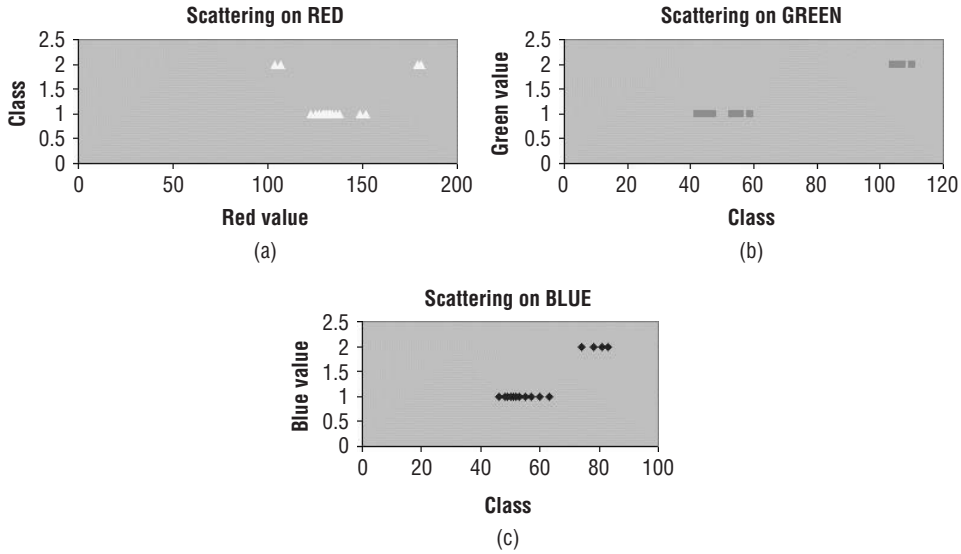


Figure 8.6: Scattergrams showing the connection between object class and color.

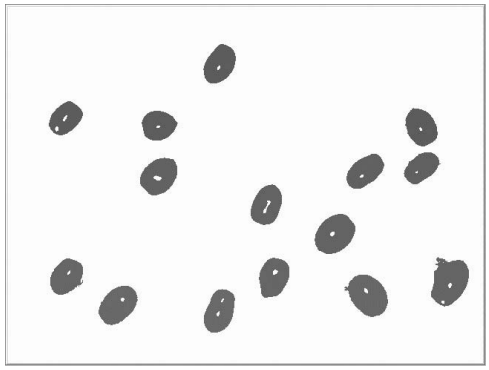


Figure 8.7: Tomatoes classified using the blue color component only. All tomatoes are found; no errors are encountered.

According to these scattergrams, only one color feature is necessary to distinguish between a tomato and a carrot. However, this is based on only one image. There would be little point in creating a vision system to analyze one image, so an assumption is that the classifier will be applied to a large set of images, one after the other, in some industrial or other real-world setting. So, it would seem that there are issues to be considered: how much data is needed to experiment with, how the values of features vary across images, and how to find features that can be used, individually or in combination, to classify an object in an image.

It is standard practice to measure and classify a set of data to establish a normal range for the features to be used in automatic classification. This is what is referred to as *training*, and it is an essential part of building a recognition

system for visual objects. The system *learns* — that is, establishes ranges for the features that were selected for use — by being given a known object and then identifying some pattern among the feature values. This works better if a large number of objects and images are used in the training process, and that means having a large set of classified objects on hand before the system is even completed. This is called *training data*. It may turn out that some of the selected features are not useful and will need to be discarded or replaced, and this will require modifications to the system.

So, for each object in each test image, all the proposed features are measured and stored. A classifier is built that uses these features to determine the class of the objects as well as can be done. Rarely will this be perfect, but it could be perfect for the training data. Finding out the actual rate of successful classification must be done using a different set of data, not the training data, because the system has been tuned specifically to recognize the training data set. We must know the actual classifications for the test data, too, since we need to determine how often the system returns the correct class. If we have 100 objects that are of known classes, then this set of data needs to be split into two sets: one for training, one for testing. For the time being, they should be split into two equal parts, but alternatives will be described starting in the next section and in the remainder of the chapter.

8.1.3 Variation: In-Class and Out-Class

Part of the problem with visual classification is that objects do not look the same in different images, in different orientations, and when seen through different cameras. Examining the data for the carrot problem, it is easy to see that tomatoes have a variety of different values for each of the features we have measured so far: color (red, green, blue) and area. Indeed, no two tomatoes have the same values for these four features. Consider the green component of tomato regions: the values in each row of the following table are an average of the green components of all pixels in the region, meaning that even within each region they are not all the same. The means over the various regions are not the same either.

	GREEN	AREA
MEAN OVER ALL TOMATOES	48.80	1756.00
STANDARD DEVIATION	4.95	299.79
MEAN OVER ALL CARROTS	106.0	2520.40
STANDARD DEVIATION	2.28	173.16

Samples of features such as these usually follow a statistical normal distribution. This is the famous bell-shaped curve, where the mean is in the center and the standard deviation specifies the width of the bell. The variation of the measurements is greater if the bell is wide, of course. A narrow range of

values, or a small standard deviation, is desired because it corresponds to an easier thresholding problem. It would also mean that the feature values would be less likely to overlap with those of other objects. A large distance between means of classes to be separated is important, too.

The situation of Figure 8.8a is a desirable one for a classification problem. Here, classes P and Q have very distinct means and a relatively small standard deviation, and so the feature values involved have a very small region where they can overlap. In this region it is not possible to accurately identify the class of the object from this feature. The situation of Figure 8.8b is much worse, because the means of the two distributions are closer together and the area of overlap is larger. There will be a greater proportion of measurements that fall into this ambiguous area. The best threshold to use is the feature value that corresponds to the point of intersection of the two normal curves, but in Figure 8.8b it seems certain this will not yield a correct classification in all cases.

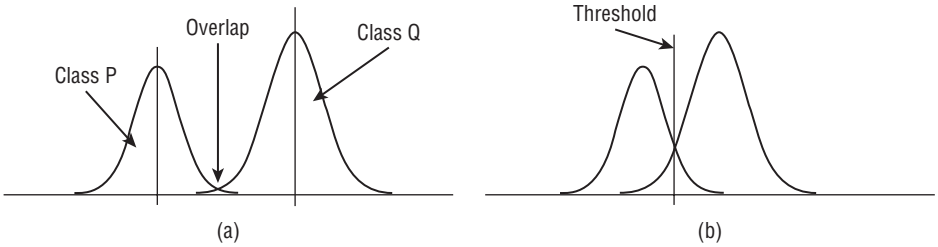


Figure 8.8: (a) The distribution of feature values between two classes, P and Q. The overlap between these distributions is small, meaning that this feature alone can distinguish between these classes. (b) A larger overlap area increases the number of feature measurements that are ambiguous. The vertical line here shows the location of the likely best threshold.

If one feature does not distinguish between the classes, then perhaps two will. As an example, let's use a classic set of data from many years ago, the *Iris* data set [Fisher, 1936; Anderson, 1935]. These data appear in Table 8.2 as numbers, and we'll not be concerned here with how the measurements were obtained. The interesting thing is how the data can be used to distinguish between three species of Iris: *setosa*, *versicolor*, and *virginica*. The measurements are width and length of petals and sepals, which are anatomical features of any flower, as illustrated in Figure 8.9a.

That no single feature can be used to classify all instances into a correct category can be established using scattergrams, or even by examining the data. Which combination is best is a harder question to answer, and how to tell is an interesting process to observe. Plotting pairs of features is useful in this case, and showing the class of the object as color in the scattergram gives effectively a third dimension to the plot, as shown in Figure 8.9b. Note that a straight line can be drawn that separates the red class (*setosa*) from the blue (*versicolor*), but no such line exists between the blue and the green (*virginica*).

Table 8.2: The Iris data set.

SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS
5.1	3.5	1.4	0.2	*setosa*	7.0	3.2	4.7	1.4	*versicolor*	6.3	3.3	6.0	2.5	*virginica*
4.9	3.0	1.4	0.2	*setosa*	6.4	3.2	4.5	1.5	*versicolor*	5.8	2.7	5.1	1.9	*virginica*
4.7	3.2	1.3	0.2	*setosa*	6.9	3.1	4.9	1.5	*versicolor*	7.1	3.0	5.9	2.1	*virginica*
4.6	3.1	1.5	0.2	*setosa*	5.5	2.3	4.0	1.3	*versicolor*	6.3	2.9	5.6	1.8	*virginica*
5.0	3.6	1.4	0.2	*setosa*	6.5	2.8	4.6	1.5	*versicolor*	6.5	3.0	5.8	2.2	*virginica*
5.4	3.9	1.7	0.4	*setosa*	5.7	2.8	4.5	1.3	*versicolor*	7.6	3.0	6.6	2.1	*virginica*
4.6	3.4	1.4	0.3	*setosa*	6.3	3.3	4.7	1.6	*versicolor*	4.9	2.5	4.5	1.7	*virginica*
5.0	3.4	1.5	0.2	*setosa*	4.9	2.4	3.3	1.0	*versicolor*	7.3	2.9	6.3	1.8	*virginica*
4.4	2.9	1.4	0.2	*setosa*	6.6	2.9	4.6	1.3	*versicolor*	6.7	2.5	5.8	1.8	*virginica*
4.9	3.1	1.5	0.1	*setosa*	5.2	2.7	3.9	1.4	*versicolor*	7.2	3.6	6.1	2.5	*virginica*
5.4	3.7	1.5	0.2	*setosa*	5.0	2.0	3.5	1.0	*versicolor*	6.5	3.2	5.1	2.0	*virginica*
4.8	3.4	1.6	0.2	*setosa*	5.9	3.0	4.2	1.5	*versicolor*	6.4	2.7	5.3	1.9	*virginica*
4.8	3.0	1.4	0.1	*setosa*	6.0	2.2	4.0	1.0	*versicolor*	6.8	3.0	5.5	2.1	*virginica*
4.3	3.0	1.1	0.1	*setosa*	6.1	2.9	4.7	1.4	*versicolor*	5.7	2.5	5.0	2.0	*virginica*
5.8	4.0	1.2	0.2	*setosa*	5.6	2.9	3.6	1.3	*versicolor*	5.8	2.8	5.1	2.4	*virginica*
5.7	4.4	1.5	0.4	*setosa*	6.7	3.1	4.4	1.4	*versicolor*	6.4	3.2	5.3	2.3	*virginica*
5.4	3.9	1.3	0.4	*setosa*	5.6	3.0	4.5	1.5	*versicolor*	6.5	3.0	5.5	1.8	*virginica*
5.1	3.5	1.4	0.3	*setosa*	5.8	2.7	4.1	1.0	*versicolor*	7.7	3.8	6.7	2.2	*virginica*
5.7	3.8	1.7	0.3	*setosa*	6.2	2.2	4.5	1.5	*versicolor*	7.7	2.6	6.9	2.3	*virginica*
5.1	3.8	1.5	0.3	*setosa*	5.6	2.5	3.9	1.1	*versicolor*	6.0	2.2	5.0	1.5	*virginica*
5.4	3.4	1.7	0.2	*setosa*	5.9	3.2	4.8	1.8	*versicolor*	6.9	3.2	5.7	2.3	*virginica*
5.1	3.7	1.5	0.4	*setosa*	6.1	2.8	4.0	1.3	*versicolor*	5.6	2.8	4.9	2.0	*virginica*
4.6	3.6	1.0	0.2	*setosa*	6.3	2.5	4.9	1.5	*versicolor*	7.7	2.8	6.7	2.0	*virginica*
5.1	3.3	1.7	0.5	*setosa*	6.1	2.8	4.7	1.2	*versicolor*	6.3	2.7	4.9	1.8	*virginica*
4.8	3.4	1.9	0.2	*setosa*	6.4	2.9	4.3	1.3	*versicolor*	6.7	3.3	5.7	2.1	*virginica*

Continued

Table 8.2: (*continued*)

SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS
5.0	3.0	1.6	0.2	*setosa*	6.6	3.0	4.4	1.4	*versicolor*	7.2	3.2	6.0	1.8	*virginica*
5.0	3.4	1.6	0.4	*setosa*	6.8	2.8	4.8	1.4	*versicolor*	6.2	2.8	4.8	1.8	*virginica*
5.2	3.5	1.5	0.2	*setosa*	6.7	3.0	5.0	1.7	*versicolor*	6.1	3.0	4.9	1.8	*virginica*
5.2	3.4	1.4	0.2	*setosa*	6.0	2.9	4.5	1.5	*versicolor*	6.4	2.8	5.6	2.1	*virginica*
4.7	3.2	1.6	0.2	*setosa*	5.7	2.6	3.5	1.0	*versicolor*	7.2	3.0	5.8	1.6	*virginica*
4.8	3.1	1.6	0.2	*setosa*	5.5	2.4	3.8	1.1	*versicolor*	7.4	2.8	6.1	1.9	*virginica*
5.4	3.4	1.5	0.4	*setosa*	5.5	2.4	3.7	1.0	*versicolor*	7.9	3.8	6.4	2.0	*virginica*
5.2	4.1	1.5	0.1	*setosa*	5.8	2.7	3.9	1.2	*versicolor*	6.4	2.8	5.6	2.2	*virginica*
5.5	4.2	1.4	0.2	*setosa*	6.0	2.7	5.1	1.6	*versicolor*	6.3	2.8	5.1	1.5	*virginica*
4.9	3.1	1.5	0.2	*setosa*	5.4	3.0	4.5	1.5	*versicolor*	6.1	2.6	5.6	1.4	*virginica*
5.0	3.2	1.2	0.2	*setosa*	6.0	3.4	4.5	1.6	*versicolor*	7.7	3.0	6.1	2.3	*virginica*
5.5	3.5	1.3	0.2	*setosa*	6.7	3.1	4.7	1.5	*versicolor*	6.3	3.4	5.6	2.4	*virginica*
4.9	3.6	1.4	0.1	*setosa*	6.3	2.3	4.4	1.3	*versicolor*	6.4	3.1	5.5	1.8	*virginica*
4.4	3.0	1.3	0.2	*setosa*	5.6	3.0	4.1	1.3	*versicolor*	6.0	3.0	4.8	1.8	*virginica*
5.1	3.4	1.5	0.2	*setosa*	5.5	2.5	4.0	1.3	*versicolor*	6.9	3.1	5.4	2.1	*virginica*
5.0	3.5	1.3	0.3	*setosa*	5.5	2.6	4.4	1.2	*versicolor*	6.7	3.1	5.6	2.4	*virginica*
4.5	2.3	1.3	0.3	*setosa*	6.1	3.0	4.6	1.4	*versicolor*	6.9	3.1	5.1	2.3	*virginica*
4.4	3.2	1.3	0.2	*setosa*	5.8	2.6	4.0	1.2	*versicolor*	5.8	2.7	5.1	1.9	*virginica*
5.0	3.5	1.6	0.6	*setosa*	5.0	2.3	3.3	1.0	*versicolor*	6.8	3.2	5.9	2.3	*virginica*
5.1	3.8	1.9	0.4	*setosa*	5.6	2.7	4.2	1.3	*versicolor*	6.7	3.3	5.7	2.5	*virginica*
4.8	3.0	1.4	0.3	*setosa*	5.7	3.0	4.2	1.2	*versicolor*	6.7	3.0	5.2	2.3	*virginica*
5.1	3.8	1.6	0.2	*setosa*	5.7	2.9	4.2	1.3	*versicolor*	6.3	2.5	5.0	1.9	*virginica*
4.6	3.2	1.4	0.2	*setosa*	6.2	2.9	4.3	1.3	*versicolor*	6.5	3.0	5.2	2.0	*virginica*
5.3	3.7	1.5	0.2	*setosa*	5.1	2.5	3.0	1.1	*versicolor*	6.2	3.4	5.4	2.3	*virginica*
5.0	3.3	1.4	0.2	*setosa*	5.7	2.8	4.1	1.3	*versicolor*	5.9	3.0	5.1	1.8	*virginica*

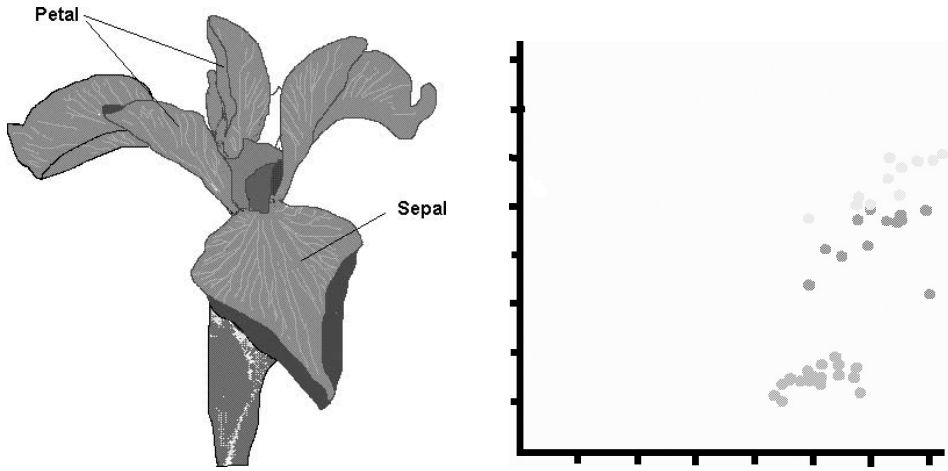


Figure 8.9: (a) The anatomy of a flower, showing the petals and sepals that are key to the Iris data set. (b) A scattergram of Sepal length vs. petal length for the three classes. Color codes the classes; note the spatial groupings.

A line breaks the green-blue region into two parts such that almost all green points are on one side and almost all blue points are on the other. This could be used to distinguish between the two classes with a small error. The line that does this is not horizontal, but that does not matter. This is called a *linear discriminant* and is commonly used in data classification and machine learning. There are many references to this technique in the literature. It is, of course, just one of many possible methods for classifying data.

8.2 Minimum Distance Classifiers

Looking again at the scattergram of Figure 8.9b, note that the data are grouped into two-dimensional regions such that it is possible to draw a curve that surrounds each class. Of course, such a curve can get very complex, and the curve would only surround the points we knew about. A new object and set of measurements may lie well outside of the curve. If an unknown object is measured and if the measurements form a point that falls inside that curve, then it probably should be classified with the others within the curve.

Because the curve is too complex to identify and hard to use as a classifier, we can introduce a simpler scheme: an unidentified region that is classified according to how far away it is (as a point) from any of the other points in the training set. Depending on how ''how far away'' is defined, this could work pretty well. This is what is commonly known as *distance*, and there are several reasonable ways to define and implement it.

8.2.1 Distance Metrics

The common, intuitive definition of distance is called *Euclidean distance*, because of Euclid's connection with many other common geometric concepts. It should be (and was in the past) called the *Pythagorean distance* because it uses the famous formula for the hypotenuse. The distance between a point $P = (p_1, p_2)$ and a point $Q = (q_1, q_2)$ is:

$$d = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} \qquad \text{(EQ 8.1)}$$

For points in a space having more than two dimensions, say N dimensions, this formula generalizes as:

$$((p_1 - q_1)^2 + (p_2 + q_2)^2 + \cdots + p_N + q_N)^2)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^{N}(p_i - q_i)^2} \qquad \text{(EQ 8.2)}$$

This is the distance ''as the crow flies,'' and while it makes sense in everyday life, there are problems with it in images. The main one is that pixel locations are integers, whereas the distance between pixels can be floating point. Another practical problem is that this calculation requires a square root operation, which is likely to take a hundred times longer to calculate than a simple integer operation. It is true that computers are faster than they used to be, but images have gotten bigger, too. Therefore, it is usual to omit the square root and work with d^2 whenever possible.

A commonly used distance measure when using pixels is the 8-distance. This is the maximum of the horizontal and vertical difference between the coordinates of the pixel; or, for the previously defined P and Q:

$$d_8 = max(|p_1 - q_1|, |p_2 - q_2|) \qquad \text{(EQ 8.3)}$$

One way to think of this is as the number of pixels between P and Q. It is called *8-distance* because the path traced between P and Q uses the eight discrete directions that are possible on a discrete grid.

If there is an 8-distance, then why not a 4-distance? There is, and it is also called *Manhattan distance* or *city block distance*. It is the distance in pixels between P and Q using only up/down and left/right directions (4 connected pixels). Mathematically:

$$d_4 = |p_1 - q_1| + |p_2 - q_2| \qquad \text{(EQ 8.4)}$$

Finally, at least for the purposes here, there is the most exotic, complex, and useful distance measure: *Mahanalobis distance*. It is difficult to explain in the general case, but for specific examples in classification it is more obvious. Consider the data in Table 8.1 again. If P and Q are the first two

entries, tomatoes both, and the area and the green component are used in a classification, then the data points are:

$$P = (1634, 46)$$
$$Q = (1384, 53)$$

The Euclidean distance between these two is:

$$\sqrt{(1634 - 1384)^2 + (46 - 53)^2} = \sqrt{62500 + 49} - 250.1$$

Now change the green component of P by 1 to (1634, 45). The distance between P and Q is now 250.13. Changing the area component by 1 so that $P = (1635, 46)$ changes the $P - Q$ distance to 251.1. This shows that a change in the first coordinate makes a bigger difference in the distance than does a change in the second. Or in other words, the scales of the two coordinate axes are different. This is very common in computer vision problems, and it really does make sense. Why would we expect that each of the measurements would have units of the same size?

Normalizing with respect to scale can be done using statistics. The standard deviation is a measure of variability, or what the range of values is. Dividing sample values by the standard deviation should narrow the range of values, and convert the units to universal ones. This is the basic idea behind Mahanalobis distance. For example, consider the same points P and Q as before and the normalized points P' and Q'. The overall standard deviations are:

$$s_{\text{area}} = 429.5 \qquad s_{\text{green}} = 25.2$$

The points are:

$$P = (1634, 46) \qquad Q = (1384, 53) \qquad \text{distance}\,(P, Q) = 250.1$$
$$P' = (3.8, 1.83) \qquad Q' = (3.2, 2.1) \qquad \text{distance}\,(P', Q') = 0.64$$

The standard deviations are used to normalize the raw sample values before computing distance. It's actually more complex than that; reality tends to make the math harder. The formula for computing the Mahanalobis distance between P and Q is:

$$d_M(P, Q) = \sqrt{(P - Q)^T S^{-1} (P - Q)} \qquad \text{(EQ 8.5)}$$

which is a matrix equation, in which P and Q are the points (vectors) for which the distance is being computed, $(P - Q)^T$ is the transpose of the difference of the vectors, and S is the *covariance matrix*.

The variance is the mean of the squared distances between a value and the mean of those values:

$$VAR = \frac{\sum_{i=1}^{n} (P_i - \mu_i)(P_i - \mu_i)}{n - 1} \qquad \text{(EQ 8.6)}$$

When two (or more) values are involved, this calculation can include combinations of the variables. In the case of P and Q:

$$COV = \frac{\sum_{i=1}^{n}(P_i - \mu_i)(Q_i - \mu_i)}{n-1} \quad \text{(EQ 8.7)}$$

So, covariance is a generalization of variance for multiple variables. The Mahanalobis distance is much more computationally expensive than the other distance measures, but it does have the important advantage of being scale independent, so is often used. However, for simplicity many people use Euclidean distance, too, and without loss of generality most of the rest of the examples will use Euclidean distance. Any distance measure may be substituted, of course.

8.2.2 Distances Between Features

Many pattern recognition tasks use a large number of features to distinguish between many classes. The Iris data set has four features, which is too many to visualize in a straightforward way, to characterize three classes. This data set will be used to illustrate distance-based classifiers, starting with the *nearest neighbor* classifier.

Given N classes $C_1, C_2, \ldots, C_N$ and M features $F_1 .. F_M$, consider the classification of an object, P. Measure all features for this object and create an M-dimensional vector, v, from them. Feature vectors for all objects in all N classes have also been created; the first such in class C_1 will be ${C_1}^1$, the eighth one in class 3 will be ${C_3}^8$, and so on. Classification of P by the nearest neighbor method involves calculating the distances between v and all feature vectors for all the classes. The class of the feature vector having the minimum distance from v will be assigned to v.

The name of the method is very descriptive. The class of an unknown target will be the same as that of its nearest neighbor in feature space. Let's see how this works using the Iris data set. First, the set needs to be broken into training data and test data: select the first half of the data for each class to be training data, and the last half as test data.

Next, feature vectors are created from the training data items. There are four features, so each vector has four components. This vector is compared against (i.e., the distance is computed to) all the training data vectors, and the class of the one with smallest distance is saved: this will be the class given to the target. This is done for each of the test data items, and success rates are computed; the raw success rate, the number of correct classifications divided by the number of test data items, is a good indicator of how good the features are and of how well the classifier will work overall.

There is another, better, way to evaluate the results. A *confusion matrix* is a table in which each column represents an actual class, and each row represents a class delivered by the classifier (an outcome). For the Iris data experiment, the confusion matrix is:

	SETOSA	VERSICOLOR	VIRGINICA
SETOSA	25	0	0
VERSICOLOR	0	24	3
VIRGINICA	0	1	22

The columns add up to the number of elements in each class, and the rows add up to the number of classifications that the classifier made to each class. The trace (sum of the elements along the diagonal) is the number of correct classifications, and the success rate is the trace divided by the total number of trials. In this instance, the success rate is nearly 95%, which is pretty good.

The nearest neighbor method is commonly implemented for a classifier because it is simple and gives pretty good results. However, if one neighbor gives good results, why not use many neighbors? This simple thought leads to the *k-nearest neighbor* method, in which the class is determined by a vote between the nearest k neighbors in feature space. This is a little more work, and can lead to ties in some cases. There are a two main ways to implement this: compute all distances and then sort them into descending order and read off the smallest k of them, or keep only the smallest k in a table and test/insert after every distance calculation. The example program provided (`nkn.c`) uses the first method. This allows the specification of k to change without much modification of the program so that the effect of changes to k can be explored.

The k-nearest neighbor algorithm should yield the same results as nearest neighbor for $k = 1$; this is a test of correctness. The results for the Iris data are as follows:

K	SUCCESS	K	SUCCESS
1	95%	12	93%
2	92%	13	95%
3	93%	14	93%
4	95%	15	93%
5	92%	16	95%
6	92%	17	96%

Continued

(*continued*)

K	SUCCESS	K	SUCCESS
7	93%	18	96%
8	95%	19	95%
9	95%	20	95%
10	93%	21	95%
11	93%	22	92%

The success of the *k*-nearest neighbor method depends on the way the data points are scattered near the overlap areas. In this case, it seems no better than the simple nearest neighbor method, but this is hard to predict in general, and it will be better sometimes.

The *nearest centroid method* uses many points as a basis for comparison, but it combines this with an ease of calculation that makes it attractive. The *centroid* is the point in a set of feature data that is in some sense the mean value. This point is a good representation of the entire set if any such place exists. The coordinates of the centroid are the mean values of the coordinates of all the points in the set; that is, the first coordinate of the centroid is the mean of all the first coordinates, and so on. For the Iris data set, this means that there are three centroids, one for each set. They are:

- Centroid 1 = (5.028000, 3.480000, 1.460000, 0.248000)
- Centroid 2 = (6.012000, 2.776000, 4.312000, 1.344000)
- Centroid 3 = (6.576000, 2.928000, 5.639999, 2.044000)

So, the nearest centroid classifier computes the distance between the sample point and the centroids, and the centroid at the smallest distance represents the classification. This has fewer computations at classification time, because the centroids are pre-computed and there is a need for only one distance calculation per class.

The results of the nearest centroid classifier for the Iris data set are precisely the same as for the nearest neighbor classifier. This will not be true for all data sets.

8.3 Cross Validation

Splitting the data sets into training and testing sets is necessary to avoid getting inflated success rates. One would expect high success on the data used for training. In the nearest neighbor classifier, for example, the success rate

on the training data should be 100%, because each of the points will be a distance of zero from at least one other — itself. Still, the selection of training versus test data is arbitrary, and the two data sets could be exchanged without distorting the results. If this is done for the Iris data using the nearest neighbor classifier, the results become as follows:

	SETOSA	VERSICOLOR	VIRGINICA
SETOSA	25	0	0
VERSICOLOR	0	23	2
VIRGINICA	0	2	23

The success rate is the same as before, but the details of the confusion matrix are different. Repeating the classification with the roles of the testing and training sets reversed gives us two different trials, though, and should give us more confidence, especially since there is relatively little data here. This process could be described as a 2-way (or 2-fold) cross validation.

The general description of cross validation is a process for partitioning data repeatedly into distinct training and testing sets. There are many ways to do this, some of them wrong. Any partition that uses the same samples in both sets would normally be in error, for example, and creating new data points based on statistical samples may in some instances be fine, but is not cross validated. Cross validation takes the data that exists and partitions it into training/testing sets multiple times so that the sets are different.

An *n-way cross validation* breaks the data into *n* more-or-less equal parts. Then each of these in turn is used as test data, while all the other parts together are used as training data. This gives *n* results, and the overall result is the average of those *n*. The Iris data set has 150 samples in all, so a 5-way cross validation would provide a convenient partitioning into 5 groups of 30 points each. There is no rule that says there have to be exactly the same number of samples in each set, although there should be as many examples of each class as possible.

The program `cross5.c` works the same way as the nearest neighbor program, except that it reads all the Iris data into one large array at the beginning and then partitions it before each experiment. The result is five distinct experiments with five confusion matrices and success rates:

	PARTITION 1	PARTITION 2	PARTITION 3	PARTITION 4	PARTITION 5
SUCCESS	96.7	96.7	93.3	93.3	100.0

This yields an average of 96%.

Cross validation can be done using random samples of the data, too. A test set would be built from random selections of the full data set, making sure

not to choose the same item more than once. All the items not selected will be the training set. In principle, this can be repeated arbitrarily many times, but nothing is gained by doing so. Between 5 and 10 trials would be sufficient for the Iris data set. Using random cross validation, keeping the classes balanced, and with 10 examples from each class in the test set, and overall success rate averaged over ten trials, a 93% success rate was obtained. This would be a little different each time due to the random nature of the experiment.

What might be called the ultimate in cross validation picks a single sample from the entire set as test data, and uses the rest as training data. This can be repeated for each of the samples in the set, and the average over all trials gives the success rate. For the Iris data, there would be 150 trials, each with a single classification. This is called *leave-one-out cross validation*, for obvious reasons.

For the Iris set again, leave-one-out cross validation leads to an overall success rate of 96% when used with a nearest neighbor classifier; it's probably the best that can be done. This is a good technique for use with smaller data sets, but is really too expensive for large ones.

8.4 Support Vector Machines

Section 8.1.3 discussed the concept of a linear discriminant. This is a straight line that divides the feature values into two groups, one for each class, and is an effective way to implement a classifier if such a line can be found. In higher dimensional spaces — that is, if more than two features are involved — this line becomes a plane or a hyperplane. It's still linear, just complicated by dimensionality. Samples that lie on one side of the plane belong to one class, while those on the other belong to a different class. A *support vector machine* (SVM) is a nitro-powered version of such a linear discriminant.

There are a couple of ways in which an SVM differs from simpler linear classifiers. One is in the fact that an SVM attempts to optimize the line or plane so that it is the *best* one that can be used. In the situation illustrated in Figure 8.10 there are two classes, white and black. Any of the lines shown in 8.10a will work to classify the data, at least the data that is seen there. New data could change the situation, of course. Because of that it would be good to select the line that does the best possible job of dividing the plane into the two areas occupied by the two classes. Such a line is shown in Figure 8.10b. The heavy dark line is the best line, and the thin lines on each side of it show the space between the two classes — the heavy line divides this space evenly into two parts, giving a maximum *margin* or distance between the groups. The point of an SVM is to find the maximum margin hyperplane. A line divides two-dimensional data into two parts; a plane divides three-dimensional data into two parts; and a hyperplane is a linear function that divides *N*-dimensional data into two parts. The maximum margin hyperplane is always as far from both data sets as possible.

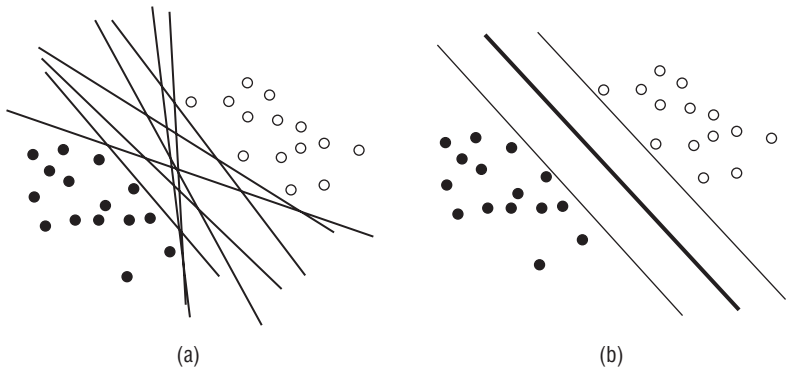


Figure 8.10: (a) A collection of straight lines that separate two classes. (b) The best line, or maximum margin line/plane/ hyperplane. The white area between the classes is the margin.

Finding a maximum or minimum margin is an optimization problem, and there are many methods for solving these [Bunch, 1980; Fletcher, 1987; Kaufman, 1998; Press, 1992], but they are beyond the scope of the present discussion. It suffices to say that it can be done. The basic idea, though, is to use feature vectors on the convex hull of the data sets as candidates to be used to guide the optimization. The candidates are called *support vectors* and are illustrated, along with the convex hulls for the data sets, in Figure 8.11. The support vectors completely define the maximal margin line, which is the line that passes as far as possible from all three of those vectors. There can be more than three support vectors, but not fewer.

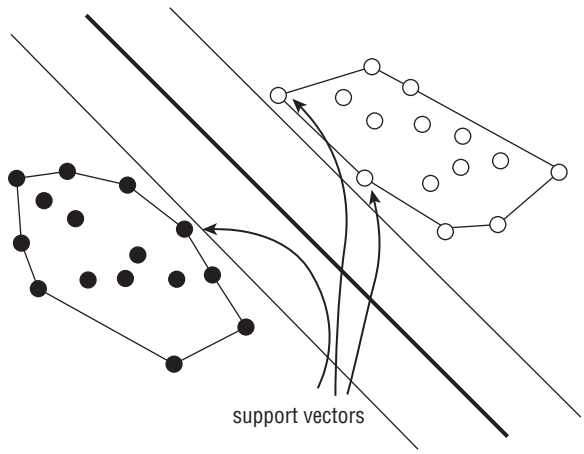


Figure 8.11: The convex hull of the feature vectors for the two classes, and the three support vectors for the final maximal margin line.

Support vector machines can also find non-linear boundaries between classes, which is their other major advantage over other methods. This is not

done by finding curved or piecewise linear paths between the feature vectors of each class, but in fact is accomplished by transforming those feature vectors so that a linear boundary can be found. A simple and clear example of this situation can be seen in Figure 8.12a, where the vectors of one class completely surround those of the other. It is obvious that there is no line or plane that can divide these vectors into the two classes.

A transformation of these vectors can yield a separable set. The vectors shown are in two dimensions; they lie in a plane. If we add a dimension and transform the points appropriately into a third dimension, a plane can be found that divides the classes (Figure 8.12b). The data has been projected into a different, higher dimensional feature space. In SVM parlance, this transformation uses a *kernel*, which is the function that projects the data. There are many possible kernels; Figure 8.12 shows the result of using a Gaussian (a *radial basis* function), but polynomials and other functions can be used, depending on the data.

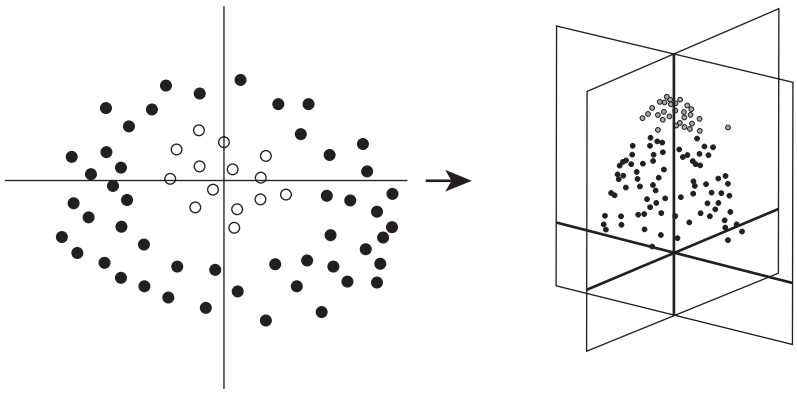


Figure 8.12: (a) Feature vectors for two classes that cannot be separated linearly. (b) The same vectors after being projected into a third dimension using a radial basis function. The can now be separated using a plane.

So, the points near the origin are given a larger value in the third dimension than those farther away, pushing the feature vectors near (0, 0) to a greater height. The maximal margin plane will cut the points in two in this third dimension, giving a perfect linear classifier.

Incidentally, SVMs can distinguish between only two classes. If there are more classes, an SVM classifier must approach them pair-wise. This is true of any classifier that uses linear discriminants.

This has been a fairly high-level description of support vector machines. It is a complex subject about which volumes have been written; see Burges,

Vapnick, and Witten for more details. Actual working software can be found in many places on the Internet, including the WEKA system (`www.cs.waikato.ac.nz/ml/weka/`), SVM^{light} (`http://svmlight.joachims.org/`), and LIBSVM (`www.csie.ntu.edu.tw/~cjlin/libsvm/`), for starters. There are many links at `www.support-vector.net/software.html`.

8.5 Multiple Classifiers—Ensembles

In complex situations, where there are many classes and many features, it is often true that some classifiers work better for some of the classes than others. One classifier may be able to identify cars in an image, for example, while another is better at trucks, or perhaps even hatchbacks. It may also be that some classifiers work better in some kinds of lighting, or in the presence of specific sorts of noise. In those situations it may be desirable to use more than one kind of classifier, and to merge the results after classification. These are referred to as *ensemble classifiers*.

The key with an ensemble is to find a way to merge the diverse results from the individual classifiers. They may be of quite different types and have very different methods, but all have the same basic goal, even if the problem has been distributed. In the following description, the hand-printed digital recognition problem of Chapter 9 will be developed. In this problem, an image is presented to the classifier that contains a single hand-printed digit, 0 through 9, which has been scanned or otherwise converted into image form. The question: what digit is this?

8.5.1 Merging Multiple Methods

A classifier can produce one of three kinds of output. The simplest and probably the most common is a basic, unqualified expression of the class determined for the data object. For a digit-classification scheme, this would mean that the classifier might simply state, ''This is a SIX,'' for example; this will be called a *type 1* response [Xu, 1992]. A classifier may also produce a ranking of the possible classes for a data object. In this case, the classifier may say, ''This is most likely a FIVE, but could be a THREE, and is even less likely to be a TWO.'' Probabilities are not associated with the ranking. This will be called a *type 2* response. Finally, a classifier may give a probability or other such confidence rating to each of the possible classes. This is the most specific case of all, since either a ranking or a classification can be produced from it. In this case, each possible digit would be given a confidence number that can be normalized to any specific range. This will be called a *type 3* response.

Any reasonable scheme for merging the results from multiple classifiers must deal with three important issues:

1. The response of the multiple classifier must be the best one given the results of the individual classifiers. It should in some logical way represent the most likely true classification, even when presented with contradictory individual classifications.
2. The classifiers in the system may produce different types of response. These must be merged into a coherent single response.
3. The multiple classifier must yield the correct result more often than any of the individual classifiers, or there is no point.

The first problem has various potential solutions for each possible type of response, and these will be dealt with first.

8.5.2 Merging Type 1 Responses

Given that the output of each classifier is a single, simple classification value, the obvious way to combine them is by using a voting strategy. A majority voting scheme can be expressed as follows: let $C_i(x)$ be the result produced by classifier i for the digit image x, where there are k different classifiers in the system; then let *H(d)* be the number of classifiers giving a classification of d for the digit image x, where d is one of {*0,1,2,3,4,5,6,7,8,9*}. H can be thought of as a histogram, and could be calculated in the following manner:

```
for (i=0; i<k; i++)
       H[ Ci(x) ] += 1;
```

Then, the overall classification E, expressing the opinions of the k classifiers, could be:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } H(j) > \dfrac{k}{2} \\ 10 & \text{otherwise} \end{cases} \qquad \text{(EQ 8.8)}$$

This is called a *simple majority vote* (SMV). For comparison, a *parliamentary majority vote* would simply select j so that *H(j)* was a maximum. An easy generalization of this scheme replaces the constant $k/2$ in the above expression with $k^*\alpha$ for $0 <= \alpha <= 1$ [Xu, 1992]. This permits a degree of flexibility in deciding what degree of majority will be sufficient, and will be called a *weighted majority vote* (WMV). This scheme can be expressed as:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } H(j) > \alpha k \\ 10 & \text{Otherwise} \end{cases} \qquad \text{(EQ 8.9)}$$

For example, many important votes in government and administrative committees require a 2/3 majority in order to pass. This would be equivalent to a value of $\alpha = 2/3$ in Equation 8.9.

Neither of the preceding two equations takes into account the possibility that all the dissenting classifiers agree with each other. Consider the following cases. In case A there are ten classifiers, with six of them supporting a classification of ''6,'' one supporting ''5,'' one supporting ''2,'' and two classifiers rejecting the input digit. In case B, using the same ten classifiers, six of them support the classification ''6,'' and the other four all agree that it is a ''5.'' Do cases A and B both support a classification of ''6,'' and do they do so equally strongly?

One way to incorporate dissent into the decision is to let max1 be the number of classifiers that support the majority classification j (max1 = $H(j)$), and to let max2 be the number supporting the second most popular classification h (max2 = $H(h)$). Then the classification becomes:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } \max1 - \max2 >= (\alpha k) \\ 10 & \text{Otherwise} \end{cases} \qquad \text{(EQ 8.10)}$$

where α is between 0.0 and 1.0. This is called a *dissenting-weighted majority vote* (DWMV).

8.5.3 Evaluation

A multiple classifier system involves passing the input image to each classifier, gathering the results from each, and then using those as a vote on the final result. This must be done repeatedly for images having a known content, and the results of the vote compared against the known true value. Each of these is one trial, and the percentage of the trials that yield a correct answer gives a key metric of the value of the classifier combination. Some methods are more difficult to evaluate. For example, WMV requires an assessment of the effect of the value of α on the results. A way to deal with this is to write a small program to vary α from 0.05 to 0.95, classifying all sample digits on each iteration.

This evaluation process can be then repeated multiple times, omitting one of the classifiers each time to test the relative effect of each classifier on the overall success. If omitting a classifier actually improves the result, then that classifier should be removed from the collection for that kind of data. This much data requires a numerical value that can be used to assess the quality of the results. The recognition rate could be used alone, but this does not take into account that a rejection is much better than a misclassification; both would count against the recognition rate. A measure of *reliability* can be computed as:

$$\text{Reliability} = \frac{\text{Recognition}}{100\% - \text{Rejection}} \qquad \text{(EQ 8.11)}$$

The reliability value will be low when few misclassifications occur. Unfortunately, it will be high if recognition is only 50%, with the other 50% being rejections. This would not normally be thought of as acceptable performance. A good classifier will combine high reliability with a high recognition rate; in

that case, why not simply use the product *reliability*recognition* as a measure of performance? In the 50/50 example above, this measure would have the value 0.5: reliability is 100% (1.0) and recognition is 50% (0.5). In a case where the recognition rate was 50%, with 25% rejections and 25% misclassifications, this measure will have the value 0.333, indicating that the performance is not as good. The value *reliability*recognition* will be called *acceptability*. The first thing that should be done is to determine which value of α gives the best results, which is more accurately done when the data is presented in tabular form or as a graph of alpha versus acceptability. For example, consider the data in Table 8.3.

Table 8.3: Acceptability of the Multiple Classifier Using a Weighted Majority Vote

ALPHA	ACCEPTABILITY
0.05	0.992
0.25	0.993
0.50	0.978
0.75	0.823

Given that this is a table of results from the multiple classifier using WMV, it can be concluded that α should be between 0.25 and 0.5, for in this range the acceptability peaks without causing a drop in recognition rate.

8.5.4 Converting Between Response Types

Before proceeding to analyze methods for merging type 2 responses (ranks), it would be appropriate to discuss means of converting one response type to another. In particular, not all the classifiers yield a rank ordering, and this will be needed before merging the type 2 responses with those of types 1 and 3.

- **Type 3 to Type 1** — Select the class having the maximum confidence rating as the response.
- **Type 3 to Type 2** — Sort the confidence ratings in descending order. The corresponding classes are in rank order.
- **Type 2 to Type 1** — Select the class having the highest rank as the type 1 response.

Converting a type 1 response to a type 3 cannot be done in a completely general and reliable fashion. However, an approximation can be based on the measured past performance of the particular algorithm. Each row in the confusion matrix represents the classifications actually encountered for a

particular digit with that classifier expressed as a probability, and the columns represent the other classifications possible for a specified classification; this latter could be used as the confidence rating. The conversions from type 1 can be expressed as:

- **Type 1 to Type 3** — Compute the confusion matrix K for the classifier. If the classification in this case is j, then first compute:

$$S = \sum_{i=0}^{9} K(i,j) \qquad \text{(EQ 8.12)}$$

  Now compute the type 3 response as a vector V, where

$$V(i) = \frac{K(i,j)}{S} \qquad \text{(EQ 8.13)}$$

- **Type 1 to Type 2** — Convert from type 1 to type 3 as above, and then convert to type 2 from type 3.

8.5.5 Merging Type 2 Responses

The problem encountered when attempting to merge type 2 responses is as follows: given M rankings, each having N choices, which choice has the largest degree of support? For example, consider the following 3-voter/4-choice problem [Straffin, 1980]:

Voter 1: a b c d **Voter 2:** c a b d **Voter 3:** b d c a

This case has no majority winner; a, b and c each get one first place vote. Intuitively, it seems reasonable to use the second place votes in this case to see if the situation resolves itself. In this case, b receives two second place votes to a's one, which would tend to support b as the overall choice. In the general case, there are a number of techniques for merging rank-ordered votes, four of which will be discussed here.

The *Borda count* [Borda, 1781; Black, 1958] is a well-known scheme for resolving this kind of situation. Each alternative is given a number of points, depending on where in the ranking it has been placed. A selection is given no points for placing last, one point for placing next to last, and so on, up to $N-1$ points for placing first. In other words, the number of points given to a selection is the number of classes below it in the ranking. For the 3-voter/4-choice problem, the situation is:

Voter 1: a (3) b (2) c (1) d (0)

Voter 2: c (3) a (2) b (1) d (0)

Voter 3: b (3) d (2) c (1) a (0)

where the points received by each selection appears in parentheses behind the choice. The overall winner is the choice receiving the largest total number of points:

$$a = 3 + 2 + 0 = 5$$
$$b = 2 + 1 + 3 = 6$$
$$c = 1 + 3 + 1 = 5$$
$$d = 0 + 0 + 2 = 2$$

This gives choice b as the ''Borda winner.'' However, the Borda count does have a problem that might be considered serious. Consider the following 5-voter/3-choice problem:

Voter 1: a b c **Voter 2:** a b c **Voter 3:** a b c

Voter 4: b c a **Voter 5:** b c a

The Borda counts are $a = 6$, $b = 7$, $c = 2$, which selects b as the winner. However, a simple majority of the first place votes would have selected a! This violates the so-called *majority criterion* [Straffin, 1980]:

If a majority of voters have an alternative X as their first choice, a voting rule should choose X.

This is a weaker version of the *Condorcet winner criterion* [Condorcet, 1785]:

If there is an alternative X which could obtain a majority of votes in pair-wise contests against every other alternative, a voting rule should choose X as the winner.

This problem may have to be taken into account when assessing performance of the methods.

A procedure suggested by Thomas Hare [Straffin, 1980] falls into the category of an *elimination* process. The idea is to repeatedly eliminate undesirable choices until a clear majority supports one of the remaining choices. Hare's method is as follows: If a majority of the voters rank choice *X* in first place, then *X* is the winner; otherwise, the choice with the *smallest number of first place votes* is removed from consideration, and the first place votes are re-counted. This elimination process continues until a clear majority supports one of the choices.

The Hare procedure satisfies the majority criterion but fails the Condorcet winner criterion, as well as the *monotonicity criterion*:

If X is a winner under a voting rule, and one or more voters change their preferences in a way favorable to X without changing the order in which they prefer any other alternative, then X should still be the winner.

No rule that violates the monotonicity criterion will be considered as an option for the multiple classifier. This decision will eliminate the Hare procedure, but not the Borda count. With the monotonicity criterion in mind, two relatively simple rank merging strategies become interesting. The first is by Black [Black, 1958], and chooses the winner by the Condorcet criterion if

such a winner exists; if not, the Borda winner is chosen. This is appealing in its simplicity, and can be shown to be monotonic. Another strategy is the so-called *Copeland rule* [Straffin, 1980]: for each option compute the number of pair-wise wins of that option with all other options, and subtract from that the number of pair-wise losses. The overall winner is the class for which this difference is the greatest. In theory this rule is superior to the others discussed so far, but it has a drawback in that it tends to produce a relatively large number of tie votes in general.

8.5.6 Merging Type 3 Responses

The classifier systems discussed so far have no single classifier that gives a proper type 3 response. Because of this, the problem of merging type 3 responses was not pursued with as much vigor as were the type 1 and 2 problems. Indeed, the solution may be quite simple. Suen [Xu, 1992] decides that any set of type 3 classifiers can be combined using an averaging technique. That is,

$$P_E(x \in C_i|x) = \frac{1}{k}\sum_{j=1}^{k} P_j(x \in C_i|x), i = 1, \ldots, M \qquad \text{(EQ 8.14)}$$

where P_E is the probability associated with a given classification for the multiple classifier, and P_k is the probability associated with a given classification for each individual classifier k. The overall classification is the value j, for which

$$P_E(x \in C_j|x) \qquad \text{(EQ 8.15)}$$

is a maximum.

8.6 Bagging and Boosting

The methods referred to in the literature as bagging and boosting are re-sampling and weighting schemes designed to improve the overall success rate of a classifier.

8.6.1 Bagging

Bagging (or *bootstrap aggregation*) involves creating multiple training sets from the overall set of training data. Each set is drawn at random from the base set, with replacement. This means that the same training item could appear more than once in a particular training set. Each set has the same size, N, and there are T sets. A classifier is trained using each of the T data sets (sometimes called *bootstrap samples*), meaning that the classifier is trained using bootstrap sample

1 and the result is called *classifier 1*; then the classifier is trained again using bootstrap sample 2, and this is called *classifier 2*; and so on for all T samples, yielding T classifiers. These T classifiers are then combined using a majority vote to give a single classification that is based on many similar but differently trained classifiers.

A popular claim, due to Breiman [1996], is that bagging works best on ''unstable'' learning algorithms. In these cases, a small change in the training set can create a large change in classifications. Such unstable algorithms include neural networks and decision trees.

8.6.2 Boosting

The idea behind bagging is fundamentally the technique called *boosting*. In this technique, classifiers are trained on a sequence of training data sets. Unlike in bagging, however, the training sets are selected with a purpose. Samples that have failed to be classified by previous iterations of the process become increasingly likely to be used in training sets. Thus, the current iteration of boosting is an attempt to explicitly create a classifier, for example, where the previous failed to succeed. Another, early, name for boosting was *arcing*, from the descriptive phrase **a**daptively **r**esample and **c**ombine.

The method begins at iteration 1. The training set will contain N items from a set of M in total, and these are selected at random. That means that the probability of any item being used in the training set is $1/N$. The classifier is then trained on these data and tested. At this point, a set of probabilities is calculated for each data item based on whether it was classified successfully. Training set items remain at $1/N$, whereas items that failed to be classified have their probability increase. Then a second set of training items is chosen, one in which the failed items from the previous trial are more likely to be included. Selection is done with replacement, so the same hard to classify items could be used in the same set many times. The process creates a set of classifiers, each one more likely to succeed on the difficult items. All classifiers are used in an ensemble system, and the overall success rate should be higher than any one classifier. A boosting scheme usually has a weighted voting scheme, where each classifier's vote has a different value. Bagging considers the votes of each classifier as being worth the same amount.

After classifying all items in iteration t, an error is computed as:

$$e_t = \sum_{\text{All misclassified items } i} \frac{w^t i}{\sum_i w^t i} \qquad \text{(EQ 8.16)}$$

These error values will be used to update the weights for the next iteration. First, a scale factor is determined. One example is:

$$a_t = \frac{1}{2}\log\left(\frac{(1-e_t)}{e_t}\right) \quad \text{(EQ 8.17)}$$

Then the weights are updated according to the following scheme:

$$\begin{aligned} w_i^{t+1} &= w_i^{t-a_t} e \quad \text{if } i \text{ represented a correct classification} \\ w_i^{t+1} &= w_i^{t a_t} e \quad \text{if } i \text{ represented a incorrect classification} \end{aligned} \quad \text{(EQ 8.18)}$$

The weights are then normalized so that they sum to 1.0 (divide each one by the sum of them all).

There are many kinds of boosting algorithms listed in the literature. Most differ in the way that the probabilities are computed for selecting data for the next iteration, and in the way that the classifier votes are weighted in the ensemble. The result is a linear combination of the classifier sequence:

$$f(x) = \sum_{t=1}^{T} w_t C_t(x) \quad \text{(EQ 8.19)}$$

where the w_t are weights and the C_t are the classifiers.

Adaboost (Adaptive boosting) [Freund, 1995] was an early development and remains a popular choice today. (It uses the scale factor of Equation 8.17, by the way.) C++ code can be found on the Web (e.g., `www.di.unipi.it/%7Egulli/coding/adaboost.tgz`). *LPBoost* is a scheme that uses linear programming to maximize the margin between training sets [Demiriz, 2002].

8.7 Website Files

`nn.c`	C program to compute the nearest neighbor classification of the Iris data
`nkn.c`	k-nearest neighbor classifier for Iris data
`nc.c`	Nearest centroid classifier
`reg1.c`	Region marking program for tomato/carrot image
`cross5.c`	5-way cross validation for Iris data
`loo.c`	Leave-one-out cross validation
`iris-train1.txt`	Training data, Iris *setosa*
`iris-train2.txt`	Training data, Iris *versicolor*

`iris-train3.txt`	Training data, Iris *virginica*
`iris-test1.txt`	Test data, Iris *setosa*
`iris-test2.txt`	Test data, Iris *versicolor*
`iris-test3.txt`	Test data, Iris *setosa*
`iris-data.txt`	Complete Iris data set

8.8 References

Anderson, E. ''The Irises of the Gaspé Peninsula.'' Bulletin of the American Iris Society 59 (1935): 2–5.

Black, D. *The Theory of Committees and Elections*, Cambridge: Cambridge University Press, 1958.

Borda, Jean-Charles de. ''Mémoire sur les Elections au Scrutin,'' *Histoire de l'Académie Royale des Sciences*. Paris, 1781.

Breiman, L., ''Bagging Predictors.'' *Machine Learning* 24, no. 2 (1996): 123–140.

Brams, S. J., and P. C. Fishburn. *Approval Voting*, Boston: Birkhauser, 1983.

Bunch, J. R., and L. Kaufman. ''A Computational Method for the Indefinite Quadratic Programming Problem.'' *Linear Algebra and its Applications* 34 (1980): 341–370.

Burges, C. J. C. ''A Tutorial on Support Vector Machines for Pattern Recognition,'' *Data Mining and Knowledge Discovery* 2 (1998): 121–167.

Condorcet, Marquis de. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Paris, 1785.

Demiriz, A., K. P. Bennett, and J. Shawe-Taylor. ''Linear Programming Boosting via Column Generation.'' *Kluwer Machine Learning* 46 (2002): 225–254.

Devijver, P. A., and J. Kittler, *Pattern Recognition: A Statistical Approach*. London: Prentice-Hall, 1982.

Duda, R. O., P. E. Hart, and D. H. Stork. *Pattern Classification* (2nd ed.). Wiley Interscience, 2000.

Enelow, J. M., and M. J. Hinich. *The Spatial Theory of Voting: An Introduction*. Cambridge: Cambridge University Press, 1984.

Farquharson, R. *Theory of Voting*, New Haven: Yale University Press, 1969.

Fisher, R. A. ''The Use of Multiple Measurements in Taxonomic Problems.'' *Annals of Eugenics* 7 (1936): 179– 188. `http://digital.library.adelaide.edu.au/coll/special//fisher/138.pdf`.

Fletcher, R. *Practical Methods of Optimization*. 2nd. ed. John Wiley and Sons, Inc., 1987.

Freund, Y., and R. E. Schapire. ''A Short Introduction to Boosting.'' *Journal of Japanese Society for Artificial Intelligence* 14, no. 5 (September, 1999):771–780.

Freund, Y. ''Boosting a weak learning Algorithm by Majority.'' *Information and Computation* 121, no. 2 (1995): 256–285.

Freund, Y. ''An Adaptive Version of the Boost by Majority Algorithm.'' *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, 1999.

Ho, T. K., J. J. Hull, and S. N. Srihari. ''Decision Combination in Multiple Classifier Systems.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16, no. 1 (January 1994).

Kaufman, L. ''Solving the Quadratic Programming Problem Arising in Support Vector Classification.'' In *Advances in Kernel Methods: Support Vector Learning*, edited by Bernhard Schölkopf, Chrisopher J. C. Burges, and Alexander J. Smola. Cambridge, MA: MIT Press, 1998.

McLachlan, G. J. *Discriminant Analysis and Statistical Pattern Recognition*. Wiley Interscience, 2004.

Parker, J. R. *Practical Computer Vision Using C*. New York: John Wiley & Sons, Inc., 1994.

Picard, Richard, Dennis Cook. ''Cross-Validation of Regression Models.'' *Journal of the American Statistical Association* 79, no. 387 (1984): 575–583.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vettering. *Numerical Recipes in C: The Art of Scientific Computing*. 2nd ed. Cambridge: Cambridge University Press, 1992.

Shawe-Taylor, J., and N. Cristianini. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge: Cambridge University Press, 2000.

Straffin, P. D., Jr. *Topics in the Theory of Voting*. Boston: Birkhauser, 1980.

Vapnik, V. *The Nature of Statistical Learning Theory*. Springer, 1995.

Witten, I. H., and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann, 2000.

Xu, L., A. Krzyzak, and C. Y. Suen. ''Methods of Combining Multiple Classifiers and Their Application to Handwriting Recognition.'' *IEEE Transactions on Systems, Man, and Cybernetics* 22, no. 3.

CHAPTER

9

Symbol Recognition

9.1 The Problem

Reading is such a fundamental part of daily life that few of us give much thought to how it is accomplished, unless we are actively involved in either learning to read or teaching someone else how to read. The first step in teaching reading is often the teaching of the alphabet. The ability to recognize letters and digits (*characters*) is fundamental to intepreting printed language; however, for a computer, a character on a page is merely another image or object to be recognized. Much of the power of the techniques that have been discussed to this point needs to be brought to bear on the apparently simple problem of recognizing the letter *a*.

It is not known how humans recognize visual objects with so little effort. Even if it were, there is no compelling reason to think that the same method could be used on a computer; the human brain and a computer are not all that similar at a detailed level. The problem of *optical character recognition* (OCR), which is the problem of the automatic recognition of raster images as being letters, digits, or some other symbol, must be approached like any other problem in computer vision.

The problem is useful and interesting because of how much information is stored in printed form. A visit to the local library is enough to convince anyone of the utility of a computer system that can visually process the printed word. In addition, characters appears in almost any real-world scene, printed on billboards, license plates, menus, signs, and even tattooed on flesh. Maps and charts (even in digital form) frequently contain raster versions of place names

and directions. Even the ubiquitous fax machine transmits its data as an image; most computers that can receive a fax cannot convert it into an ASCII text file, but instead store it as a (much larger) binary image.

An optical character recognition system must do a number of things, and do them with a high degree of precision. Let's assume that the input to the system is an image of a page of text. The first thing to do is to confirm the orientation of the text on the page; sometimes a page is not quite square to the scanning device. The image must then be segmented, first into black and white pixels, then into lines of text, and finally into individual *glyphs*, which are images of individual symbols. A recognition strategy is then applied to each glyph. If the symbol is one that the system has been trained to recognize, then there is a measured probability of a correct recognition, and usually a nonzero probability of a wrong answer. The recognized characters are collected into words and sentences, and must be output in the correct order.

A good spelling checker is useful here. Most people read words rather than individual letters, and the additional information provided by context can be useful. For a simple example, consider the word *Because*. The uppercase letter *B* is often confused with the digit *8* by character recognition systems, and for an obvious reason. If this occurred, the result would be *8ecause*, and a spelling checker would immediately discard this as a possibility — a good one would even fix it! Otherwise, the letters would be replaced, one after the other and starting with the most likely error, with the most probable mistakes. Sooner or later, the *8* would be replaced by a *B*, and the word would be complete.

There are other problems to be solved, especially in mixed documents containing both graphics and text. What parts of the page correspond to graphics? Should the graphics areas be examined to see if they contain text, also? What about different sizes and styles of text (italics, bold, various fonts)? And, in particular, handprinted characters are a nightmare; no two of them are exactly alike, even when printed carefully by the same writer.

These problems will be dealt with one at a time, and in isolation from one another. The goal is to produce a working OCR system that can be used to extract the text from a fax received by a PC, but other issues such as handprinted characters will be examined as well.

9.2 OCR on Simple Perfect Images

The basic recognition problem will be addressed first, followed by a discussion of the outlying problems. Consider that a bilevel image of a page of text exists: The problem is to recognize the characters (and therefore the words) on the page, and create a text file from them. Except in special circumstances (such as pages containing mathematics or multiple languages) there would be no more than 128 different characters that need to be identified. Since a computer

is a rather fast machine, it is possible to check an incoming glyph against all possible characters, and *classify* the input as the best match.

In order to do this the program must be trained to recognize all the possible characters. This may be a simple matter of providing templates for each one, or a complex issue involving the processing of thousands of known documents, depending on the method used. A standard text image is useful at this stage, one that contains all the characters to be recognized in a known order. In this way the program can train itself; the first character on the text image is known, and the example on the page is measured by the program and used as an example of its class.

Figure 9.1 shows the training image that will be used by the system being devised. All the typical characters found on a North American keyboard are present twice. The use of variations in spacing is intended to at least partially account for variations in sampling. A letter that occurs one centimeter from the left margin may have a different appearance as a raster glyph from the same letter that occurs 1.5 centimeters from the margin, depending on the actual size of a pixel.

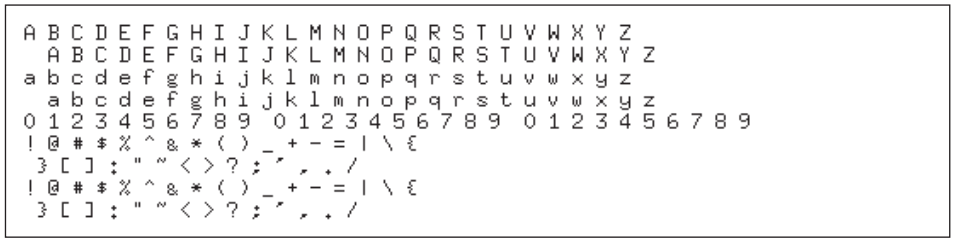


Figure 9.1: Example text image, used to teach the OCR system the basic character set. All the usual characters on a North American keyboard are present in various positions.

Since the orientation is perfectly horizontal, the first step is to determine the position and extent of the lines of text in the image. This can be done by constructing a horizontal *projection* and searching it for minima. The projection is simply the sum of the pixel values in a specified direction, so a horizontal projection is the sum of the pixels in each row. The row of pixels that begins a new line will be one in which some of the pixels are black, and the last row belonging to that line will be the last one having any black pixels. The start and end columns for the line are found by searching the rows that belong to that line, from column zero through to the first column having a set pixel. The same is done, but in the reverse direction, to find the last set pixel in a line.

When this has been done, the bounding box for the text lines is known. Individual characters can be found beginning at the leftmost column belonging to the uppermost line. Starting at the first column containing a black pixel, copy pixels from the image into a temporary glyph image until a column

having no black pixels is seen; the range for the rows is the row range for the entire line, as found previously (the row extents of the bounding box). Now, the eight lines of text are known initially, since the program will be run using the test image as input data. This means that each character read from the image has a known classification. The width, height, and spacing of these characters can be measured and saved in a database for this particular font, along with the sample of the glyph that was extracted from the image.

The database thus obtained has a very simple structure, as shown in Figure 9.2. The entire collection of information is stored as an array having 256 elements, and which is indexed by the ASCII code for the character involved. Each entry in this array is a list of templates in which the actual size of the glyph (as well as the pixels themselves) can be found. The list contains multiple entries that can be used to store variations on the extracted glyphs, including different sizes and styles of the same character, if needed. The database can be saved in a file and used when needed. New fonts and styles can be learned and added to the database; as it grows, the recognition system that uses it becomes able to recognize a wider variety of characters.

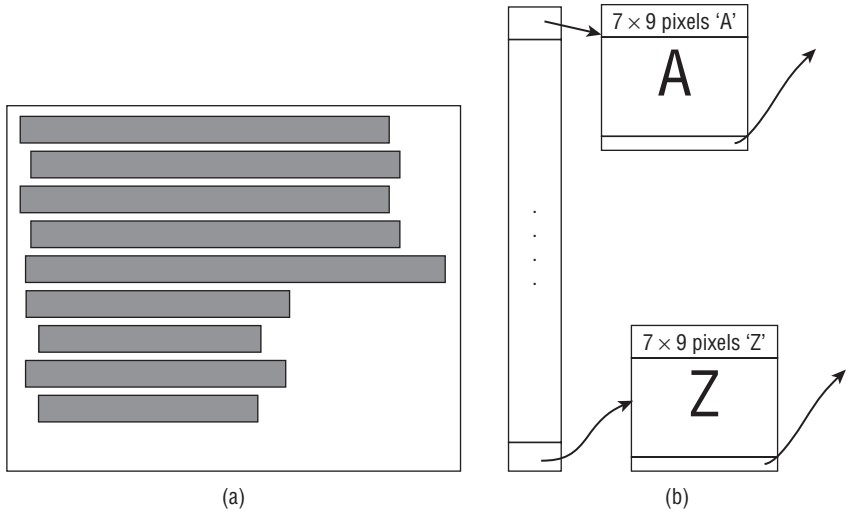


Figure 9.2: (a) The lines found in the training image (Figure 9.1). (b) The structure of the database in which the glyph information is stored. Everything needed for template matching is available in the initial version seen here.

The program named `learn.c` will examine a sample image and create a new database. The sample image must be that of Figure 9.1, and the database file created is specified as an argument. For example,

```
learn testpage.pbm helv.db
```

was used to create the small database for the Helvetica-like font; this database can be found on the website. The image `testpage.pbm` was obtained from a screen capture.

In this form, the database can be used for the simplest form of character recognition: *template matching*. In this case, template matching amounts to performing a pixel-by-pixel comparison between the saved glyphs and the input glyphs. The saved ones are classified, so a perfect match implies that the class of the input glyph is the same as that of the saved one.

Consider the input image seen in Figure 9.3. The database contains saved templates for each character, and each of these is compared against the input glyph. Figure 9.3b, for example, is a comparison of the input against the character *A*. Note that the black pixels that the two glyphs have in common are counted in favor of the match, and differences count against. White pixels do not count; if they did, it would introduce a bias in favor of large characters having few black pixels.

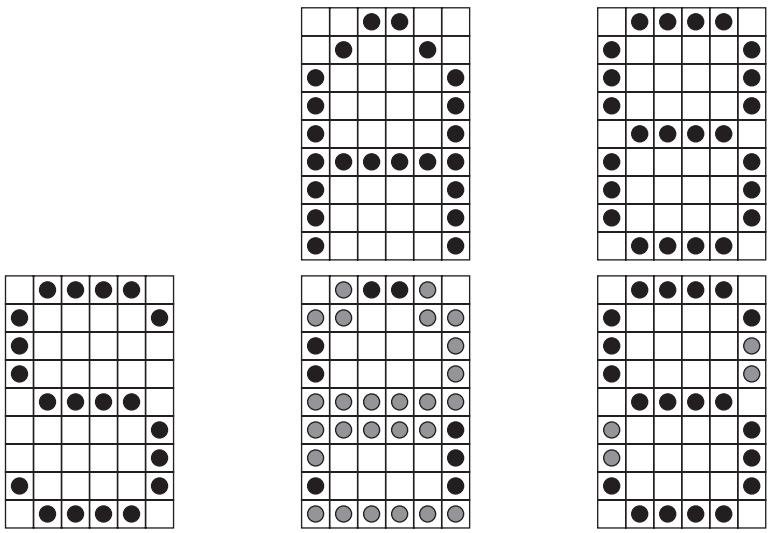


Figure 9.3: Template matching. (a) Third input glyph, extracted from an image containing text. The question is: "What character is this?" (b) The letter A as it appears in the database. The pixels in common with the input character appear below, in black. Pixels that do not match are grey. There are eight pixels in the match. (c) The digit 8 as it appears in the database, and the match with the input. The match is better in this case.

The simplest possible OCR system uses this scheme to recognize characters. The program `ocr1.c` assumes that the image is bilevel, aligned properly, and that the database created by learn exists and is correct. It determines the positions of the lines by using projections, extracts characters from each line

one at a time, and performs a template match against all the characters in the database. The best match corresponds to the correct classification.

The existence of spaces (blanks) is inferred from the character spacing. Any character having a distance greater than the mean plus the width of a space as measured from the training image is assumed to be followed by a space; multiple spaces are found in the same way. The `learn.c` program determined the size of a space by examining the differences in where the various lines began. For example, the second line is indented an extra space relative to the first line; this difference is measured as pixel widths, and averaged over the four instances of indented lines.

When the `ocr1.c` program is run there are a few problems, none terribly serious. The actual position of the glyphs relative to one another is not saved in the database, which means that commas are recognized as single quotes; they are identical except for position. This could easily be repaired by simply noting that commas must appear at the bottom of the line, whereas quotes are at the top. Another problem occurs with double quotes, which are the only character having two horizontally separated black regions. They would be extracted as a pair of single quotes, which could be converted into a double quote with a post-processing stage.

When run using a text image that was captured from the screen of a small workstation `sample.pbm`, the `ocr1.c` program successfully recognized all characters in the sample except for the commas. Of course, the font was the same one on which it was trained.

9.3 OCR on Scanned Images—Segmentation

When using a scanner as a text input device, the problem becomes much more difficult. Most scanners will produce a grey-level image, so thresholding becomes an issue. The resolution of the scanner is finite, so the position of the document when scanned will affect the pixel values over the entire image. This means that a letter will have slightly different pixel values depending on its horizontal and vertical position on the page, and so will have a number of possible templates after thresholding.

The orientation of the image is no longer assured, either. Although it should be close, the page need not be aligned exactly to the horizontal; the `ocr1.c` program assumed a perfect alignment. Finally, and most difficult of all, the characters in the text image may touch each other after thresholding. Two or three characters that are connected by black pixels, such as those in Figure 9.4, will be extracted as a single glyph. The problem is serious, because not only is it not known where to split this glyph to get the three characters, but it may not even by clear that the glyph contains more than one character. Because of the use of the proportional fonts in many documents, in which the spacing

between characters is a function of the character, the width of a multiple glyph does not always exceed that of a single one. For example, the characters *ij* occupy less horizontal space than does the single character *M*.

Figure 9.4: Glyphs extracted from text that uses proportional spacing, and where the characters are too close together to be easily separated. The fact that the characters are connected by black pixels leads to the problem of selecting a location at which to split them.

Finally, the use of a scanner introduces noise to the image. When thresholded, grey-level noise becomes random black pixels, or small black regions. All these issues combine to greatly increase the complexity of the situation.

Let's deal with these problems one at a time. The thresholding issue was dealt with in Chapter 3, and almost any of the methods described there should work reasonably well on a text image. The adaptive algorithm, found in the Chapter 4 program `thrdd.c` is, in addition to being acceptable, fairly quick, and will be used in the examples that follow. Therefore, the first problem to be dealt with should be that of noise reduction.

9.3.1 Noise

If noise is a problem, there are a few ways in which it might be dealt with if it is to be done before thresholding. The first step is to acquire multiple images of the same page. Averaging the grey levels of each pixel across all the samples will give a much better image as far as noise is concerned. Averaging four samples, for example, cuts the noise in half. Of course, this takes longer, and care must be taken not to move the document at all between scans. Another possibility is to acquire multiple samples of the page, threshold them, and use only those pixels that are black in all samples (or a majority vote). This takes even longer.

If it is not possible to acquire multiple samples, then a *median filter* will reduce the noise level. It will, unfortunately, also reduce the contrast of the edges, and might result in the closing of small gaps. A median filter is a pass through all pixels in the image, looking at an $N \times M$ region centered at each pixel. The pixel in the center is replaced by the median value of all the pixels in the region. Not all the pixels must be considered when computing the median; for example, if only a horizontal row of pixels is used, then vertical edges are preserved.

The median filter is slow, requiring not only a pass through the image of the window, but also needing a sort of the pixels values in that window to find

the median; in a sorted array of 100 elements, the median is found at array element 50. While the mean is easier to calculate, the blurring introduced by replacing a pixel with the mean of its neighbors is generally more than can be tolerated [Huang, 1979].

If the noise is to be removed after thresholding, then the problem becomes one of filling small holes in the characters and removing small isolated black regions. Specific noise reduction filters have been designed for use in OCR systems that take advantage of existing knowledge about the characteristics of text images. Some of these are small (3×3) templates that are passed over the image, deleting or setting a pixel whenever a match is encountered.

For larger regions morphology has been used, but a relatively recent method called the kFill filter [O'Gorman, 1992] is very interesting. This method uses a square $k \times k$ pixel window passed over the image multiple times. The outer two rows and columns of the window are called the *neighborhood*, with the center portion called the *core*. The first pass of the window will set all the core pixels to white (the background level) if certain parameters computed from the neighborhood allow this; the second pass will set the core pixels to black, again depending on the neighborhood.

For the sake of explanation, assume that $k = 5$. Since the neighborhood consists of the outside two rows and columns, this means that the core is just the single pixel in the middle of the window. On the first of the two passes of the 5×5 window over the image, we are looking for locations where the core is black. At any such locations, the following values are measured:

1. The total number of white (background) pixels in the neighborhood. This value is n.
2. The number of white corner pixels. This value is r.
3. The number of distinct connected white regions encountered in the neighborhood. This value is c.

If c has any value except 1, this window is left alone and the next one is processed. The reason is that the core pixels may connect two regions, and deleting the core pixels will create two objects where one existed before.

Assuming that $c = 1$, the core pixels are set to white if:

$$\left(n > 3k - \frac{k}{3}\right) \vee \left(\left(n = 3k - \frac{k}{3}\right) \wedge (r = 2)\right) \qquad \text{(EQ 9.1)}$$

Figure 9.5a shows a glyph containing salt-and-pepper noise resulting from thresholding a noisy image. After the first pass, the isolated black pixels are gone (Figure 9.5b), as are some of the boundary pixels. After the next pass the isolated white pixels are gone, too (Figure 9.5c), and processing could stop here. However, the algorithm continues until no changes occur in two

consecutive passes, and there remain a few boundary pixels that satisfy the removal criterion. The final glyph appears in Figure 9.5d.

Figure 9.5: Use of the kFill noise-reduction algorithm. (a) Original noisy glyph. (b) After the first pass with k=5; the isolated black pixels are removed. (c) After the second pass, which removes isolated white regions. (d) Processing continues until no further changes are seen in two consecutive passes.

The program `kfill.c` provides a sample implementation of this technique.

9.3.2 Isolating Individual Glyphs

The connected glyphs in Figure 9.4 illustrate a part of the problem that occurs when multiple glyphs connect to each other, due to noise or undersampling. The problem is quite serious, because a template match will give very poor results in these cases, and a statistical approach (Section 8.1.1) requires isolated glyphs from which to measure features. Ultimately, separating these connected glyphs is essential if a reasonable recognition rate is to be achieved, but there are many ways in which the connections can be formed, and some valid glyphs can appear to be two connected ones. An example of the latter case is the letter *m*, which can be split into two good matches of *r* and *n*. It might well be that a good separation cannot be done as an isolated case, and that contextual information will be necessary for a proper segmentation. Continuing the example, if the letter *m* appears at the end of the word *farm*, then the separation into *rn* would yield the word *farrn*, which would not make sense.

It would be only fair to point out that this is an unsolved problem. There is no algorithm that works in all the cases that might be encountered while scanning any document of a reasonably large size. Still, something must be done, and the most commonly encountered methods use some variation of a vertical projection. Most simple of all is to locate the minima in the vertical projection and segment the image at those places, but this has some unfortunate problems.

Consider the small text image in Figure 9.6. Many of these characters have clear separations, and would be isolated by the process of identifying

connected components. The vertical profile (projection) appears below as a line graph, and solid vertical lines spanning the two shows the positions where the characters should be split. The dashed vertical lines show places where a projection-based method would split a character into two parts, creating an error. These locations occur in places where the projection is a local minimum having a smaller value than the minimum between the *t* and *h* in the word *the*. Thus, any reasonable threshold value used to identify minima would either fail to split the *th* in *the*, or would also split the *n* and the *h* into two parts.

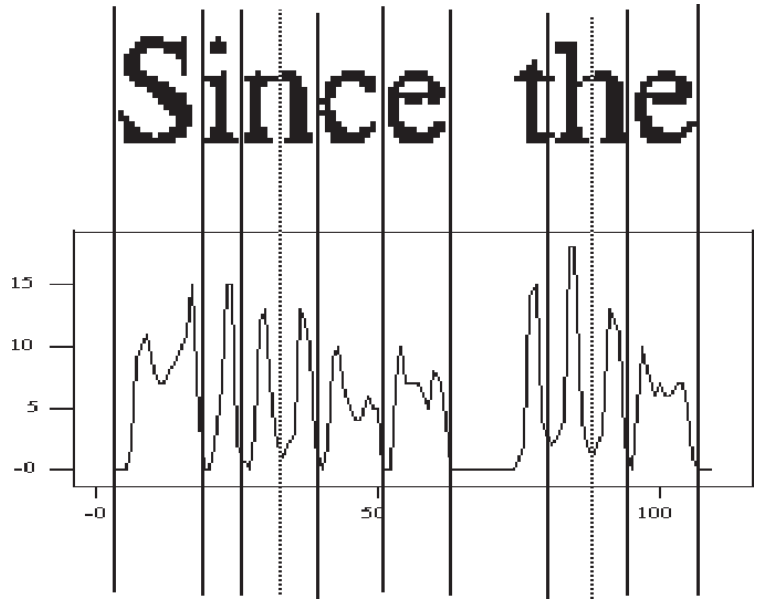


Figure 9.6: Using minima in the profile results in splitting some legitimate glyphs into two parts. Splitting the *n*, for example, could result in seeing *ri* instead.

A related technique computes a *break cost*, which uses two adjacent columns [Tsujimoto, 1991]. This value is defined as the number of black pixels in a column that also have a black neighbor in the same row of the previous columns, as shown in Figure 9.7. Columns in which the break cost is small are candidates for locations at which to split the glyph. The break cost is plotted in Figure 9.7 below the text image and, as before, solid lines are drawn where the segmentation into glyphs would occur — in this case, places where the break cost is zero.

Although the segmentation of the image in the figure is perfect, this is not always the case. The addition or removal of one strategic pixel could alter the situation; for instance, if the *t* in *the* were connected at the bottom by one more pixel, then the *t* and the *h* would not be split.

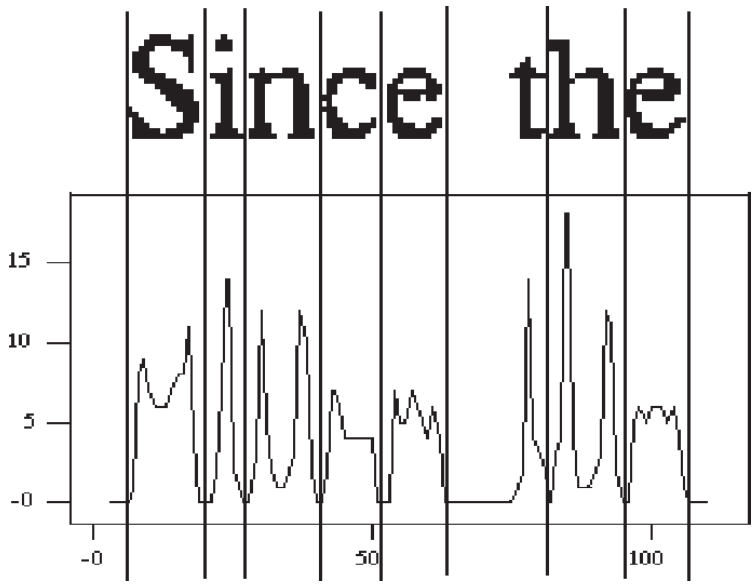


Figure 9.7: The use of break cost for locating isolated characters. The vertical lines are drawn where the break cost is zero. This happens to segment the image perfectly, which is not true in general.

Applied alone, none of the glyph segmentation methods will successfully isolate a large fraction of the connected glyphs. The problem might be fundamentally ambiguous, and so the solution is to use other information. The use of context is possible, as is the optimization of the recognition probabilities for the entire collection of connected components. This latter idea can be implemented at a relatively low level, and so is generally tried first. The use of context generally means checking words to make sure they exist in a dictionary, and correcting the spelling accordingly. This can only be done after the entire set of component glyphs have been extracted and recognized.

A typical approach would work as follows: The connected glyph image would have cut locations identified by one of the previously discussed methods. The pixels between each two cut positions would then be considered to be a glyph, and recognition would be attempted. The set of cut positions for which all glyphs were recognized (and having the highest probability of being correct) would be chosen as the correct segmentation.

Consider the image in Figure 9.8. This consists of five character glyphs, all of which are connected as a single connected region. The cut positions as determined by break costs are shown in the figure as arrows pointing to the position at which to cut; there are six of them, any of which (or any consecutive combination of which) might be an actual character. A possible next step in the segmentation process is to create a *decision tree* containing the possible groupings of the regions in the image. Each contiguous region corresponds

to a node in the tree, as shown in Figure 9.8b. Also associated with each node is a character (which is the classification of the region) and a measure of likelihood. The most likely path through the tree will be selected as the correct segmentation, and the classifications will therefore be immediately available.

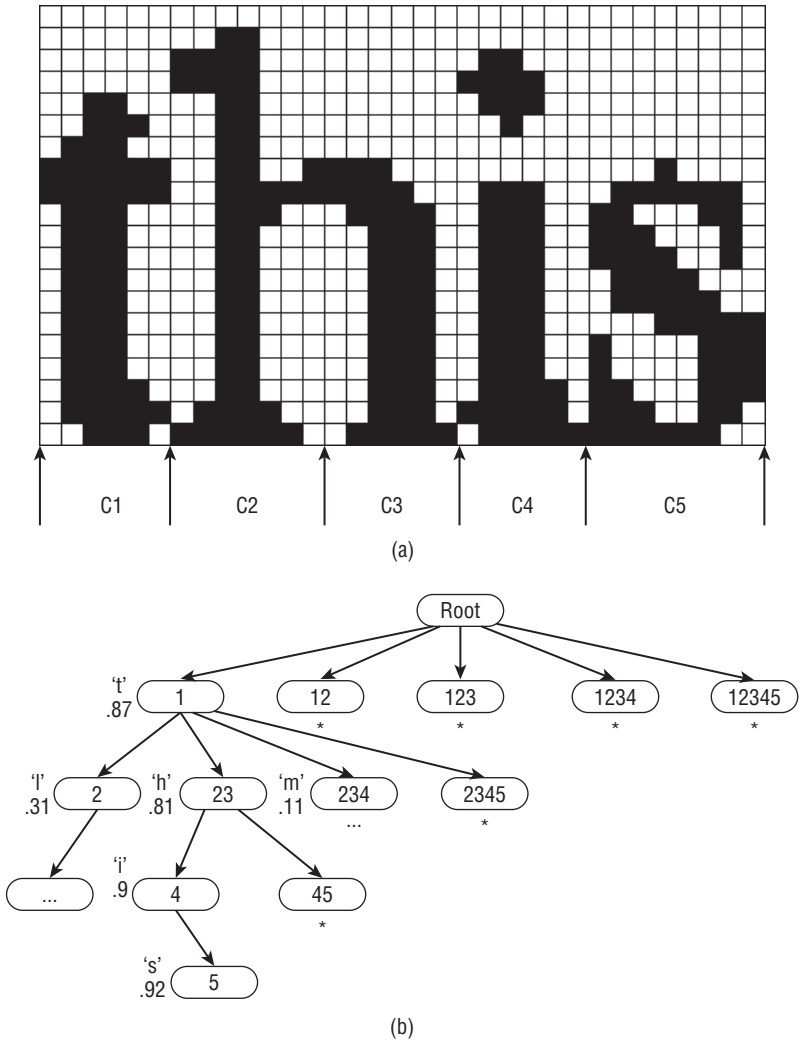


Figure 9.8: A connected glyph and its decision tree. (a) Each of the five regions may themselves be a glyph, or any set of adjacent regions may combine to form a glyph. (b) The decision tree. At the first stage, the first region is determined to be a *t*. The second character could be any of *l, h*, or *m*, with only the path corresponding to this giving a reasonable classification.

When a node (a set of contiguous regions) is able to be accepted (classified with a significant likelihood) the classification and likelihood are stored in the

relevant node. Only then are the child nodes processed; the children are the possible combinations of the remaining nodes (regions). Nodes that give a very poor classification are not evaluated further, at least initially. If it happens that none of the paths through the tree result in a good classification, then the less likely branches can be evaluated.

The tree can be constructed as described, or a recursive decomposition can be devised. The set of all possible recursive calls forms a tree, and the best segmentation will be the only one retained, the others being discarded as soon as a better one is found.

This approach depends for its accuracy on the character recognition method that is applied to the cut regions, but is independent of that algorithm in the sense that any recognition method will result in a segmentation.

9.3.3 Matching Templates

In general, template matching does not work as well on scanned glyphs as it did on the perfect images of the previous section. Noise and the resulting errors caused by thresholding creep into the picture, causing deviations to be registered when matching characters against even correct templates. Taking noise reduction seriously does help, and using a good thresholding algorithm helps, too. Possibly more important is to use a wide selection of templates, though. The standard image used for teaching a font (Figure 9.1) has only two instances of each character, when in fact a dozen may not be too many. Variations can be obtained by scanning the template image many times, saving each set of templates in the same database. More templates can be had by using variations on the thresholds, again saving all the variants.

There are also variations possible in the template-matching procedure. When dealing with perfect glyphs, the number of mismatches was subtracted from the number of matches. It is better to normalize the match measure to account for the number of pixels involved. For example, a normalized match index (NMI) can be found by counting the number of pixels that match (M^+) and those that do not (M^-). The normalized match index is then:

$$NMI = \frac{M^+ - M^-}{M^+ + M^-} \qquad \text{(EQ 9.2)}$$

This value has a minimum of -1 and a maximum of 1, which is convenient for many reasons.

Small variations in the glyphs will produce glyphs of slightly varying sizes. This has the effect of causing a misalignment of the template and the glyph, which can produce a poor match. Therefore it is necessary to find a number of possible matches, each corresponding to a different position of the template over the glyph. A few pixels (perhaps five or six in each dimension) are all that

is needed, but this means between 25 and 36 matching attempts. This slows down the template-matching process dramatically.

Another issue is that of holes. The background pixels that match are not counted, as a general rule, since that would give a strong bias towards matching a large, empty glyph. However, a hole can be considered to be structural. The large set of background pixels in the center of, for instance, a letter *O*, can be used to match the character.

To do this, the holes are marked with a pixel value other than those used for background and object pixels. Then an extra case is inserted into the function that computes the match: A hole pixel in the template should match a hole pixel in a glyph, and (if so) record a match; else record a mismatch. In essence, hole pixels are treated in the same way as object pixels, but are a distinct class and don't match an object pixel.

Given an isolated glyph, the holes can be found in the following way: Any background pixel connected to the bounding box of the glyph is marked with, for example, the value 3. Then any background pixel connected to a 3-valued pixel is also marked with a 3, recursively. When no further pixels are marked or markable, any remaining background pixels within the bounding box belong to a hole. These are then marked, while the 3-valued pixels revert to the background level. The use of 4-connected pixels in the marking process ensures that a thin line will not be ''jumped over,'' resulting in a hole being missed.

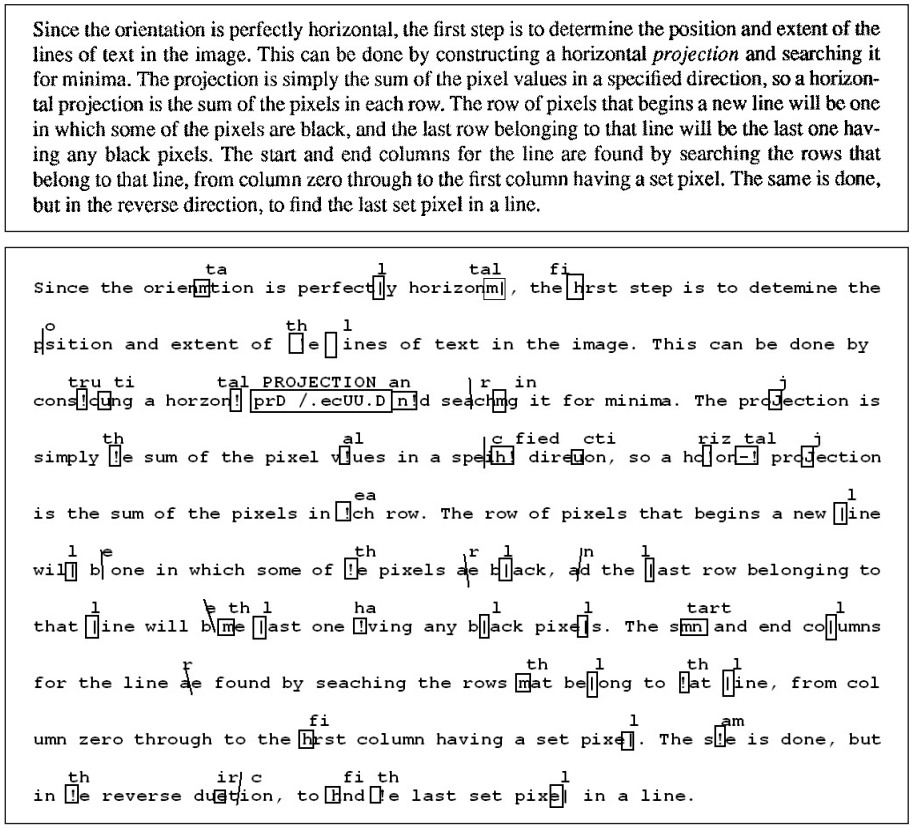
This is the template-matching process used by the program `ocr2.c` and the corresponding learning module, `learn2.c`. The `learn2.c` module reads the standard test image and creates a database of templates. `ocr2.c` reads the database, and then attempts to recognize the characters in an input image by using a template match. The input images correspond to scanned documents. For example, the image file `paged.pbm` is an image of printed 10-point text, scanned at 300 DPI (dots per inch). Of course, the test image must also be 10-point text in the same font, and must have been scanned at the same rate; the corresponding test image is `pagec.pbm`. These images have already been thresholded.

A sample result from `ocr2.c` appears in Figure 9.9. The data used was scanned from a page in this book, and appears in the figure above the text that was extracted, the spaces having been edited. Most of the errors occurred because of a failure to separated joined glyphs, and because the letter *l* was repeatedly mistaken for the special character |.

The errors in the first two lines of output and their causes are as follows:

1. *m* should have been *ta*. Caused by the *t* being joined to the *a*.
2. | should have been *l*. Template matching error.
3. *m* should have been *ta*. Joined glyphs, again.

4. | should have been *l*.
5. *h* should have been *fi*. Joined glyphs.
6. Missing *o*. Joined glyphs
7. *!* should have been *th*. Joined glyphs.
8. | should have been *l*.



Since the orientation is perfectly horizontal, the first step is to determine the position and extent of the lines of text in the image. This can be done by constructing a horizontal *projection* and searching it for minima. The projection is simply the sum of the pixel values in a specified direction, so a horizontal projection is the sum of the pixels in each row. The row of pixels that begins a new line will be one in which some of the pixels are black, and the last row belonging to that line will be the last one having any black pixels. The start and end columns for the line are found by searching the rows that belong to that line, from column zero through to the first column having a set pixel. The same is done, but in the reverse direction, to find the last set pixel in a line.

Figure 9.9: A text image (above) and the text extracted by template matching (below).

There is a pattern in the repetition of errors. The mismatch of | for *l* is pathological, and occurs 14 times in this small image (counted as 14 distinct errors). Certain combinations of letters are repeatedly misclassified; *m* is seen for *ta*, *h* for *fi*, and ! for *th*. This latter error, for example, happens four times in this passage alone. One serious set of errors occurs in the second line of the original text, where the word *projection* appears in italics. Since there are no templates for the italic font, it is not too surprising that the recognizer fails miserably at this point. For this example, the overall success rate (percentage of correct characters) is 86.3%.

Adding code that will attempt to split a glyph if it classifies with a low probability does improve the situation. Deleting the template for | is also a good idea, at least in this case; it causes more errors than its removal would cause, by a large factor. The addition of glyph splitting and deletion of the | improves the recognition, giving a success rate of 88.9% for the test data of Figure 9.9.

Template matching is an intrinsically slow process. One way to speed it up is to initially match only a sample of the pixels; say, one in every four. This can be thought of as a resampling. If the match to the subset of pixels is sufficiently good, then the remainder can be tried, to get the actual match index. If the match is very poor, then there would appear to be no point in pursuing the template involved, and so the match would be aborted. Figure 9.10 shows how this would work if the template were split into four parts. A serious failure at any of the four parts will abort the match. This method works quite well in some cases, and is used in the printed music recognition system (Section 9.7). In this case a speedup of about three times is achieved over the basic template software.

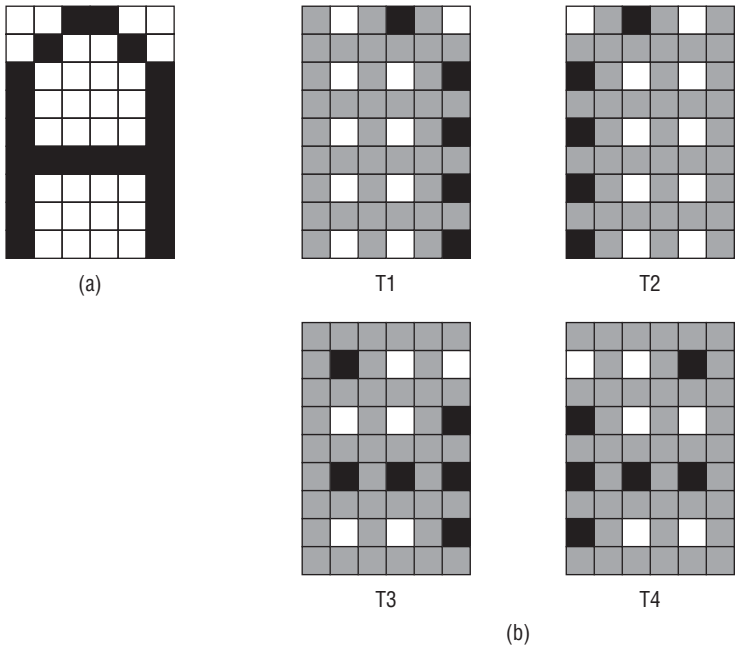


Figure 9.10: Hierarchical template matching. (a) The original template. (b) The four parts of the template used in a match. Each of these has one quarter of the pixels of the template, so early match failures can speed things up by a factor of four. Actually, a factor of about three is attained.

9.3.4 Statistical Recognition

There are two ways to use features to classify objects. In *statistical* approaches, which were discussed extensively in Chapter 8, many features are combined into a large feature vector. A feature is a measurement made on a glyph, and combining them into a vector is a simple way of collating multiple measurements. Because of this, the same object can correspond to a wide variety of feature vectors just through the errors in the measurements. However, these measurements will be clustered in some region of N-space, where N is the dimension of the feature vector. Hence, a statistical recognizer will construct a feature vector from a data object and classify it based on how far (Euclidean distance) it is in N-space from the feature vectors for known objects.

As a contrast, the basic idea behind *structural* pattern recognition is that objects are constructed from smaller components using a set of rules. Characterizing an object in an image is a matter of locating the components, which at the lowest level are features, and constructing some representation for storing the relationships between them. This representation is then checked against known patterns to see if a match can be identified. Structural pattern recognition is, in fact, a sophisticated variation on template matching, one that must match relationships between objects as well as the objects themselves. The problems involved in structural pattern recognition are two: locating the components, and finding a good representation for storing the relationships between the components. Some structural approaches will be discussed in Section 9.4.

A successful statistical classifier depends on an astute selection of the features (measurements, or properties) and their accurate measurement. Of course, very large vectors can be used, but the execution time grows as the size of the vector grows, and at some point the problem becomes intractable. Still, sizes of many hundreds of elements have been used in real systems.

The features themselves should be easy to measure or, again, the execution time of the classifier will become an issue. Fortunately there are many such features. For instance, the ratio of the area of a glyph to its perimeter can characterize shape, although crudely. For a circle, the area is computed by $a = \pi r^2$, and the perimeter by $p = 2\pi r$. The ratio of p^2/a is $4\pi^2 r^2/\pi r^2 = 4\pi$. Hence, the expression:

$$C = \frac{p^2}{4\pi a} \qquad \text{(EQ 9.3)}$$

will be unity, or nearly so, for a circular object, and will increase in value as the shape of the region deviates from a circle (a disk, really). This measure, called *circularity*, can be used as one of the features.

Since the area has already been calculated, another feature that is simple to compute is *rectangularity*, denoted by R. This is simply the ratio of the area of the region to that of its bounding box; as the shape of the region varies from perfectly rectangular, the value of R decreases.

The aspect ratio can be defined as the ratio of the object's height H to its width W. If rectangularity has been calculated, then the bounding box is available, and the aspect ratio A can be found from that.

Other features that can be used include: the number, size, and location of any *holes* (enclosed background regions) in the object; the *convexity*, which is the relative amount that the object differs from a convex object; *moments*, of any order; *shape numbers* [Gonzalez, 1992]; *Euler* number; and a host of others [Parker, 1994].

Some features consist of a collection of values, and are themselves vectors. For example, a glyph can be resampled to a small, constant size. For example, to resample a 12×12 glyph to size of 3×3, simply group the pixels into three groups of four in each direction (Figure 9.11). The value of each of the new pixels is the average of the pixels in each of the nine regions, scaled to a known range so that they can be compared fairly. The nine pixels in the resulting 3×3 image can be stored in consecutive locations in the feature vector. This will be called a *multiple* feature.

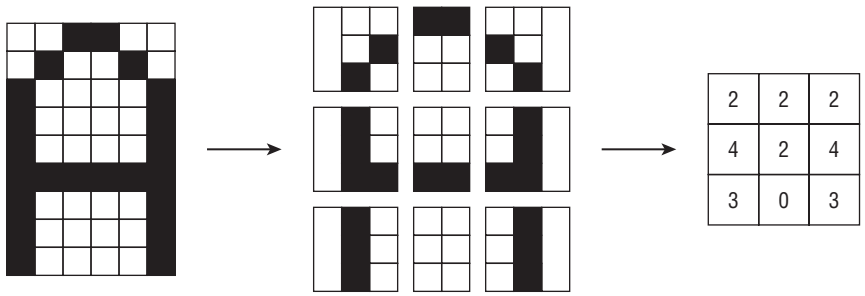


Figure 9.11: Resampling of a glyph to provide a multiple feature. The original is split into three parts, both horizontally and vertically. The number or fraction of black pixels in each of the nine resulting regions contributes to the feature vector.

Other examples of a multiple feature include the *slope histogram*, *profiles* in any direction, and *signatures*. A slope histogram (for example, as in [Wilkinson, 1990]) is the histogram of directions of the lines comprising the boundary of the object. The size of the histogram depends on the bin size; for machine-printed characters, it is unlikely that more than 16 directions will be useful, and eight would be sufficient when used with other features. A profile, as has already been discussed, is the sum of the pixel values in some specified direction. Multiple directions can be used, which is rather like taking a CAT scan of the glyph.

A signature is an interesting concept, and it can be defined in a few different ways. While some define a signature as being the same as a projection, it can also be defined as a one-dimensional representation of the boundary. This is

found by computing the distance from the centroid (center of ''mass'') of the object to the boundary as a function of angles, from zero degrees to 360 in any chosen increment. The resulting set of distances, when properly scaled, can be included in the feature vector. Figure 9.12 illustrates this process. In this example, the signature values are distances indexed by the angle to the boundary pixel. If a distance already exists at a particular angle the longer of the two is used, yielding an outer boundary signature.

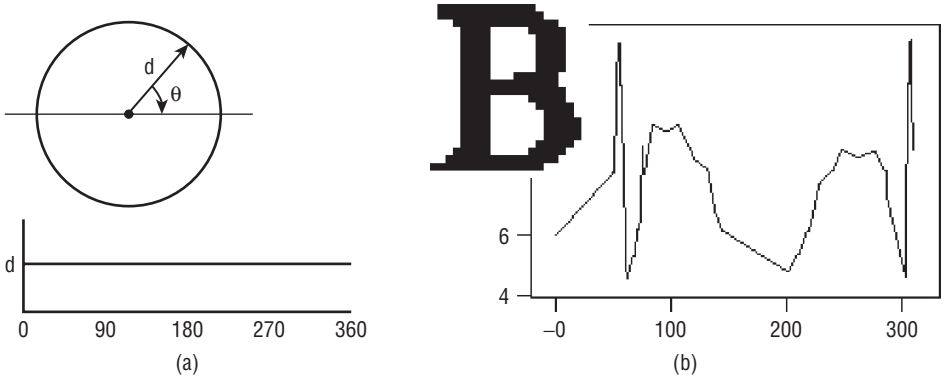


Figure 9.12: One way to compute a signature. (a) For a perfect, real circle. (b) For a sampled glyph.

The signature computed in this latter manner is dependent on rotation and scaling. This means that a signature found for a 12-point character cannot be used as a template for matching a 10-point character. Attempts have been made to normalize the signature to make it scale-independent. This is not as simple as it may seem, although dividing all bins by the variance has been used with some degree of success [Gonzalez, 1992]. The program `sig.c` is one implementation of the procedure.

9.4 OCR on Fax Images—Printed Characters

The use of fax images introduces further complications. Although it is easily possible to place a document on a scanner and collect an image that is within three degrees of the true orientation, fax machines generally have page feeders, and the intrinsic error is greater. Noise is also a greater problem; often the image has been sent thousands of miles over somewhat grubby phone wires, and fax image standards can use lossy compression methods. A better choice of features may be of some help in decreasing the impact of the intrinsic distortions.

9.4.1 Orientation—Skew Detection

When using a scanner, a careful user can place a document so that the lines of text are within about three degrees of the true horizontal. When using a fax or other such device, there is no guarantee of this. The device itself often feeds the paper across a recording device, and the text on the page may or may not be properly aligned in the first place. Thus, one of the first steps in attempting to read a fax image is to estimate the orientation angle of the lines of text; this is called the *skew angle*, and the measurement is called *skew detection*.

Skew detection can be performed in many ways, and the methods described here form only a summary of the possibilities. In the case where an approximate skew angle is known, a horizontal projection can be used. Recall that these were used to identify the line positions of the text in Section 9.3.2. In cases where the skew angle is large, there will be no parts of the projection that correspond to a completely white band between lines. However, if the approximate angle is known, the image can be rotated by that amount; then projections are computed for small angles until a maximum line height and maximum white space between lines is found. The resulting angle is the skew angle, assuming a reasonable noise level.

For any character not having a descender (characters other than *g, j, p, q,* and *y*) the bottom of the characters in each line are colinear. Therefore, if the bounding box of each glyph is found, the center point of the bottom edge of the boxes should be colinear for each line of text.

This suggests the following algorithm [Baird, 1987]:

1. Identify all the connected regions; presume that each represents a character, unless it exceeds some size threshold.
2. Find the bounding box of each region, and locate the center of the bottom edge of the bounding box.
3. For a given angle θ compute the projection of the points obtained in step 2, yielding a one-dimensional projection array $P(\theta)$. Baird uses a bin size corresponding to ⅓ of the size of a six-point character at the sampled resolution. $P_i(q)$ is the value of the ith bin found for angle θ.
4. Maximize the function:

$$A(\theta) = \sum_{i=1}^{n} P_i^2(\theta) \qquad \text{(EQ 9.4)}$$

where n is the number of bins. The angle that gives the maximum is the correct skew angle.

Once again, an estimate of the skew angle is useful, since it will reduce the amount of searching needed to find a maximum. If no estimate is available, an exhaustive search can be used, but this is time-consuming. However, once in the correct neighborhood, a coarse-to-fine search can be used. A nonlinear least-squares method has also been suggested. The reported accuracy of this algorithm can approach 1/30 of a degree, with 1/2 of a degree being typical.

Figure 9.13 provides an example of this method applied to a rotated image; this was done so that the skew angle would be known, and could be compared against the value found by the algorithm. Figure 9.13a is a portion of a text image that has been rotated by 10 degrees; Figure 9.13b shows a 15-degree rotation. After thresholding (Figures 9.13c–d) the first two steps of the Baird algorithm were applied, and the points at the center of the bounding boxes were drawn as black pixels (Figure 9.13e–f). The best-fit straight lines through the black pixels should be at the skew angle of the text.

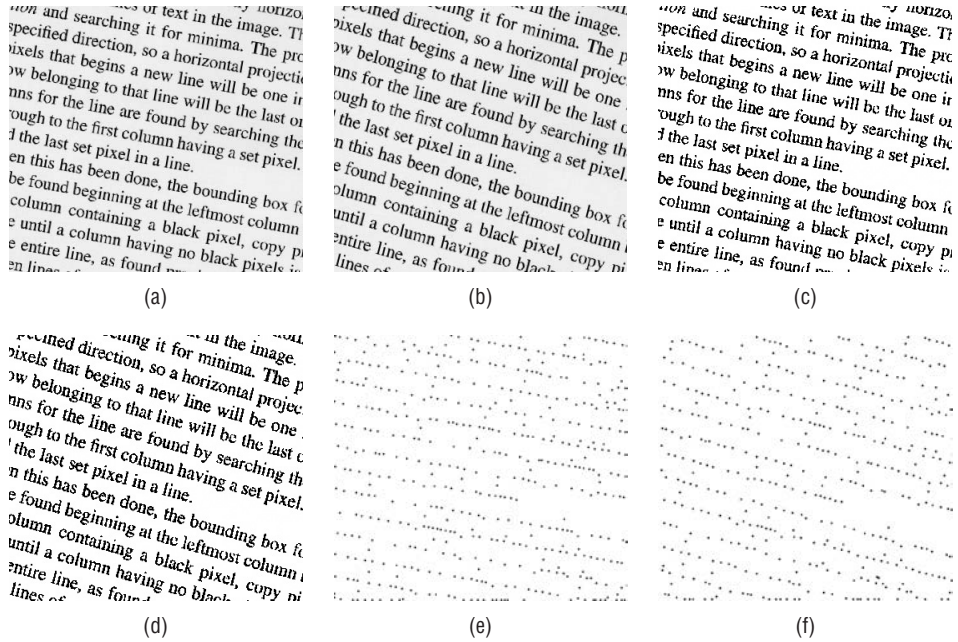


Figure 9.13: The Baird algorithm for skew detection. (a) Document rotated 10 degrees. (b) Document rotated 15 degrees. (c) Thresholded version of (a). (d) Thresholded version of (b). (e–f) Black pixels represent the bottom center of the bounding boxes of the connected regions; these should be characters, or parts of characters. Connecting the dots in the best way should give lines at 10 and 15 degrees.

However, instead of completing the algorithm and determining the best angle by using projections, histograms, and least-squares computations, a

different method for estimating the angle was tried. After all, the least-squares criteria have been used many times already, and we will not learn anything new. Let's try a *Hough transform* for finding the lines.

The Hough transform is a method for detecting straight lines in a raster image [Hough, 1962]. Any black pixel in an image has infinitely many straight lines that *could* pass through it, one for every possible angle. Each of these lines can be characterized by the slope-intercept form of the equation for a line:

$$Y = mX + b \qquad \text{(EQ 9.5)}$$

where the coordinates of the pixel involved are (X,Y), the slope of the line is m, and the intersection of the line with the Y axis occurs at $Y = b$. Now, if this equation is interpreted differently, so that X and Y are constants and m and b are the coordinates, the equation can be reorganized as:

$$b = -Xm + Y \qquad \text{(EQ 9.6)}$$

which is the equation of a line in (m,b) space. Thus, a single point in two-dimensional image space (X,Y) corresponds to a straight line in (m,b) coordinates (Figure 9.14b).

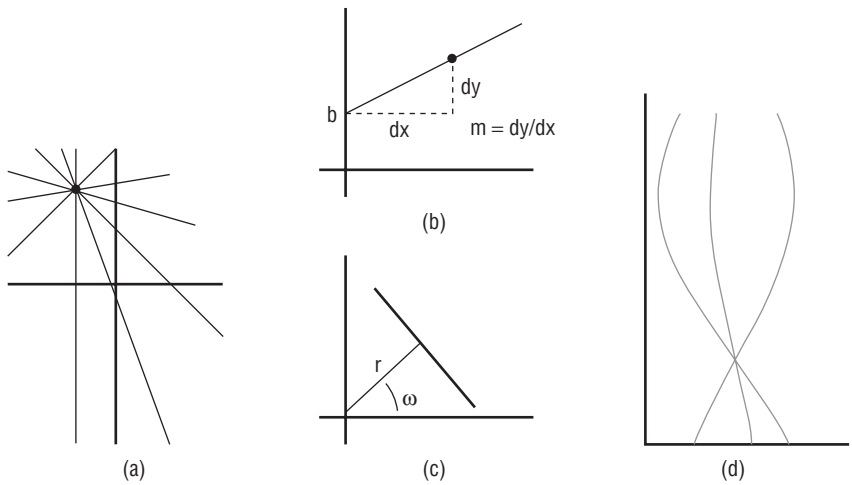


Figure 9.14: The Hough transform. (a) Family of lines through an image pixel. (b) Slope-intercept form of a straight line; one form of Hough space. (c) Normal form, which can represent lines of any angle. (d) (r,w) Hough representation of three colinear points; (2,2), (22,22) and (55,55). Angle should be 45 degrees; the estimate is 44 degrees.

Each pixel in an image corresponds to such a family of lines, expressed in (m,b) space as a straight line itself. What is more, places in (m,b) space, which will now be called *Hough space*, where two lines intersect correspond to colinear points in image space. This is no big deal, because *any* two points

are colinear, but the same is true for multiple intersections. This leads to the following observation:

> *If the* N *straight lines in Hough space that correspond to* N *given pixels in image space intersect at a point, then those* N *pixels reside on the same straight line in image coordinates. The parameters of that line correspond to the Hough coordinates (*m,b*) of the point of intersection.*

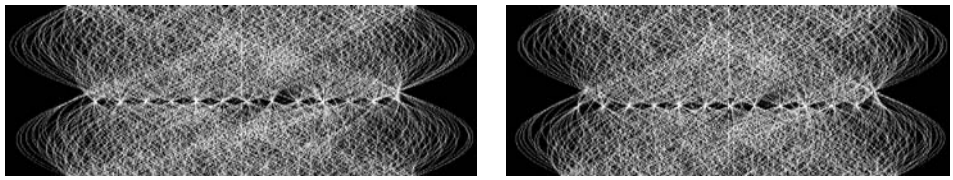
This is the basis for the Hough transform; all pixels are converted into lines in (*m*,*b*) space, and the points of intersection of many lines are noted, and collected into line segments. Because there are in reality infinitely many lines, and infinitely many points on each, the implementation is actually very much like a histogram. A degree of quantization in (*m*,*b*) coordinates is decided upon in advance, and a *Hough image* is created. For each pixel in the original image, the line in Hough space is computed, and each pixel on that line in the Hough image is incremented. After all pixels have been processed, the pixels in the Hough image that have the largest values correspond to the largest number of colinear pixels in the original image.

The slope-intercept line equation has the unfortunate property of being unable to deal with vertical lines: The slope becomes infinite. There are other forms of the equation of a line that do not have this pitfall, including the *normal form*:

$$r = x\cos\omega + y\sin\omega \qquad \text{(EQ 9.7)}$$

where r is the perpendicular distance from the origin to the line, and ω is the angle of that perpendicular line to the x axis. Using this equation, the Hough space coordinates are (r, ω) as shown in Figure 9.14c–d.

The relationship between the Hough transform and skew detection should be clear. The first steps of Baird's skew-detection algorithm give an image with a large collection of sets of colinear pixels. The Hough image of this should have a primary peak at an angle corresponding to the skew angle. Taking the Hough transform of the two examples seen in Figure 9.13 yields skew angles of 10 and 15 degrees respectively, both of which are exactly correct. The Hough images for these examples can be seen in Figure 9.15; the source code for a Hough transform implementation is also shown.



(a) (b)

Figure 9.15: The Hough transform. (a) Hough transform of the image seen in Figure 9.13e. (b) The Hough transform of the image seen in Figure 9.15f. (c) Source code for implementing a Hough transform.

```
void hough (IMAGE x, float *theta)
{
     float **z;
     int center_x, center_y, r, omega, i, j, rmax, tmax;
     double conv;
     double sin(), cos(), sqrt();
     float tmval;

     conv = 3.1415926535/180.0;
     center_x = x->info->nc/2;center_y = x->info->nr/2;
     rmax =
       (int)(sqrt((double)(x->info->nc*x->info->nc +
                           x->info->nr*x->info->nr)) /2.0);

/* Create an image for the Hough space - choose your own sampling */
     z = f2d (180, 2*rmax+1);
     for (r = 0; r < 2 * rmax+1; r++)
        for (omega = 0; omega < 180; omega++)
           z[omega][r] = 0;

     tmax = 0; tmval = 0;
     for (i = 0; i < x->info->nr; i++)
        for (j = 0; j < x->info->nc; j++)
            if (x->data[i][j])
              for (omega = 0; omega < 180; ++omega)
              {
                 r = (i - center_y) * sin((double)(omega*conv))
                    + (j - center_x) * cos((double)(omega*conv));
                 z[omega][rmax+r] += 1;
              }

     for (i=0; i<180; i++)
        for (j=0; j<2*rmax+1; j++)
          if (z[i][j] > tmval)
          {
            tmval = z[i][j];
            tmax = i;
          }
        *theta = tmax;
        free (z[0]); free (z);
}
```

(c)

Figure 9.15: (*continued*)

Another skew-detection method, and one that also requires that the connected regions be identified first, is based on the observation that the spacing between characters on a line is smaller than the spacing between lines [Hashizume, 1986]. This being the case, a line drawn between the centroids of pairs of nearest neighbors should join adjacent characters, and the angle of the line will be near to the skew angle. The histogram of the angles should have a peak at the best estimate of the skew angle.

9.4.2 The Use of Edges

A glyph has, until now, been treated as a connected black region; it is fundamentally a raster entity. On the other hand, it is clear that the character can be recognized using only the edge information. The basic shape and topological properties of the character are present in an edge representation, and there are fewer pixels to be considered, so processing times may be shorter. The problem of locating the edges was discussed in Chapter 1, leaving the problem of *edge linking* and *classification* using edges to be dealt with here.

Edge linking is the process of collecting the pixels into line segments. This has also been called *vectorization*, when used in the context of creating a line drawing from a raster image of an engineering diagram or a map. We begin with either a set of edge pixels, found by applying an edge detector to a grey-level image, or the object boundary, obtained from the bilevel glyph. Each pixel is presumed to belong to a straight-line segment, and so adjacent pixels are added to a set until some linearity constraint is violated. When a set is complete, the line segment is saved as its endpoints, and the pixels in the set are removed from the image; the next line is extracted in the same way, and so on until no pixels remain.

The program `vect.c` is a simple yet quite effective boundary vectorizer that can be used on characters. It works in the following way:

1. The object boundary is identified, and all nonboundary object pixels are set to the background level.
2. Starting at any pixel on the outside boundary, the *chain code* [Freeman, 1961; Parker, 1993] of the outline is found; then a list of pixels comprising the outline is created.
3. Starting at the first pixel in the list created in 2 above, add the next pixel in the list to a set that will be the set of pixels belonging to the next line in the boundary.
4. Check the distance between all pixels in the set created in 3 above and the real straight line between the first and last pixels in the set.
5. If the distance found is less than some threshold D, then continue from step 3, using the next pixel in the list.
6. If the distance is greater than D, then stop adding pixels to the current line. Omit the first and last pixel in the set as the segment endpoints, and let the next pixel in the list be the new starting point.
7. Resume the process from step 3.

This is a classical vectorization strategy, and succeeds for smaller images like glyphs but may be less than useful for large images, such as maps and line drawings.

An alternative to this process involves finding the longest straight line that can be found that passes through black pixels only. This line is saved, the component pixels are removed, and the process is repeated. One problem with this latter method is that the lines found need not form a connected sequence, and some post-processing work is required to reconnect the segments into a sensible representation of the glyph.

Figure 9.16 shows the vectorized version of a glyph, in this case a *B* (first seen in Figure 9.5), and shows the effect of varying the distance threshold. The advantages of a vectorized outline are many, including that the length and the orientation of the line segments are now easily and accurately found, and a rotation can be done precisely, so that the glyph orientation can be repaired.

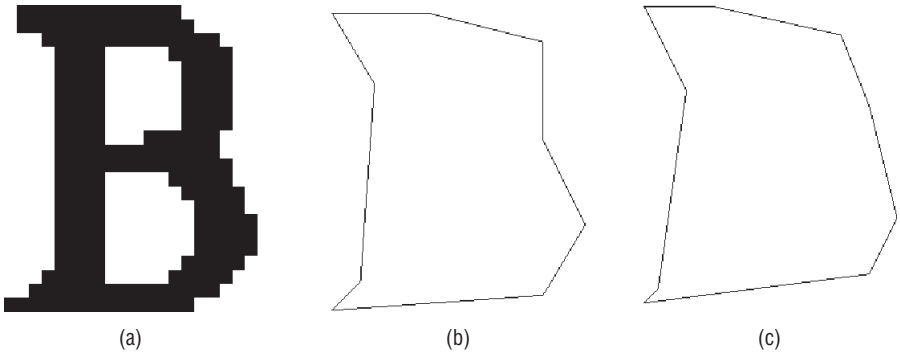


Figure 9.16: Vectorizing a glyph boundary. (a) Original glyph. (b) Vectorization of outer boundary with an error threshold of 1.0. (c) The same vectorization, but with a threshold of 2.0.

Once the boundary of an input glyph is available in vector form, there are a few ways that it might be used in classification. Edge matching is a tricky technique that attempts to identify edges in the input image that correspond to edges in each of a set of model images. The models are classified, so the best match will provide a classification. The vectorized edges are converted into the form (X_c, Y_c, L, θ), where (X_c, Y_c) is the center of the line, L is its length, and θ is the angle the line segment makes with the x axis. If properly scaled, a distance measure can be used to match the lines in the image with the lines in the model. However, inaccuracies creep in since small errors in the outline can be reflected in the vectors extracted, and the coordinate system is usually based on a bounding box (which can also vary slightly).

The *slope histogram*, as mentioned previously, can also be useful, and is more accurately generated from a vector image than from a raster one. We start by quantizing the possible angles of line segments in the image; then we simply create a frequency histogram of the angles that actually appear in a specific

glyph; this defines the slope histogram. The frequency with which an angle occurs will be the sum of the lengths of the line segments having that angle.

The degree of quantization will dictate the number of bins needed in the histogram, and will also be reflected in the accuracy with which a match can be made. It seems likely that no more than between 8 and 16 different angles are required in the case of character recognition.

The example in Figure 9.17 uses eight different angles, in 22.5 degree increments from 0 to 180 degrees. The histograms computed for the samples in the figure are:

B: (0.3815, 0.0, 0.1439, 0.0, 0.3307, 0.0, 0.1439, 0.0)

C: (0.2733, 0.0, 0.26, 0.0, 0.2050, 0.0, 0.2609, 0.0)

D1: (0.3409, 0.0, 0.1875, 0.0, 0.2841, 0.0, 0.1875, 0.0)

D2: (0.3118, 0.0, 0.2004, 0.0, 0.2740, 0.0, 0.2138, 0.0)

D3: (0.3714, 0.0, 0.1751, 0.0, 0.2785, 0.0, 0.1751, 0.0)

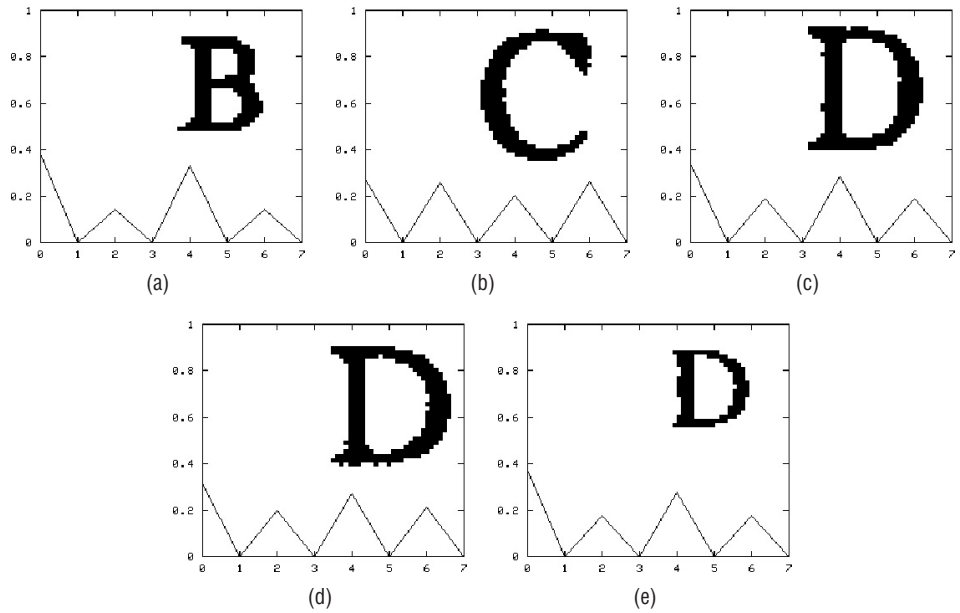


Figure 9.17: Slope histograms for a set of sample glyphs. (a) The B glyph, seen previously (Figure 9.16a, for example). (b) A C glyph. (c) The D1 glyph. (d) D2, a D the same size as D1. (e) D3, a D smaller than D1 and D2. All the D histograms are closer to each other (Euclidean distance) than they are to the other glyphs.

Written as they are above, these histograms are feature vectors, or components of a larger feature vector. This set of characters was selected because, on the face of it, they appear to contain similar sets of directions (although in

different places). The *C* was selected for contrast. When the histograms, which have been normalized by dividing by the total length of all the vectors, are treated as vectors, the smallest distance between any of the *D* histograms is smaller than the distance between a *D* glyph and either of the others. There is one case (D3) where the *B* is closer to D3 than is D2, but the glyph would still be classified correctly. The program `slhist.c` provides a sample implementation.

The matrix of distances for this small case is:

	B	C	D1	D2	D3
B	0.00000	0.23383	0.08730	0.12709	0.06909
C	0.23383	0.00000	0.14653	0.10961	0.17203
D1	0.08730	0.14653	0.00000	0.04251	0.03562
D2	0.12709	0.10961	0.04251	0.00000	0.07557
D3	0.06909	0.17203	0.03562	0.07557	0.00000

One last suggestion is a *vector template* style of match; this involves reducing the shortest vector to one pixel in length, and then scaling the rest of the vectors by the same factor. The result is replotted in a small raster, and a standard template match is performed against templates found in the same way from training data. If the match value is worse than some value *X*, we rescale again using the next shortest line, and so on.

The program `learn3.c` will read the standard test image and create a database of feature vectors. It can be modified to calculate any set of features desired, and store these in a file. `ocr3.c` will read this database, then read a text image and attempt to match the feature vectors from the image against those in the database, performing a minimum distance classification.

9.5 Handprinted Characters

The major problem encountered when attempting to classify handprinted characters is their inconsistency. It is not possible for a human to print a character in exactly the same way twice. Many matching algorithms, especially those using templates, depend on consistency in order to function, and the further apart the template is from the input glyph, the less likely the correct match is to be located. What is needed are methods that use characterizations of general shape and structure, rather than pixel-level features (which are more likely to vary from glyph to glyph).

In this section a number of methods for classifying images as handprinted digits will be considered, both statistical and structural. In addition to providing an exposure to a diverse collection of methods, it is also possible that all

the algorithms can be applied to the same glyph. The result of using many different techniques is an increased confidence in the result in the cases where they agree, and an increased ability to determine the possibility of error in cases where they do not agree. It is generally considered that an algorithm that can indicate that it cannot classify a glyph is superior to one that performs the classification in error. The goal is to minimize the error rate, even if this increases the number of unclassified characters. An inability to classify is termed a *rejection*, and is not normally counted as an error.

Four methods for handprinted-digit recognition will be examined in this section. The test data, which cannot be included on the website for reasons of copyright, consists of two sets of 1000 digits each, in glyph form. One set was used for training the recognition algorithms, and the other was used for assessing their performance. One should not test a recognizer using the same data on which it was trained.

One way to examine the behavior of a symbol-recognition program is to use the *confusion matrix*. This is a two-dimensional array of classifications; the columns represent the classification of the symbol by the algorithm under scrutiny, and the rows represent the actual value of the symbol.

For example, the following confusion matrix was computed for a program that would recognize the four letters *A*, *B*, *C*, and *D*:

	A	B	C	D
A	96	2	0	2
B	0	98	0	2
C	0	0	100	0
D	0	1	0	99

There were 100 test glyphs of each type used in this example; note that each row sums to 100. This will be true unless there are rejected (unclassifiable) glyphs in the data set. The values in the first row represent the number of classifications of each type for the glyphs that were actually *A*. Most of the *A* glyphs were in fact classified as *A*, but some were not. In the failed cases, we can see what, and how severe, the confusion was.

The columns represent the data as classified. For example, in the last column we can see that for the glyphs that were classified as *D*, 99 of them were correct, and four were in error. There is a lot of useful information in a confusion matrix, as will be seen in later sections.

9.5.1 Properties of the Character Outline

In a collection of interesting articles, Shridhar et al. [Shridhar, 1986; Shridhar, 1987; Kimura, 1991] describe a collection of topological features that can be

used to classify handprinted numerals. Most of these features are properties of the outline, or *profile*, of the numeral. For instance, a digit *8* might be described as having a smooth profile on both the left and the right sides, and as having the width at a minimum in the center region. Not all handprinted *8* digits would be recognized by this description, and certainly some other digits might also have this description; the idea is to provide a sufficient number of descriptions for each digit that a high recognition rate can be achieved.

Figure 9.18 shows how the left and right profiles are defined and calculated for a sample digit *9*. After the digit is isolated and thresholded, the number of background pixels between the left side of the character's bounding box and the first black pixel is counted and saved for each row in the bounding box. This gives a sampled version of the left profile (LP), which is then scaled to a standard size; in this case, 52 pixels. A similar process gives the right profile (RP); the difference is that the last black pixel on each row is saved.

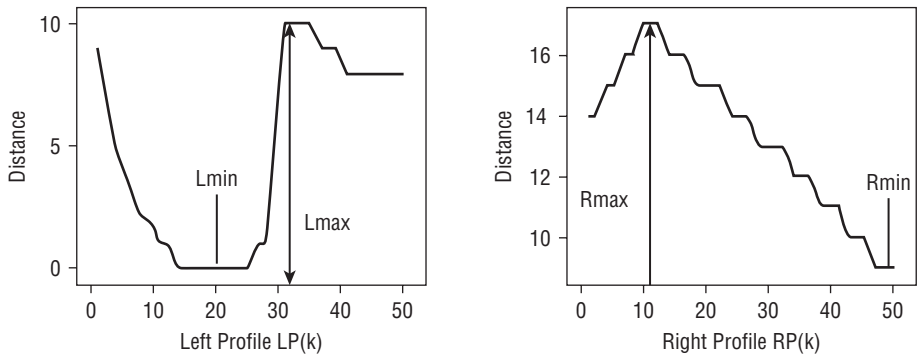


Figure 9.18: Simple properties of the Left and Right profiles. These are the profiles for a 9 glyph.

Having the profiles, the next step is to measure some of their properties. For example, one important property is the location of the extrema: $L_{\min}$ is the location of the minimum value on the left profile, and $L_{\max}$ is the location of the maximum value. There are similar properties $R_{\min}$ and $R_{\max}$ on the right profile. For a *9* digit, it would be expected that $L_{\min}$ would occur in the top half of the character, as shown in Figure 9.1, but for a *6*, $L_{\min}$ would be in the lower half. Using these measures alone can often distinguish between some of the digits.

At this stage some other simple measures can also be useful, such as:

- $W(k)$ = width at position k = $RP(k) - LP(k)$.
- $W_{\max}$, the maximum width of the digit; this is $W(k)$ at some point k where $RP(k) - LP(k)$ is a maximum.
- R, the ratio of height to maximum width.

The use of *R* alone can identify a large majority of the *1* digits; a *1* has a very large *R* value compared with all other digits.

Now another set of features can be defined on the left and right profiles based on first differences. Let LDIF(k) = LP(k) − LP(k − 1) and RDIF(k) = RP(k) − RP(k − 1). Large values of LDIF imply a large change in the profile at the specified point, and this can assist in classification. For example, the change in the profile of the *9* in Figure 9.18, near position 30, would give a large value of LDIF at this point, and that fact is characteristic of the digits *3* and *9*. The locations of the actual peaks in LDIF and RDIF and their values happen to be quite important in characterizing numerals, and the peaks are located using a range rather than a single position. Thus, a digit *5* would have the RDIF peak near the top of the digit, and the peak would have a relatively large value. This set of features is not comprehensive. In all, 48 features are used and others could be defined. The features used in the recognizer described here [Shridhar, 1986] are listed in Figure 9.19.

In the training phase, all 48 features are computed for each sample numeral and a feature vector is created in each case. The features are binary, being either TRUE or FALSE; for example, feature number 43 is TRUE if the width of the character at row 20 is greater than or equal to the width at row 40 (W(20) >= W(40)). Then all the resulting bit strings for each digit are searched for common elements, and the features in common for each digit class are stored in a library. Matching is performed by extracting the profiles of the input image and measuring and saving the bit string (feature vector) that results. This string is matched against the common elements of the templates; this is obviously very fast since only bit operations are involved. A perfect match of a library bit string against an input string results in the corresponding digit class being assigned to the input digit.

The bit strings identified for the sample data (1000 digits) used for this purpose are:

```
12345678901234567890123456789012345678901234567890123456789012345678
*******0**0**********1******0**0******0****1*1**0
****************0******1*********************1*1
******************0********1***********0****0**2
0****1*******0*****1*1****0****000*******0*****13
**********0********0**********0***00*******10*4
0*0**********0*****0*******************1******15
1******0**********************0******0*1**1***16
*****1*1*****1*******1****0****0*0*************7
***00***000*0**0***1***********0*****0**0*****18
**0*0***********0****1****0****00********0*1*****9
```

where * represents a ''don't care'' situation. The results from this method are, again, only acceptable at this stage, with an overall rate of 94.2%. The recognition rates for our samples are outlined in Table 9.1.

Feature #	Feature calculation
1	$L_{peak} < 10$; $2 <= R_2 <= 50$
2	$L_{peak} < 5$; $2 <= R_2 <= 10$
3	$L_{peak} > 5$; $2 <= R_2 <= 15$
4	$L_{peak} > 10$; $2 <= R_2 <= 15$
5	$L_{peak} > 10$; $2 <= R_2 <= 20$
6	$L_{peak} > 5$; $2 <= R_2 <= 25$
7	$L_{peak} > 5$; $5 <= R_2 <= 15$
8	$L_{peak} > 5$; $5 <= R_2 <= 35$
9	$L_{peak} > 10$; $5 <= R_2 <= 40$
10	$L_{peak} > 10$; $10 <= R_2 <= 30$
11	$L_{peak} > 10$; $15 <= R_2 <= 40$
12	$L_{peak} < 5$; $25 <= R_2 <= 50$
13	$L_{peak} > 10$; $30 <= R_2 <= 50$
14	$L_{peak} < 5$; $30 <= R_2 <= 50$
15	$L_{peak} < 5$; $35 <= R_2 <= 50$
16	$L_{peak} > 10$; $35 <= R_2 <= 50$
17	$L_{peak} > 5$; $40 <= R_2 <= 50$
18	$R_{peak} > 10$; $2 <= R_2 <= 50$
19	$R_{peak} > 10$; $2 <= R_2 <= 15$
20	$R_{peak} < 10$; $2 <= R_2 <= 30$
21	$R_{peak} < 5$; $2 <= R_2 <= 45$
22	$R_{peak} < 10$; $25 <= R_2 <= 45$
23	$R_{peak} > 10$; $25 <= R_2 <= 50$
24	$R_{peak} < 5$; $25 <= R_2 <= 50$
25	$R_{peak} > 10$; $30 <= R_2 <= 50$
26	$R_{peak} > 5$; $35 <= R_2 <= 50$
27	$R_{peak} > 10$; $35 <= R_2 <= 50$
28	$R_{peak} > 5$; $40 <= R_2 <= 50$
29	$R_{min}(1 <= R_1 <= 30)$ is less than $R_{max2}(R_{min} <= R_1 <= 30)$ and greater than $R_{max1}(1 <= R_1 <= R_{min})$
30	$R_{min}(10 <= R_1 <= 40)$ is less than $R_{max2}(R_{min} <= R_1 <= 40)$ and greater than $R_{max1}(1 <= R_1 <= R_{min})$
31	$RP(R_{min})=RP(R_{max})$ where R_{min} in the range $5 <= R_1 <= 25$, and R_{max} in $1 <= R_1 <= R_{min}$
32	$RP(R_{min})=RP(R_{max})$, R_{min} in the range $5 <= R_1 <= 25$, and R_{max} in the range $1 <= R_1 <= R_{min}$
33	$RP(R_{min})=RP(R_{max})$, R_{min} in $5 <= R_1 <= 25$; R_{max} in $R_{min} <= R_1 <= 40$
34	$L_{max} < L_{min}$; L_{max} in $1 <= R_2 <= 30$; L_{min} in $1 <= R_2 <= L_{max}$
35	$L_{max} < L_{min}$; L_{max} in $10 <= R_2 <= 30$; L_{min} in $10 <= R_2 <= L_{max}$
36	$L_{max} < L_{min}$; L_{max} in $10 <= R_1 <= 30$; L_{min} in $10 <= R_1 <= L_{max}$
37	$L_{max} < L_{min}$; L_{max} and L_{min} in range $15 <= R_1 <= 45$
38	$L_{max} < L_{min}$; L_{max} and L_{min} in range $20 <= R_1 <= 50$
39	$L_{max} < L_{min}$; L_{max} and L_{min} in range $20 <= R_1 <= 50$
40	$R_{min} < R_{max}$; R_{min} in $1 <= R_1 <= 30$ and R_{max} in $1 <= R_1 <= R_{min}$
41	$R_{min} < R_{max}$; R_{min}, R_{max} in range $20 <= R_1 <= 35$
42	$R_{min} < R_{max}$; R_{min}, R_{max} in range $35 <= R_1 <= 50$
43	$W(20) >= W(40)$
44	$W(25) >= W(10)$
45	$W(25) >= W(40)$
46	$W(25) >= W(45)$
47	Ratio > 2.5
48	$W_{min} < W_{max1}$ and $W_{min} < W_{max2}$ where $W_{min} = W(L_{min})$ between 10 and 40; W_{max1} is max of W between 1 and L_{min}; W_{max2} is max of W from L_{min} to 50.

Figure 9.19: The 48 profile features used to recognize digits.

Table 9.1: Object Outline Method — Recognition Rates

	0	1	2	3	4	5	6	7	8	9
% RIGHT	94	95	96	100	95	100	84	94	90	94
% ERROR	1	4	1	0	5	0	10	5	4	6
% REJECT	5	1	3	0	0	0	6	1	6	0

The recognition rates could be improved by using multiple feature sets for each digit. However, identifying optimal feature sets automatically is a computationally hard problem. One other option would be to specify the features in subsets: For example, a *6* may be classified by the fact that one of the features {2,6,7,21} is true, one of {5,43} is true, and all but one of {1,32,33,41} are true. Again, the number of possible combinations makes identification of optimal feature groupings a very difficult task.

Since an optimal template or set of templates is hard to find, one other possibility is to use a set of 1000 digits as training data and build a library or database of bit strings having known classifications. These are used to identify unknown glyphs using the standard statistical method: the bit string in the database that is closest, in a Euclidean sense, to the bit string obtained from the unknown glyph provides the classification.

The purely statistical method above does give a slightly higher recognition rate at 94.7%, but takes significantly longer to execute. In addition, some digits are recognized very well (100% of the *2*, *3*, and *5* samples) whereas others are quite poor (86% of *7*s). The program `recp.c` (on the website) is an implementation of this algorithm. It uses the file `prof.db` as the database of bit strings, obtained by processing the test digits.

9.5.2 Convex Deficiencies

A digitized character image consists of pixels, usually black on a background of white. Structural character-recognition techniques attempt to collect the black, or object, pixels into sets that represent a feature in a model of an object to be recognized. Features may be lines or curves, for example. Then an effort is made to determine the relationships between the features and to compare these against the relationships in the model, hoping to find a match. Statistical techniques involve the determination of statistical relationships between properties of object pixels. For example, the character image could be divided into nine equal areas and the average grey level or number of set pixels in each area could be measured. This gives nine values that can be stored in a feature vector and matched against existing template feature vectors by using a distance function.

There are numerous problems with existing methods, the basic one being that none of them work well enough in all situations. This suggests a slightly different approach — why not use the more numerous background pixels as a basis for classification? Noise, in the structural sense, ought to have less influence, but similar methods to the ones applied to object pixels should still yield results.

Although most character-recognition systems are concerned with the pixels belonging to the characters themselves, there are good arguments to be made for analyzing the size, shape, and position of the background regions surrounding the character image. Certainly the number and position of the *holes* has been used; for example, an *8* has two holes, one in the upper part and one in the lower part of the character image, and a *0* has one in the middle. However, there are other such features that might be used to classify images; for example, a numeral *2* has a left-facing concave region in the top half of the image and a right-facing one in the bottom half. A more complete analysis of these *convex deficiencies* may permit the development of a classification scheme based on the background regions.

For use with digital character images, a relatively crude but effective scheme has been developed. From each background pixel in an input image an attempt is made to draw a line in each of the four major directions (up, down, left, and right). If at least three of these lines encounter an object pixel, then the original pixel is labelled with the direction in which an object pixel was *not* encountered — called the *open* direction. If none of the four directions are open then the pixel concerned is part of a hole, and is labelled with a zero. Figure 9.20 shows the direction labels, and illustrates the process of locating and labelling the convex deficiencies.

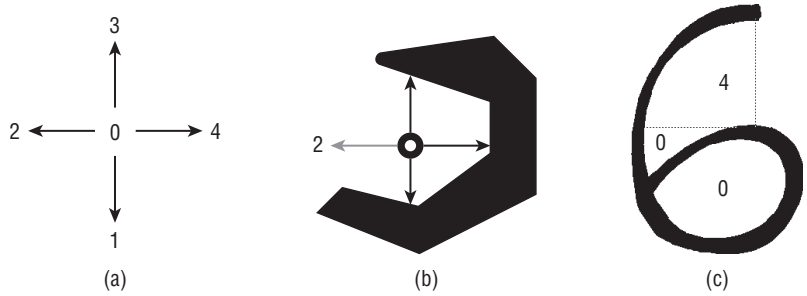


Figure 9.20: Locating convex deficiencies. (a) Codes for the directions. (b) Background pixels are tested to find the open directions (this one is 2). (c) Regions found for a 6 — one of the holes (connected to the 4 region) is not real, and will be connected to another region.

Some of the holes identified in this way are not holes at all, but are part of a convoluted region in the image. A real hole will have boundary pixels that all belong to the object; if this is not true, then the hole is false, and it is converted into a region labeled by its most common nonobject neighbor. Figure 9.21 shows the C code needed to locate and classify the basic convex deficiencies in the raster image.

```
int czones (struct image *x)
{
     int i,j,k,left,right,up,down;
     struct image *z=0;

     copy (x, &z, &k);
     for (i=0; i<z->info->nr; i++)
         for (j=0; j<z->info->nc; j++)
         {
             if (x->data[i][j] == 0) z->data[i][j] = 255;
             else
             {
                 up = 0; down = 0; left = 0; right = 0;

                 for (k=i-1; k>=0; k--) /* Check upwards */
                     if (x->data[k][j] == 0)
                     {
                         up = 1; break;
                     }
                 for (k=i+1; k<x->info->nr; k++) /* Check down */
                     if (x->data[k][j] == 0)
                     {
                         down = 1; break;
                     }
                 for (k=j; k>=0; k--) /* Check left */
                     if (x->data[i][k] == 0)
                     {
                         left = 1; break;
                     }
                 for (k=j+1; k<x->info->nc; k++) /* Check right */
                     if (x->data[i][k] == 0)
                     {
                         right = 1; break;
                     }

/* Consolidate these directions */
                 if (left && up && right && down)
                   z->data[i][j] = 0;
                 else if (left && up && right)
                   z->data[i][j] = 1; /* Open down */
                 else if (up && right && down)
                   z->data[i][j] = 2; /* Open left */
                 else if (right && down && left)
                   z->data[i][j] = 3; /* Up */
                 else if (down && left && up)
                   z->data[i][j] = 4;
                 else z->data[i][j] = 255;
             }
         }

}
```

Figure 9.21: Sample C code for finding and classifying convex deficiencies.

After all the regions are labelled, they are counted and measured. Very small regions, relative to the number of pixels in the object, are ignored, and the largest four regions are used to classify the image. Sometimes a relatively simple relationship exists. For example, 99% of all zeros can be identified by the large central hole and lack of other convex deficiencies. The next more complex scheme uses relative positions of the regions; for example, an *8* has two holes, one above the other, a left-facing region on the left of the two holes, and a right-facing region on the right of the holes. The most complex schemes require shape information in addition to size and position. As an example, some *7* digits and some *3* digits both have a large left-facing region on the left side of the image. However, this region is convex for the *7* but nonconvex for the 3. This fact can be used to discriminate between some *7*s and *3*s.

An important aspect of the method involves not being too specific about the sizes of the regions. The algorithm discards as unimportant any region that has an area less than 10% of the object. Then regions are classified as large or small. In addition, the regions are sorted according to size and only the largest, whatever the actual area, are used. For example, when looking for a 2, only the largest two regions are examined: one is expected to be a left-facing region at the top, and the other is expected to be a right-facing region at the bottom. Position, too, is intentionally classified in only a crude manner as top, bottom, left, or right, with a special flag set for those regions that are sufficiently close to the center (either horizontally or vertically or both). This provides an overall description of the background geometry that provides enough information to classify the image according to digit type, but is not overly affected by the usual variations in line thickness, orientation, size, and shape found in handprinted character images.

Table 9.2 provides a sample set of digit descriptions in terms of convex deficiencies. It is not complete; sometimes a dozen different descriptions are needed for a single digit. Still, it should serve to give the flavor of the kind of description that will work.

The recognition rate achieved using this method is acceptable, averaging about 94%. Table 9.3 details the recognition rate for convex deficiencies.

This approach has the advantage of not requiring a thinning step [Holt, 1987; Zhang, 1984], although smoothing the outline does help, and a morphological closing step might improve the rates for *8*, *6*, and *9*.

The 94% recognition figure could be improved with the addition of more feature sets, or sets that discriminated more thoroughly. Once the obvious features are included, what remains is essentially a matter of common sense and trial-and-error to find the features that discriminate the best.

Table 9.2: Convex Deficiencies Digit Descriptions

DIGIT	OPEN DIRECTION	SIZE	LOCATION	SHAPE
0	0	middle	center	convex
1	None	—	—	—
2	2	middle	upper left	—
	4	middle	lower right	—
3	2	large	left-of	nonconvex
	4	small	right	—
4	3	middle	upper	—
5	4	middle	upper right	—
	2	middle	lower left	—
6	4	middle	above	—
	0	middle	lower	—
7	2	middle	left	—
8	0	middle	upper	—
	0	middle	lower	—
	2	small	left	—
	4	small	right	—
9	0	middle	not-center, above	—
	4	middle	lower	—

Table 9.3: Recognition Rate — Convex Deficiencies

0	1	2	3	4	5	6	7	8	9
99%	94%	98%	96%	94%	90%	90%	93%	95%	92%

9.5.3 Vector Templates

Template-matching techniques in many forms have been applied to the problem of recognizing handprinted digits using a computer. The basic idea is that each digit has a particular shape that can be captured in a small set of

models, usually stored as raster images. An incoming (unknown) digit, also in raster form, is compared against each template, and the one that matches most closely is selected as belonging to the same digit class as the unknown. The system that performs template matching can be ''taught'' new forms by simply adding new templates to its sets. This is often done when an incoming digit cannot be identified well enough; a human classifies it, and the unknown image can be added as a new template if desired.

Once the learning phase is complete, template-based recognition methods work quite well for machine-printed characters. These are uniform in size, shape, and orientation, and preprocessing methods can be devised that produce quite recognizable characters for any particular document or set of documents that were created in the same way. On the other hand, characters printed by a human show a large degree of variation in shape, size, orientation, and grey-level intensity, even in sets of characters printed by the same person. This variation mitigates against the use of templates.

One likely solution is to represent the template digits as vectors. This is commonly done in computer typography systems, where the fonts are stored in vector form. This permits easy scaling and rotation, allowing one set of characters to be used for all sizes, plus bold and italic forms. The fonts were originally produced, painstakingly and with human assistance, so as to be of high visual quality when scaled to large sizes and thick line widths. Although fonts are often stored as outlines, it seems that the vector form generally has the properties needed of a good template.

For applications in digit recognition, vectors that form the skeleton of the characters are used rather than those that form the outline. This yields a good abstraction of the shape, and permits the lines to be thickened in an arbitrary way. The templates are stored as sets of four integers, those being the starting and ending row and column on a standard grid. All templates have the same size, 10 by 10; this means that all coordinates in any template have an integer value between 0 and 9 inclusive. Given a scale and rotation, then, all templates in the collection would be modified in a consistent way.

There have been some efforts to normalize handprinted characters, but these are only successful for certain types of variation. Orientation, slant, and scale can be accounted for to some extent, but other aspects, such as line thickness, have not been. Thus, either an enormous number of raster templates are needed to account for all possible variations, or standard template-matching techniques fail. There is enough difference in shape between different digits to permit human recognition at high rates of success; perhaps the templates should abstract the shape more accurately than possible using a raster model, which depends on individual pixel-to-pixel correspondence and not more global shape properties.

A vector template can be produced using only a pencil and perhaps some graph paper, and indeed, the first set of templates was generated in just this way. An example appears in Figure 9.22, which shows a template for the

digit 2. Figure 9.22a shows the vector coordinates that were obtained manually from a line drawing of a 2 on a 20×20 grid. This is drawn as lines (Figure 9.22b) using the original scale, and also using a new scale: 20×10 (Figure 9.22c). This particular template is, by itself, able to match over 80% of the sample 2 images encountered in the test data set, when the matching method described below was used.

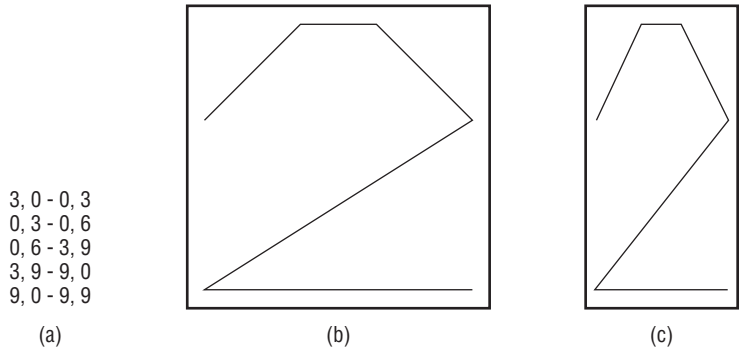


Figure 9.22: An example vector template. (a) The coordinates of the vector endpoints for a 2 template. (b) Vectors drawn on a 20×20 grid. (c) Vectors drawn on 20×10 grid.

It is also possible to create a template from a data image, and this might be desirable when starting to process data from a new source. The first step in this process is to threshold and then thin the input image. Thinning can be done using any competent algorithm: we have used both Holt's variation on Zhang-Suen (Section 6.3) and the force-based method (Section 5) to yield acceptable sets of skeletal pixels. The result is a binary image in which only skeletal pixels have a value of 0; all others are 1. Figure 9.23 gives an example of this process, showing the original input image, the thresholded version, and the skeleton as located by both thinning methods.

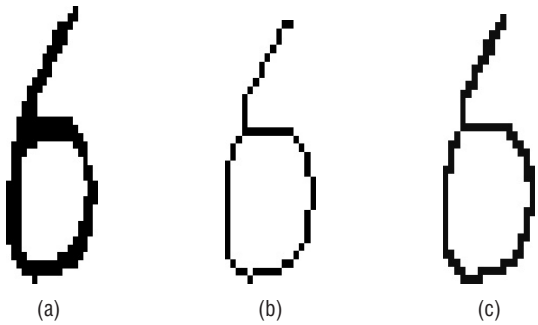
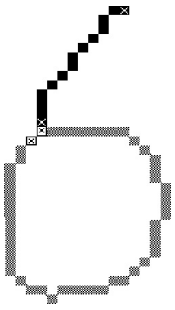


Figure 9.23: Converting a raster digit image into a vector template. (a) The original digit image. (b) Thinned version, using Zhang-Suen. (c) Thinned version using force-based thinning (Parker et al).

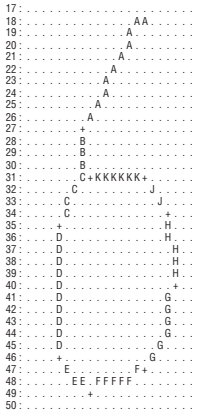
At this point the pixels are collected into sets, each representing a curve. An end pixel is found (either a pixel connected only to one other, or an arbitrary starting point if no such pixel exists) and the set of pixels connected to it are saved, taking care to trace only one curve. The method described in Lam and Suen works very well here. Finally, a set of vectors is extracted from each curve using a recursive splitting technique, a relatively simple and common method for vectorizing small, simple images. Briefly, the endpoints of the curve are presumed initially to be the start and endpoints of a line, and the distances between all pixels in the curve and the mathematical line are computed. If the maximum distance for this set of pixels exceeds a predetermined threshold, then the curve is broken into two curves at the pixel having that maximum distance, and the same procedure is applied again (recursively) to each of the two curve sections. Alternatively, if the maximum distance is less than the threshold, then the curve is presumed to be an approximation to a line, and the endpoint coordinates are saved as one of the vectors in the template. The coordinates are scaled down to the standard 10×10 grid after the extent of all the vectors has been determined. Figure 9.24 shows this vectorization process applied to the skeleton of Figure 9.23b, and shows its final appearance after being scaled. This particular template matched over half of all the *6* images in the test data set. The program that creates templates from images actually generates *C* code for the initialization of the template data array in the matching program.



(a)

A: (18, 40) (27, 32)
B: (27, 32) (30, 32)
C: (31, 32) (35, 29)
D: (35, 29) (46, 29)
E: (46, 29) (49, 33)
F: (49, 33) (47, 40)
G: (47, 40) (40, 44)
H: (40, 44) (34, 43)
J: (34, 43) (31, 40)
K: (31, 40) (31, 33)

(b)



(c)

Figure 9.24: Finding vectors in the thinned image. (a) The curves encountered (2 of them). (b) Extracted vector coordinates. (c) Vectors (linear features) marked in the thinned image.

Once all the templates have been generated and there are multiple templates for each digit, the system is ready to recognize digits. An incoming image is first preprocessed in any desired fashion and is then thresholded. The width

of the lines in the image is then estimated using horizontal and vertical scans. A histogram containing the widths of the black portions of the image on all slices is produced, and the mode of this histogram has been found to be a close enough approximation to the actual line width. A better approximation can be had by computing the gradient at each pixel on the outline of the digit and finding the width of a slice through the digit in a direction perpendicular to the outline at that point, but this rarely gives a result sufficiently better to be worth the extra computation time. While the line width is being computed, the actual extent of the digit image is also found so that the templates can be scaled. This is saved as the coordinates of the upper-left and the lower-right pixels.

At this point the scaling factors for the templates can be computed. The templates will be scaled in the x and the y directions independently, and the same scale factors can be applied to all templates. The factors include an adjustment that results in a correct scaling accounting for the thickness of the line. Now the template vectors are drawn into an otherwise clear image the same size as the input image, producing an initial raster template that represents the scaled skeleton. Finally, each pixel is ''grown'' equally on all sides to give a line width comparable to that found in the input image. The result is a raster template with some similar properties to those found in the input image. Figure 9.25 illustrates the process of generating a raster template from a vector one.

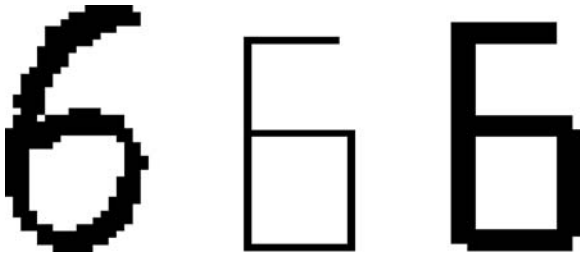


Figure 9.25: Matching using a vector template. (a) Input digit image (to be matched). (b) Scaled vector template. (c) Thickened vector template (not a good match here).

The matching process is somewhat different from that used in other template matching systems, but the goal is still to produce a measure of distance between the template and the image. The first step is to locate those pixels that are black in both images. These have a distance between them of zero, and are ignored in future processing. Next, each black pixel in the image has its nearest corresponding pixel in the template located and marked. The 8-distance between these pixels is noted, and a sum of these distances is computed. After all image pixels have been assigned corresponding pixels in the template

the total distance is an initial measure of similarity. Efforts have been made to reduce the distance total by looking at pairs of corresponding pixels and swapping those having a smaller distance after being swapped. This is a very time-consuming process, and does not greatly improve the distance.

At this point a numeric value that can be used as a goodness of match metric has been found. It is normalized to a per-pixel distance and stored as a measure for the digit having the same class as the template. The class having the smallest such measure over all templates is chosen as the class of the input digit image. Figure 9.26 shows the overlapping pixels and a distance map for the example begun in Figure 9.25.

Figure 9.26: The final stages of the template match. (a) Pixels that overlap between the template and the image. (b) Distance map between template and image. Darker pixels are farther away.

There is a problem with using this method to match a *1*, or any other object that has an extreme ratio of height to width. The problem is that the vectors for almost any template will match after having been scaled to fit. For example, the sides of a *0* will be brought together and the central hole will be filled in due to the width of the two lines. When recognizing *1* digits, we used one template for those cases where the image is sufficiently wide (2% of the sample) and a combination of aspect ratio and line width vs. image width for the rest of the cases. Once the *1* digits were recognized, templates were used in all the remaining cases.

The software implementing this method is called `recv.c`. It has been executed on many computers over the years, and is relatively quick and very accurate when correctly trained. In fact it won an international competition for handprinted symbol recognition in Barcelona in 2000 [Askoy, 2000].

Recognition rates on the 1000 digits of test data are excellent. The overall rate of recognition was 94.3%. Table 9.4 details the results.

Table 9.4: Vector Template Digit Recognition Rates

	0	1	2	3	4	5	6	7	8	9
CORRECT	99%	94%	98%	96%	94%	92%	90%	93%	95%	92%

There are multiple templates for each digit, but not necessarily the same number. Table 9.5 outlines the number of templates per digit.

Table 9.5: Number of Templates per Digit

0	1	2	3	4	5	6	7	8	9
1	1	3	1	5	3	2	4	4	4

The vector template method has been applied to symbols extracted from maps, and to music recognition. For an example of this, Figure 9.27 shows a quarter rest, the vectors, and a scaled template. Experiments have shown that scale is still an issue in that there is a better rate of recognition when the template and the data are about the same size. The method works best when large glyphs are vectorized into templates.

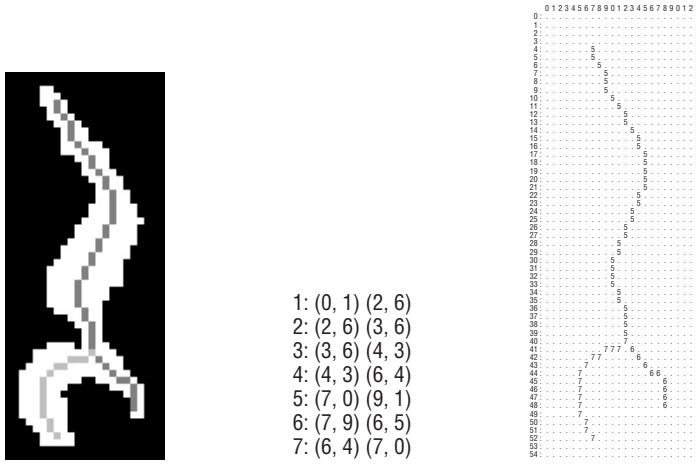


Figure 9.27: Vector templates applied to music symbol recognition. (a) A quarter rest. (b) The vectors extracted, in template creation. (c) The final template.

9.5.4 Neural Nets

The use of artificial neural systems (ANS) for various recognition tasks is well-founded in the literature [Aarts, 1989; Ansari, 1993; Carpenter, 1988; Fukushima, 1983; Mui, 1994; Shang, 1994; Touretzky, 1989; Yong, 1988].

The advantage of using *neural nets*, as they are commonly called, is not that the solution they provide is especially elegant or even fast; it is that the system ''learns'' its own algorithm for the classification task, and does so on actual samples of data. Indeed, the same basic neural net program that recognizes digits can also be used to recognize squares, circles, and triangles.

The goal of this section is not to explore thoroughly the use of neural nets; rather, the goal is to provide a summary of neural nets for the uninitiated, to show the utility of the method, and to provide pointers to more detailed information.

9.5.4.1 A Simple Neural Net

Before devising a neural net for recognizing digits, let's look at a simple example in an effort to explain the basic ideas. What is commonly called a neural net is an interconnected set of processing elements (PEs), each of which performs a very simple calculation. A single processing element, as shown in Figure 9.28, has some number of *inputs*, a *weight value* for each input, and an *output value*, which can be fanned out and used as inputs by other elements. Each of these values is numeric. The value associated with any node is called its *activation*, and is simply the sum of each input value multiplied by its respective weight value. The output of the PE may be simply the activation value, but is most often a function of the activation: the *activation function*, sometimes called the *output function*.

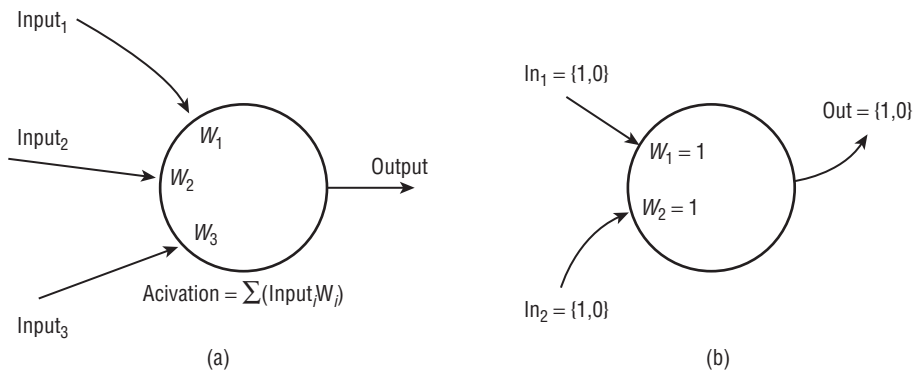


Figure 9.28: Neural net basics. (a) A single processing element with three inputs. (b) A processing element configured to operate as an AND function with two inputs.

A processing element is a primitive model of a neuron. Like a real neuron, there are many input values that interact (are summed) to produce an output. The response of a neuron is to *fire*, or to send a pulse-like signal to its output. In a real neuron, some of the inputs actually inhibit the firing; these would correspond to PE inputs having a negative weight value. Other neuronal inputs encourage the neuron to fire, which corresponds to a positive weight.

Often a single input is not sufficient to cause firing, which is why a PE sums all the weighted inputs.

The activation function ensures that the PE output values fall into a predefined legal range. For example, the PE in Figure 9.28b accepts binary input values and generates a binary output value. Since there are two inputs and both weights are 1, the four possible activations for this PE are:

INPUTS		ACTIVATION
In1	In2	
0	0	0
0	1	1
1	0	1
1	1	2

Since the output is supposed to be binary some thresholding must be performed, and this is done by the activation function. Suppose that this PE is to respond in the same way as a Boolean AND; in that case, the output should be 0 unless both inputs are 1, in which case the output should be 1. An activation function that performs in this manner is:

$$f(A) = \begin{cases} 0 & \text{if } a < 1.5 \\ 1 & \text{if } a > 1.5 \end{cases} \qquad \text{(EQ 9.8)}$$

That is, if the activation is 1 or less the output is 0; otherwise it is 1.

Processing elements are not used alone, but are connected as a *graph*, more commonly called a *network*. The output from a set of PEs can be used as inputs to another, or to many. There can be as many stages (*layers*) as is desired, and as many elements in each layer as needed.

Figure 9.29 shows two quite simple nets. The first (9.29a) is a combination of three of the two-input AND elements configured to act as a four-input AND. The interconnections are done in the obvious way, which is basically the same way we would wire up AND gates in a logic circuit. The second example (9.29b) shows a two-input AND that has a TRUE output and a FALSE output. The first layer of processing elements (to the left, and called the *input layer*) simply distribute their inputs over two outputs. The second layer of elements (to the right, called the output layer) each computes one of the AND values of their inputs. One responds if the inputs are both 1 (TRUE), whereas the other responds in all other situations. A simple way to accomplish this is to set the input weights of the input elements to 1 and pass this value through to the output. The TRUE element of the output layer is an AND element from Figure 9.27b, and the FALSE element is a TRUE element but with the

thresholding in the activation function reversed (i.e., return 0 if the activation is >1, and 1 otherwise).

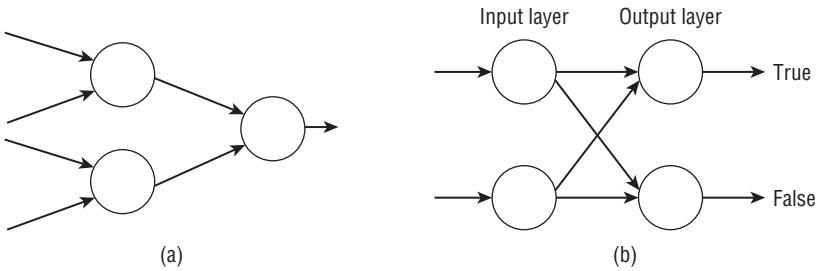


Figure 9.29: Simple neural nets. (a) A four-input AND, built from three two-input ANDs. (b) A two-input AND having discrete TRUE/FALSE output values.

Pleased with our success at this task, we now try to implement other logic functions. An OR is easy enough, but problems are encountered with the exclusive OR function (XOR). No matter what weights or activation functions we try, the XOR cannot be implemented. Why not? Let's take a closer look at what the PE for AND is actually doing.

In Figure 9.30a, the four possible inputs for the AND element are plotted in two dimensions; they can, in fact, be thought of as vectors. The line $1.5 = w_1 \text{Input}_1 + w_2 \text{Input}_2$ has been plotted in the same graph; note that the line divides the input values into two sets: one set corresponding to an output of 1 (which is circled in the figure), and one set with an output of 0. This is what neural nets do in general: divide N-dimensional space into regions, each corresponding to an output value.

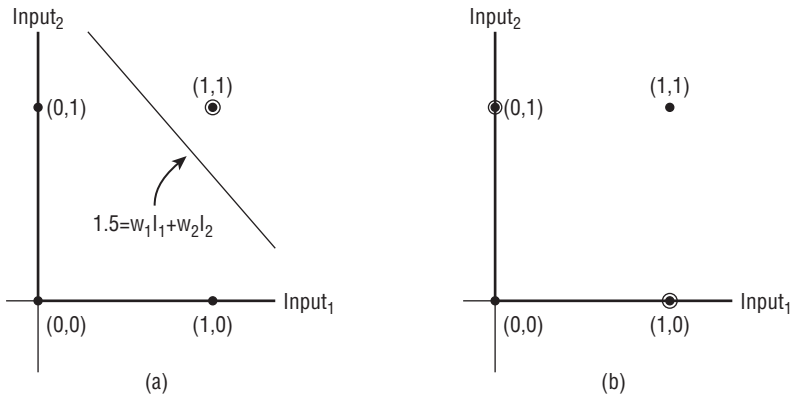


Figure 9.30: What the neural net actually computes. (a) The division of 2-space for the AND element. All inputs on one side of the line correspond to TRUE outputs (which are circled), and all inputs on the other side correspond to FALSE. (b) For the XOR function, there is no single line that can divide the TRUE (circled) points from the FALSE.

Now examine the problem of simulating an XOR in this light, as in Figure 9.30b. Note that there is no single straight line that can be drawn that has the input values that correspond to a 1 output in one region without also including at least one of the other inputs. This means that it is not possible to find a neural net of this type that can solve the XOR problem.

This is not to say that the problem cannot be solved. The solution entails straight lines, as shown in Figure 9.31. The middle region (the one between the two lines) corresponds to an output of 1, whereas the other two regions result in an output of 0. This partitioning of the input space is accomplished by adding an extra layer of processing elements (the *hidden layer*) between the input and output layers. Indeed, the use of hidden layers permits a neural net to solve quite complex problems.

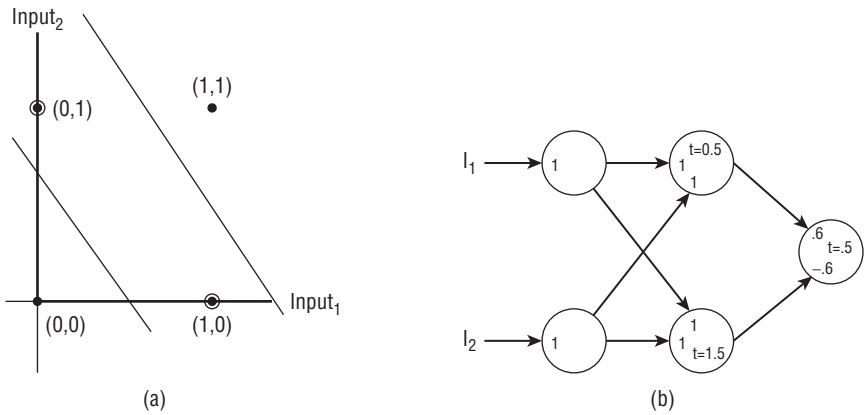


Figure 9.31: Solution to the XOR problem. (a) Dividing the solution space into three regions allows the middle region (the one containing the TRUE outputs) to be isolated from the other (FALSE) region. (b) The neural net that corresponds to this has a hidden layer of PEs that permits the extra subdivision. The numbers beside each input are the weights, and the values of t are the thresholds applied to the activation to get the output. Outputs greater than the activation yield a 1; otherwise, the output is 0.

An advantage of neural networks over more traditional computational methods is the fact that a net can *learn* to solve a particular problem. Learning, in a neural net sense, means the determination of the weights for each processing element. The nets that have been discussed to this point have been simple, and have had the weights already determined. In common use, a net is trained by presenting a sample of the data with known properties to the inputs, and then adjusting the weights until the outputs are correct. This process is repeated many times with different classes of data until the weights appear to stabilize, at which point the learning process is complete, and the net can be used to classify the inputs for unknown cases.

There are various ways in which a net can be ''taught'' a set of weights. The method that will be used for handprinted character recognition is known as *back-propagation*, and is commonly used for this sort of problem. It is based on a feedback of the outputs to the previous layers, and uses a least-squares/gradient-descent optimization method.

9.5.4.2 A Backpropagation Net for Digit Recognition

With the previous discussion on neural nets in mind, a three-layer backpropagation net (BPN) such as the one shown in Figure 9.32 is proposed. There are 48 nodes in the input layer, one node for each pixel in an image of 8 rows × 6 columns. The raster input digits are scaled to this size as a first step. The bounding box for each character is used rather than image size so that the scaling effect is consistent over all the data images.

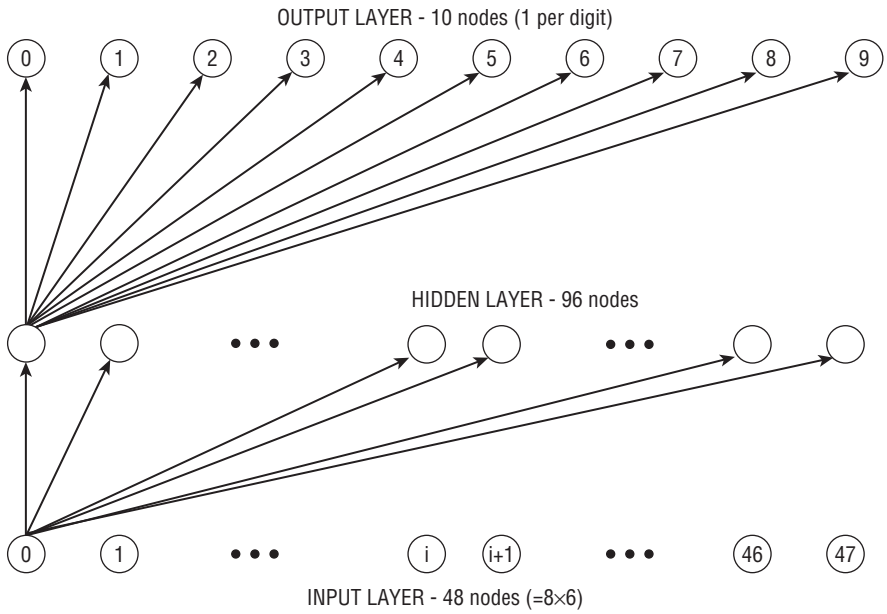


Figure 9.32: The backpropagation net proposed for digit recognition. Only some of the many connections between nodes are illustrated here.

Initially the weights are unknown. After the net is trained, the pixels of a digit image are used as the 48 input values; the output PE corresponding to the correct digit will have a 1 output, with the remaining output PEs being 0.

Training proceeds by applying the pixels of a known glyph to the inputs of the net. Once a scaled image is applied to the input layer, each input node

sends its value to all the hidden nodes. The output associated with a hidden node is the weighted sum of all the input values:

$$H_i = \sum_{j=0}^{48} W_{ij} I_j \qquad \text{(EQ 9.9)}$$

where H_i is the value associated with hidden node i, I_j is the value of input node j, and W_{ij} is the weight associated with the link between input node j and hidden node i. The output from hidden node i may simply be this value H_i, or, in the case where a binary output is needed, it may be a *logistic* function $f(H_i) = (1 + e^{-H_i})^{-1}$. In either case, a similar situation now exists between the hidden layer and the output layer; each output node has a value associated with it that is the weighted sum of all the hidden nodes:

$$O_k = \sum_{i=0}^{96} f(H_i) X_{ki} \qquad \text{(EQ 9.10)}$$

where O_k is the value associated with output node k, $f(H_i)$ is the output from hidden node i, and X_{ki} is the weight applied between hidden node i and output node k. The binary output from this output node is $f(O_k)$, and since there is one output node per digit, it is to be hoped that the largest output response will be from the node associated with the actual input digit class.

Before the net can be expected to correctly classify any digits it must be trained. This amounts to determining the weights W_{ij} and X_{ki} that lead to a correct classification. Initially these weights were set to random values between -0.5 and $+0.5$, which should lead to random classifications. The net is trained by applying known data to the inputs; that is, glyphs whose classification is known. The output values for all output nodes are computed, and the result is compared against the correct result. For example, when a three is used as input data, all the output nodes should have the output value 0 except output node 3, which should be 1. The error at each output node k can therefore be represented by:

$$\delta_k = (D_k - O_k) \qquad \text{(EQ 9.11)}$$

where D_k is the correct output value for output node k and O_k is the output value computed for output node k. The error of node O_k can be thought to have been contributed to by all the hidden nodes through the weights X_k, and so δ_k can be used to modify those weights to bring them closer to those needed to produce a correct classification. Minimizing the error E in the least-squares sense means minimizing:

$$E = \frac{1}{2} \sum_{k=1}^{10} \delta_k^2 \qquad \text{(EQ 9.12)}$$

Due more to tradition than function, most BPN systems minimize this expression using the method of *steepest descent*, which has some serious shortcomings: it is slow, inefficient, and does not always converge to the overall (*global*) minimum. Although much better schemes exist [Masters, 1995], we will stick to the most commonly encountered method and notation.

The method of steepest descent attempts to find the minimum of a function F of some number of variables N. The parameters to F can be thought of as a vector $\overline{x}$, for convenience. The process is iterative, consisting of the following steps:

1. Select a starting point in N space; this will be the vector $\overline{x}_0$.
2. Compute the gradient $\nabla F(\overline{x}_i)$ for the parameter vector $\overline{x}_i$; initially $i = 0$, and is incremented each iteration.
3. Pick an appropriate step size d_s. This is a factor by which each component of the gradient will be multiplied.
4. Compute the next set of parameters (the next point in N-space) by:

$$\overline{x}_{i+1} = \overline{x}_i - d_s \nabla F(\overline{x}_i) \quad \text{(EQ 9.13)}$$

5. Continue from step 2 until the error $\mathrm{F}(\overline{x}_i) - \nabla F(\overline{x}_i) < \epsilon$, for some predetermined value ϵ.

The backpropagation learning algorithm uses this scheme to minimize the error (Equation 9.12) by adjusting the weights on all the processing elements following the presentation of a known glyph to the input nodes. First the weights on the output nodes are adjusted, then the hidden nodes, and finally the input nodes. This is where the name *backpropagation* comes from: the changes to the weights propagates back from the output to the input.

After the mathematics of the gradient descent has been done [Freeman, 1991] the weights on the output nodes are updated by:

$$X'_{ki} = X_{ki} + \eta \delta_k f'(O_k) f(H_i) \quad \text{(EQ 9.14)}$$

and the hidden node weights are updated by:

$$W'_{ij} = W_{ij} + \eta \delta_i^h I_j \quad \text{(EQ 9.15)}$$

where the error term for the hidden node i is:

$$\delta_i^h = f'(H_i) \sum_k \delta_k X_{ki} \quad \text{(EQ 9.16)}$$

The value η in the expressions above is called the *learning rate parameter*, and controls the rate of convergence. A value near 0.25 is not unusual, but this can vary according to the type of problem being studied. A very nice discussion

of this entire procedure, and of backpropagation nets in general, can be found in Freeman [1991].

Each application of a new training image to the input layer amounts to an iteration of the steepest-descent minimization of the error in the weights. After each known glyph (training data) is applied to the neural net inputs, the weights are updated and a new error term E is computed. When E reaches an acceptably small value, training can be stopped.

This three-layer net was trained using 1000 digits (100 drawn from each class) presented alternately to the inputs; that is, one sample of each class 0 through 9 produced by a single writer was presented, followed by another set 0–9, and so on. If all the 0 digits were presented first, then the 1s and so on, the net would tend to forget the earlier digits and recognize only the later ones trained. After the training phase, a second set of 1000 digits from 100 different writers was used to test the recognition rate. The results were initially not very good, at least for nines (see Table 9.6).

Table 9.6: Backpropagation Net — Digit Recognition Rates (%)

	0	1	2	3	4	5	6	7	8	9
Correct	99	93	99	95	100	95	99	100	95	74

What was even more interesting was that when the same net was trained using the second set of data and used to recognize the first set (i.e., the opposite of the situation above) the results improved (see Table 9.7).

Table 9.7: Backpropagation Net — 2nd Digit Recognition Rates (%)

	0	1	2	3	4	5	6	7	8	9
Correct	100	99	100	94	99	99	94	98	99	98

An explanation for this may be that the digits in the second set are in some way more consistent, and do not provide a sufficiently diverse training set. The error found after training was not as good in the second case either, and perhaps more training data is needed. One common means of dealing with this problem [Gaines, 1995] is to introduce noise into the training data. For example, when black training pixels are changed to white with a probability of 0.2, the recognition rate for nines in Table 9.6 increases to 88%. However, it may be useful to have one or two low-rate digits in the multiple classifier to note the effect on overall recognition, so the rates of Table 9.6 will be used.

The logistic function gives an almost binary output that can be used as a ranking, so that a neural net of this type can provide both a simple classification and a ranking of likely classifications that can be ordered according to likelihood. This will become important in the next chapter.

The neural-net software on the website consists of two programs. The first, named `nnlearn.c`, trains the net on sample data. The weights resulting from this process are saved on a file named `weights.dat`. The second program is `nnclass.c` and this reads the weights from the file and attempts to recognize a glyph, which is in the form of an image file. The file `weights.dat` contains weights obtained by training the net on the set of 1000 digits, and can be used in conjunction with the net software to perform actual recognitions. The program `nncvt.c` can be used to create a neural net input file (i.e., one suitable for `nnclass.c`) from an image of a digit.

9.6 The Use of Multiple Classifiers

There exists quite a variety of methods for handprinted-character recognition, and it may very well be that there is no one method that can be thought of as the ''best'' under all circumstances. Each algorithm has strengths and weaknesses, good ideas and bad. There is a way to take advantage of this variety: Apply many methods to the same recognition task, and have a scheme to merge the results; this should be successful over a wider range of inputs than would any individual method [Parker, 1994]. The weaknesses should, in an ideal situation, more or less cancel out rather than reinforcing each other, giving high recognition rates under many sets of conditions. This situation can be encouraged by discarding methods that are the same, at least as far as the classifications of the data are concerned.

The previous section discussed four methods for classifying handprinted digits. Now the output from these will be combined to form a single classification.

9.6.1 Merging Multiple Methods

There are four classifiers in the multiple classifier system, those being the ones discussed in previous sections: character outline features, convex deficiencies, vector templates, and a neural network. When applied to a set of sample digit images, the simple majority vote (SMV) of these classifiers gave the following results:

Correct: 994 Incorrect: 2 Rejected: 4

This is in spite of poor results (76% recognition) from the neural net classifier (#4) on nines; indeed, 100% of the nines are recognized by the multiple classifier. This begs the question ''what is the contribution of any one classifier to the overall result?'' To determine this for the SMV case is simple.

The multiple classifier can be run using any three of the four individual classifiers, and the results can be compared against the five classifier case above to determine whether the missing classifier assisted in the classification. If omitting a classifier does not improve the results, it *can* be removed from consideration; if omitting a classifier improves the results, then that classifier *must* not be used at all.

As was pointed out in Section 8.5.3, the weighted majority vote has a parameter that must be varied over all possible values when evaluting its performance on a particular data set. The results were shown in Table 7, but are given again here in the context of the other results (see Table 9.8). The Acceptability measure is being used here.

Table 9.8: Acceptability of the Multiple Classifier Using a Weighted Majority Vote

ALPHA	ACCEPTABILITY
0.05	0.992
0.25	0.993
0.50	0.998
0.75	0.823

Duplicate rows are not shown in the table.

DWMV also uses the α parameter and can be evaluated in a fashion identical to what has just been done for WMV. The optimal value of α, obtained from Table 9.9, was found to be 0.25.

Table 9.9: Acceptability of the Multiple Classifier Using a Dissenting Weighted Majority Vote

ALPHA	ACCEPTABILITY
0.05	0.994
0.25	0.994
0.30	0.983
0.45	0.983
0.55	0.877
0.65	0.877
0.80	0.823
0.85	0.823

The Borda, Black, and Copeland rules were implemented as described in Section 8.5.5 and applied to the four-classifier problem, and the results are summarized in Table 9.10.

Table 9.10: Results of the Voting Rules for Rank Ordering (Omit #4)

RULE	RECOGNITION	ERROR	REJECTION	RELIABILITY	ACCEPTABILITY
Borda	99.9	0.1	0.0	0.999	0.998
Black	99.9	0.1	0.0	0.999	0.998
Copeland	99.6	0.2	0.2	0.998	0.994

From this table, it would appear that the Borda scheme is tied with Black, followed by Copeland. It is important to temper this view with the fact that this result was obtained from basically one observation. Confirmation would come from applying these schemes to a large number of sets of characters.

Another consideration is that a voting scheme may err in favor of the correct classification when it should, in fact, be rejected. Upon careful analysis this was found to have happened for the Borda method applied to digit #267. The rankings were:

- **Classifier 1** — 2
- **Classifier 2** — 1 7 4 2 9
- **Classifier 3** — 2
- **Classifier 4** — 1 9 6 3 2 7 8 5
- **Classifier 5** — 1 2 9

The Borda count for the one digit is 27, and for the two digit is 37, giving a classification of two even though the majority winner and the Condorcet winner is one! Thus, the Black scheme classifies this digit (correctly according to the votes, in my opinion) as a one. Given this problem, and the fact that Black and Borda are otherwise equally acceptable, my conclusion is that the Black classifier is slightly superior to the others.

There is little actual type 3 data, but it could be approximated by using the *a posteriori* method described in Section 8 and Equations 8.12 and 8.13, where it is used to convert type 1 responses to type 3 responses. Using this approximate data set, the result obtained by merging type 3 responses using averaging is given by:

Correct: 997 Incorrect: 3 Rejected: 0

Acceptability is 0.994.

9.6.2 Results From The Multiple Classifier

Using the acceptability measure to assess each of the merging methods discussed, we need to look only at the best method in each of the three groups — that is, the best multiple type 1 classifier, the best type 2, and the best type 3. The best three are shown in Table 9.11.

Table 9.11: Multiple Classifier Performance

RULE	DATA CLASS	ACCEPTABILITY
SMV	1	0.994
Black	2	0.998
Average	3	0.994

From the preceding table, it can be seen that the best classifier explored here uses the Black scheme for merging rank ordered responses.

9.7 Printed Music Recognition — A Study

In order to provide a second example of symbol recognition, a problem has been selected that is familiar to most — that of reading printed music. Like handprinted-character recognition, optical music recognition (OMR) is a problem that has yet to be solved in a satisfactory manner. There is a considerable commercial interest in doing so, and it should be concluded from this that the problem is a difficult one.

One reason OMR is hard is that most of the symbols are connected to one another. The staff lines touch most of the symbols in a score, connecting them into one large region. Since the problem of touching characters was a major reason that the character recognition system of Chapter 8 was not a great success, the importance of the staff lines cannot be overestimated. This, it would appear, is the first problem in the OMR system: to locate the staff lines and remove them without seriously affecting the remainder of the score.

Having isolated at least the majority of the symbols, the next step is to recognize them. For machine-printed scores the symbols should be uniform enough to permit a template-matching scheme to be used. This may classify enough of the symbols so that the number of possibilities for the remaining ones is significantly decreased.

Finally, the use of context will be used to detect errors and resolve ambiguities. This step is rather like the use of a spelling checker to ensure that the words extracted by an OCR system are, in fact, words. The final system will be compared against some commercially available programs to see how it stacks up.

9.7.1 Staff Lines

Figure 9.33 shows a sample piece of music notation treated as an image. The staff lines are the five long horizontal lines that provide a framework for the symbols. Each line and each space between lines represents a musical note on the scale, and so the position of the notes vertically indicates tone, whereas the horizontal position indicates the order in which they are played.

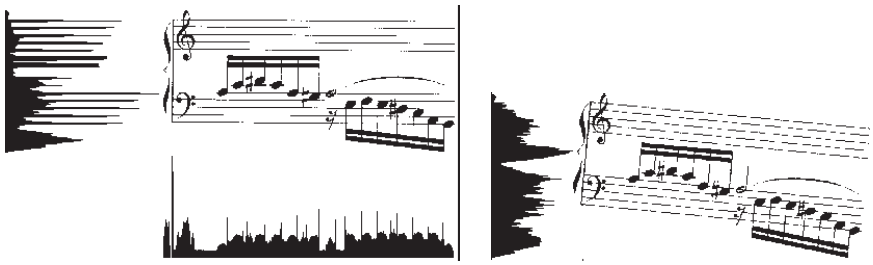


Figure 9.33: Examples of scanned music notation (a) A portion of Mozart, showing the horizontal and vertical projections. (b) The same portion rotated five degrees. The horizontal projection is virtually useless for locating the staff lines.

Horizontal projections have been used to locate the staff lines [Bulis, 1992; Kato, 1992], but even relatively small skew angles can lead to problems (Figure 9.33b). A large set of angles could be tried in an attempt to find a best angle, but for images the size of a scanned page of sheet music (10 MB and larger) this could require a great deal of time. However, the Hough transform was used with some success to identify the skew angle of printed text, and should work here. In fact, the Hough transform can be thought of as a calculation of all the projections that are possible.

Since the Hough transform is generally slow too, the structure of the staff lines can be used to speed things up. Instead of transforming all black pixels, collect pixels into short horizontal line segments and transform the coordinates of the center of the segment. Although this will actually work for a perfect image, even a small amount of curvature in the lines will prevent success

(Figure 9.34a). Curvature can result from the page not being flat against the scanner (such as when a book is being scanned). A better approach would be to break up the staff horizontally into sections small enough so that any reasonable curvature in the entire staff can be ignored in the section (Figure 9.34b). The smaller pieces are *effectively* straight. This actually does work very well, but requires about 90 seconds per page, and is quite complex to implement correctly.

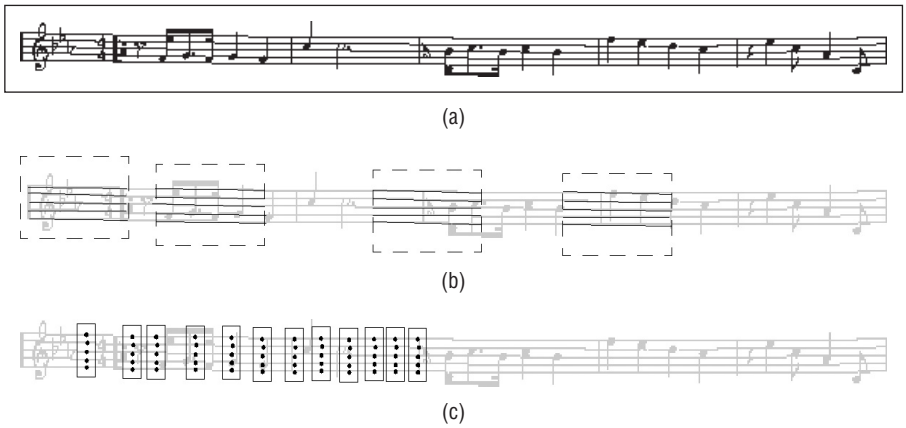


Figure 9.34: Hough transform method for locating staff lines. (a) A music staff with a slight curve — place a ruler along the top staff line. (b) Breaking the staff into small horizontal sections. The sections can be relinked later. (c) The use of vertical samples.

One method that works well and is fast enough uses the fact that the staff lines are equally spaced. Simply look at columns of pixels and look for five equally spaced and sized runs of black pixels, as shown in Figure 9.34c. A large-enough collection of these staff samples will collect into groups, each having similar spacing, thickness, and vertical position. Within each group, the angles between all the samples can be calculated, and samples that disagree wildly with the median angle can be discarded. Now that the angle has been found, a Hough transform can be computed quickly to find ρ. This method finds the staff lines in about 15 seconds per page, or four times as fast as the best alternative Hough-based method.

Now the staff lines can be removed by travelling along each line, deleting the black pixels except where the vertical run length (thickness) exceeds a specified threshold, which is dependent on the staff line thickness. This simple precaution prevents the removal of pixels belonging to note heads, sharps, flats, and other symbols that may touch a staff line. Figure 9.35 shows an image before and after staff line removal.

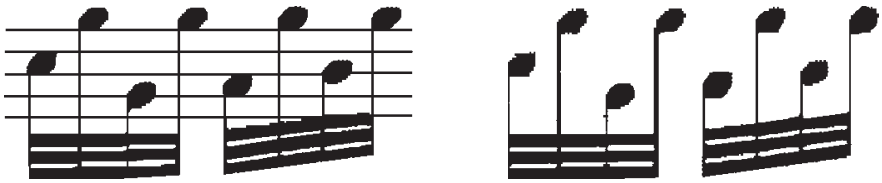


Figure 9.35: Staff line removal, using staff samples. (a) Before. (b) After.

9.7.2 Segmentation

In this instance, segmentation refers to the process of identifying the regions of the image that contain music, text, and artwork. The regions containing music can then be addressed, leaving the rest as ''noise.'' Segmentation begins on an image in which the staff lines have been removed by identifying the connected components. These are simply sets of pixels in which any pixel can be reached from any other by traveling on black pixels only.

While the connected components can be found in a recursive tracing strategy, one efficient way to find and represent them uses *line adjacency graphs* (LAGs). When scanning the image along either rows or columns, each run of black pixels becomes a node, the coordinates of which are those of the center pixel in the run. An edge is placed between two nodes if the runs associated with the nodes overlap, and the runs are on adjacent rows (columns). The basic idea is illustrated in Figure 9.36.

Now the LAG can be compressed, so as to occupy less space. This is accomplished by merging nodes A and B that have the following properties (assume horizontal run lengths):

1. A is above B (is on the preceding row).
2. A has *degree* (A, 1) and B has *degree* (1, B).
3. A and B have nearly the same run length.
4. The resulting amalgamated node can be represented approximately by a straight line.

The degree of a node is the number of edges on each side (above and below). A node has degree (2,3) if there are two edges connecting to it from above (the previous row) and three edges connecting below (the next row). Figure 9.36 attempts to make this situation more clear.

When the overall connected component analysis is complete, we have a LAG for each component in the score. Since the LAG is found after the staff lines are removed, there is a good chance that each represents a symbol, or set of related symbols. Indeed, any that would intersect a staff are considered to

be music symbols. However, there are many other symbols on a score, most notably text (e.g., lyrics), and these symbols clutter the scene significantly. Text should therefore be removed, if possible.

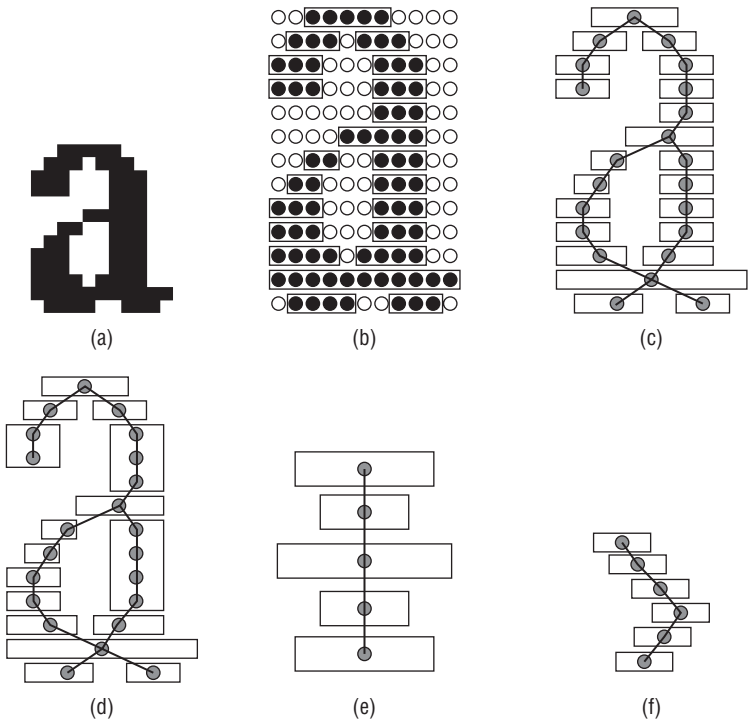


Figure 9.36: Line adjacency graphs. (a) Sample glyph. (b) Horizontal runs that are used to create the LAG. (c) The LAG found from (b). (d) The compressed version of the LAG. (e) In the compression method, different-sized runs are not merged. (f) Even if the runs have identical lengths, the set of nodes must form a linear collection, not a curve or corner.

One algorithm for doing this [Fletcher, 1988] was actually designed for distinguishing between regions of text and graphics on a document page. The basic idea is that text consists of connected components that are oriented along a straight line. A set of colinear objects having a regular spacing forms a word, and may be removed since text is of no interest to the OMR system. The centroids of each connected region are fed into a Hough transform to determine colinearity. Then any LAG sets that are words, based on their spacing, are marked as such for later deletion.

This is an oversimplification of a fairly complex algorithm. The result of this segmentation process is exemplified by Figure 9.37, in which a page from a simple score has had the text and nontext regions identified very accurately. This means that the recognition of music symbols can proceed, concentrating

on the nontext regions. Some text may remain in the score, and can later be discarded by the matching process.

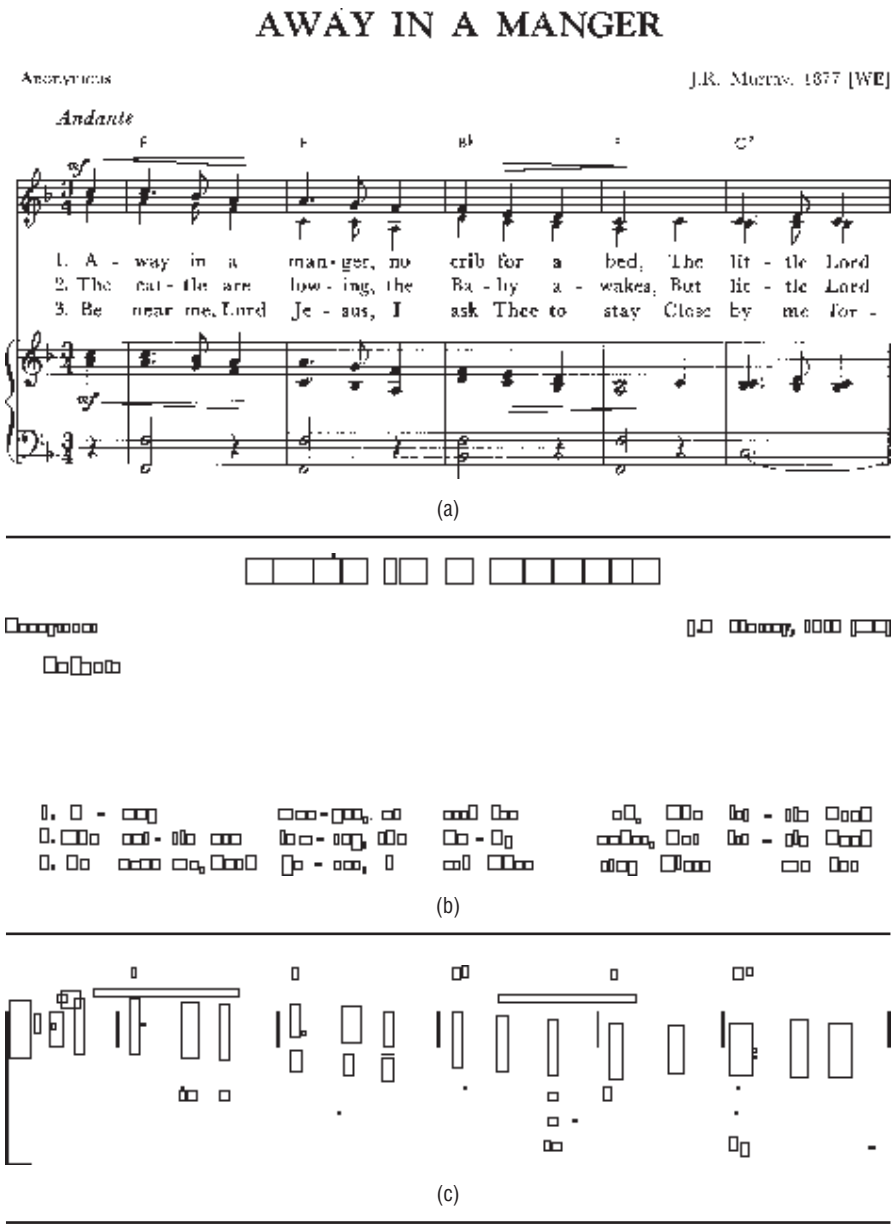


Figure 9.37: Results of segmentation. (a) Original score image. (b) The text regions identified by the Fletcher algorithm. (c) The nontext (music?) regions. Note that a little of the text remains with the music.

9.7.3 Music Symbol Recognition

Music notation is unusual in that there are a great many horizontal and vertical straight-line segments within a score. For example, measures are separated by a vertical line, and most notes consist of a vertical stem and an elliptical note head. This is convenient, because the LAG representation lends itself to the identification of lines: every node in the compressed LAG represents a line segment, and adjacent line segments can be merged into a single, longer one if their slopes are nearly the same.

Because of this, the recognition of symbols is split into three parts: horizontal lines, vertical lines, and others; this final part can be further split into note heads and symbols. Figure 9.38 shows the kind of result to be expected from the use of LAGs for the location of horizontal and vertical lines. This method is accurate enough to provide a means for identifying the other features; specifically, note heads are to be found near the end of a stem, which is a vertical line segment.

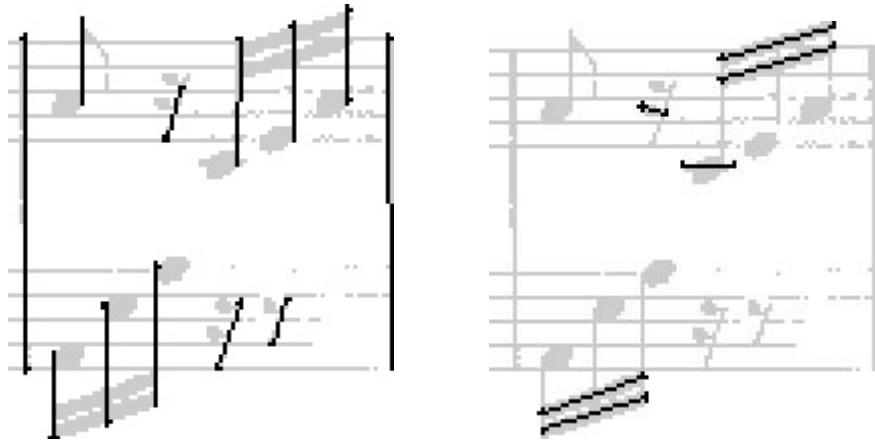


Figure 9.38: The identification of line segments using LAGs. (a) Vertical lines. (b) Horizontal lines. Note that the beams are found consistently.

For recognizing symbols (such as sharps, flats, and rests) that are often isolated by staff line removal, the method of character outlines (see Section 9.5.1) was used with great success. The features used in the recognition were found by measuring a collection of known symbols, and the resulting database of features was used to recognize the unknown glyphs.

Note heads, on the other hand, are almost never isolated, being connected to stems. The method used to locate note heads is the template-matching scheme,

last seen in Section 9.2. The hierarchical matching scheme was used to speed up the process, since the vast majority of the processing time of the entire OMR system was used in matching templates.

Many variations of the note heads were used as templates. This is essential, since the staff line removal is not perfect, and often leaves small stems where they intersect note heads. Also, in the case of whole and half notes (which have a hole), the staff line can be seen running through the center of the note. Figure 9.39 shows a few of the templates needed to recognize the head of a half note.

Figure 9.39: Templates for the head of a half note. (a) Basic template. (b) A template for a half note on a staff line. (c) Half note between staff lines. (d) Half note below a ledger line.

Once the individual symbols have been found, they are synthesized into more complex forms by using context. Structures are represented in the form of *graph grammars* [Claus, 1978; Ehig, 1990]. For example, a sharp symbol, two note heads, two stems, and a beam can, if the situation is right, be combined into a single high-level symbol (as illustrated in Figure 9.40). Graphs are created from the low-level symbols in the score, and are parsed by a *graph parser* to perform the high-level match [Fahmy, 1993; Reed, 1995].

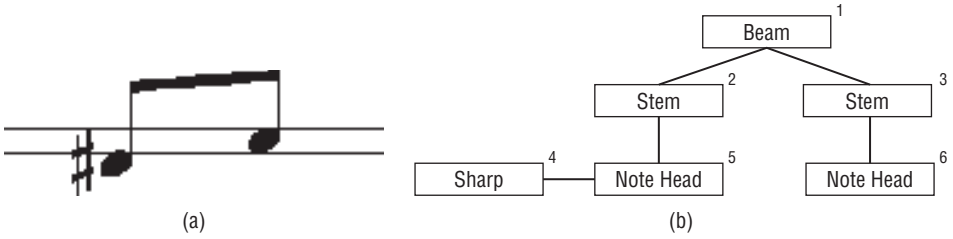


Figure 9.40: Using a graph representation to incorporate context. (a) A small sample, containing six primitive symbols. (b) The graph representation of the sample.

The OMR system described here, which is called Lemon, was tested on multiple scores and they gave an overall recognition rate of 95.2% on 10 samples containing more than 2,500 symbols.

9.8 Source Code for Neural Net Recognition System

```
/*Backpropagation network + character recognition */
/* TRIAL I: Test data is a binary to decimal conversion */
#include <stdio.h>
#include <math.h>
#include <malloc.h>

int NO_OF_INPUTS = 0;
int NO_OF_HIDDEN = 0;
int NO_OF_OUTPUTS = 0;

float *inputs;/* Input values in the first layer */

float **hweights;/* Weights for hidden layer. HWEIGHTS [i] [j] */

                /* is the weight for hidden node i from */

                /* input node j. */
float *hidden;/* Outputs from the hidden layer */
float *vhidden;
float *err_hidden; /* Errors in hidden nodes */

float **oweights;/* Weights for the output layer, as before */
float *outputs;/*Final outputs */
float *voutputs;
float *err_out;/* Errors in output nodes */

float *should;/* Correct output vector for training datum */
float learning_rate = 0.3;
FILE *training_data=0;/* File with training data */
FILE *test_data=0; /* File with unknown data */
FILE *infile;   /* One of the two files above */
int actual;     /* Actual digits - tells us the output */

void compute_hidden (int node);
void compute_output (int node);
float output_function (float x);
void compute_all_hidden ();
void compute_all_outputs ();
float weight_init (void);
void initialize_all_weights ();
float compute_output_error ();
float compute_hidden_node_error (int node);
void compute_hidden_error ();
void update_output ();
void update_hidden (void);
float * fvector (int n);
float **fmatrix (int n, int m);
```

```
void setup (void);
void get_params (int *ni, int *nh, int *no);
int get_inputs (float *x);

int printf ();
int scanf  ();

/*   Compute the output from hidden node NODE*/
void compute_hidden (int node)
{

     int i = 0;
     float x = 0;

     for (i=0; i<NO_OF_INPUTS; i++)
       x += inputs[i]*hweights [node] [i];
     hidden [node] = output_function (x);
     vhidden [node] = x;
}

/*   Compute the output from output node NODE*/
void compute_output (int node)
{
     int i = 0;
     float x = 0;

     for (i=0; i<NO_OF_HIDDEN; i++)
       x += hidden [i]*oweights[node] [i];
     outputs [node] = output_function (x);
     voutputs [node] = x;
}

/*    Output function for hidden node — linear or sigmoid*/
float output_function (float x)
{
     return 1.0/(1.0 + exp ((double) (-x)));

     return x; /* Linear */
}

/*      Derivative of the output function*/
float of_derivatives (float x)
{

     float a = 0.0;

     a = output_function (x);

     return 1.0; /* Linear */
     return a* (1.0 — a);
}

/*    Compute all hidden nodes*/
```

```
void compute_all_hidden ()
{

     int i = 0;

     for (i = 0; i<NO_OF_HIDDEN; i ++)
       compute_hidden (i);
}

/*   Compute all hidden nodes*/
void compute_all_outputs ()
{
     int i = 0;

     for (i=0; i<NO_OF_OUTPUTS; i++)
       compute_output (i);
}

/*    Initialize a weight*/
float weight_init (void)
{
     double drand48();

     return (float) (drand48 () - 0.5);
}

/*   Initialize all weights*/
void initialize_all_weights ()
{

     init i = 0; j = 0;

     for (i=0; i<NO_OF_INPUTS; i++);
       for (j=0; j<NO_OF_HIDDEN; j++)
         hweights [j] [i] = weight_init ();

     for (i = 0; i<NO_OF_HIDDEN; i++)
       for (j=0; j<NO_OF_OUTPUTS; j++)
         oweights [j] [i] = weight_init ();
}

/*    Calculate the error in the output nodes*/
float compute_output_error ()
{
     int i = 0;
     int x = 0;

     for (i=0; i<NO_OF_OUTPUTS; i++)
     {
       err_out [i] = (should [i]-outputs [i]) *
of_derivative (voutputs [i]);
       x + = err_out [i];
     }
```

```
     return x;
}

/*    What SHOULD the output vector be?*/
compute_training_outputs()
{

     int i;

     printf ("Output SHOULD be:\n");
     printf ("0 1 2 3 4 5 6 7 8 9\n");
     for (i = 0; i<NO_OF_OUTPUTS; i++)
     {
       if (i==actual) should[i] = 1.0;
         else should[i] = 0.0;
       printf ("%5.1f", should [i]);
     }
     printf ("\n");
}

/*    Compute the error term for the given hidden node*/
float compute_hidden_node_error (int node)
{

     int i = 0;
     float x = 0.0;

     for (i=0; i<NO_OF_OUTPUTS; i++)
       x += err_out[i]*oweights[i][node];

     return of_derivative (vhidden [node]) * x;
}

/*   Compute all hidden node error terms*/
void compute_hidden_error ()
{

     int i = 0;
     for (i=0; i<NO_OF_HIDDEN; i++)
       err_hidden[i] = compute_hidden_node_error(i);
}

/*   Update the output layer weights*/
void update_output ()
{

     int i=0, j=0;

     for (i=0; i<NO_OF_OUTPUTS; i++)
       for (j=0; j<NO_OF_HIDDEN; j++)
         oweights [i] [j] +=
learning_rate*err_out [i*hidden [j];
}
```

```
/*   Update the hidden layer weights*/
void update_hidden (void)
{

     int i=0; j=0;
     for (i=0; i<NO_OF_HIDDEN; i++)
       for (j=0; j<NO_OF_INPUTS; j++)
         hweights[i][j] +=
learning_rate*err_hidden[i]*inputs[j];
}

float compute_error_term ()
{

     int i = 0;
     float x = 0.0;
     for (i=0; i<NO_OF_OUTPUTS; i++)
       x += (err_out [i] *err_out [i]);
     return x/2.0;
}

float * fvector (int n)
{

     return (float *) malloc (sizeof (float) *n);
}

float **fmatrix (int n, int m)
{

     int i = 0;
     float *x, **y;

/*Allocate rows */
     y = float **)malloc (size of (float) *n);

/* Allocate NXM array of floats */
     x = (float *)malloc (size of (float) *n*m);
/*Set pointers in y to each row in x */
     for (i=0; i<n; i++)
       y[i] = & (x[i*m]);

     return y;
}

/*   Allocate all arrays and matrices*/
void setup (void)
{

     inputs       = fvector (NO_OF_INPUTS);

     hweights     = fmatrix (NO_OF_HIDDEN, NO_OF_INPUTS);
```

```
     hidden       = fvector (NO_OF_HIDDEN);
     vhidden      = fvector (NO_OF_HIDDEN);
     err_hidden   = fvector (NO_OF_HIDDEN);

     oweights     = fmatrix (NO_OF_OUTPUTS, NO_OF_HIDDEN);
     outputs      = fvector (NO_OF_OUTPUTS);
     voutputs     = fvector (NO_OF_OUTPUTS);
     err_out      = fvector (NO_OF_OUTPUTS);

     should       = fvector (NO_OF_OUTPUTS);
}

void get_params (int *ni, int *nh, int *no)
{

     printf ("How many input nodes:");
     scanf ("%d", ni);
     printf ("How many hidden nodes:");
     scanf ("%d", nh);
     printf ("How many output nodes:");
     scanf ("%d", no);
}

int get_inputs (float *x)
}
     int i, k;

     for (i = 0; i<NO_OF_INPUTS; i++)
     {
       k = fscanf (infile, "%f", &(x[i]));
       if (k<1) return 0:
     }
     if (infile == training_data)
       fscanf (infile, "%d", &actual);

     return 1;
}

void print_outputs ()
{
     int i, j;

     j = 0;
     for (i=0; i<NO_OF_OUTPUTS; i++)
{
       printf ("%f", outputs[i]);
       if (outputs[i] > outputs[j]) j = i;
     }
     printf ("Actual %d NN classified as %d\n", actual, j);
}

int main (int argc, char *argv[])
{
```

```
     int k = 0;
     float x = 0.0;
     int dset = 1;

/* Look for data files */
     if (argc < 3)
     {

          printf ("bpn <training set> <data set>\n");
          exit(1);
     }

     training_data = fopen (argc[1], "r");
     if (training_data == NULL)
     {
          printf ("Can't open training data '%s'\n",
argv[1]);
          exit (2);
     }
     infile = training_data;

/* Get the size of the net */
     get_params (&NO_OF_INPUTS, &NO_OF_HIDDEN, &NO_OF_OUTPUTS);

/* Initialize */
     setup ();
     initialize_all_weights ();

/* Train */
     k = get_inputs (inputs);
     while (k)
     {
       printf ("Training on set %d\n", dset);
       compute_all_hidden ();
       compute_all_outputs ();

/* Weight errors propagate backwards */
       compute_training_outputs ();
       compute_output_error ();
       compute_hidden_error ();

       update_output ();
       update_hidden ();

       x = compute_error_term();
       printf ("Set %d error term is %f\n", dset, x);
       k = get_inputs (inputs);
       dset++;
     }
     fclose (training_data);
     training_data = NULL; infile = NULL;

     test_data = fopen (argv[2], "r");
```

```
    if (test_data == NULL)
    {

      printf ("Can't open data '%s'\n", argv[2]);
      exit (3);
    }
    infile = test_data;

/* Now apply the NN to the remaining inputs */
    k = get_inputs (inputs);
    while (k)
    {
      compute_all_hidden ();
      compute_all_outputs ();
      print_outputs ();
      k = get_inputs (inputs);
    }
    fclose (test_data);

}
```

9.9 Website Files

`baird.c`	Baird algorithm for skew detection; creates a point image for part 2.
`hskew.c`	Baird part 2; uses Hough transform to find the skew angle.
`kfill.c`	kFill smoothing/noise reduction.
`learn.c`	Learns perfect templates from a sample image; creates a data file for template recognition (`ocr.c`).
`learn2.c`	Second version of learning of templates, this time from good quality scanned images.
`slhist.c`	Calculates a slope histogram from a given image file.
`learn3.c`	Creates file of features to recognize characters.
`ocr1.c`	Template recognition of characters.
`ocr2.c`	Recognition of characters from scanned templates.
`ocr3.c`	Use of statistical recognition for symbols.
`nnlearn.c`	Learning (training) for the basic neural network. Give it one of the data sets (e.g., datapc1, datapc2) and it will "learn" the digits, for example.
`nnclass.c`	Using learned weights, this neural network will attempt to classify objects. It uses a pre-processed data set, but that can be changed.

`nncvt.c`	Converts an image (glyph, for instance) into data that can be used by the neural net, as configured (i.e., creates 48 input data points from arbitrarily sized pixel data).
`recc.c`	Digit classifier that uses convex deficiencies. Input is a digit image (as can be found in this directory). It tells you what digit it is. There is a huge amount of interesting/useful code in here, including 4- and 8- neighbor counts, area and perimeter, object-oriented bounding boxes, basic geometry, and more.
`recp.c`	Digit classifier that uses the object outline. Input is a digit image (as can be found in this directory). It tells you what digit it is.
`recv.c`	Digit classifier that uses the vector template algorithm. Input is a digit image.
`sig.c`	Calculates the angle-distance signature of a bi-level image.
`lib.h`	Needed include file.
`vect.c`	Font outline vectorizer.
`datapc1`	Neural net input data; images 6x8 pixels as floats between 0-1.
`datapc2`	Neural net input data; images 6x8 pixels as floats between 0-1.
`helv.db`	Templates for recognizing Helvetica font characters
`prof.db`	Templates for character recognition.
`tr.db`	Templates for `ocr3.c`.
`B.pbm`	Letter *B* glyph.
`C.pbm`	Letter *C* glyph.
`D.pbm`	Letter *D* glyph.
`D2.pbm`	Letter *D* glyph.
`D3.pbm`	Letter *D* glyph.
`digi.pbm`	... where *i* is 0, 1, 2, ... 9. A handprinted digit glyph (e.g., `dig0.pbm` is the digit 0).
`paged.pbm`	Sample paragraph of Time-Roman text.
`sample14.pbm`	Sample paragraph of printed text.
`sample.pbm`	Sample paragraph of Helvetica text.
`pagec.pbm`	Training text for Time-Roman.
`sk10.pbm`	Text, skewed 10 degrees.
`sk15.pbm`	Text, skewed 15 degrees.

`text14.pbm`	Grey-level image of a paragraph.
`testpage14.pbm`	Grey-level image of training template.
`testpage.pbm`	Training template.
`testtext.pbm`	Two paragraphs, bi-level.
`weights.dat`	Stored neural net weights.

9.10 References

Aarts, A., and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. New York: John Wiley & Sons, 1989.

Ansari, N., and K. Li. ''Landmark-Based Shape Recognition by a Modified Hopfield Neural Network.'' *Pattern Recognition* 26 (1993): 531–542.

Askoy, S., M. Ye, L. Schauf, M. Song, Y. Wang, R. M. Haralick, J. R. Parker, J. Pivovarov, D. Royko, C. Sun, and G. Farneback. ''Algorithm Performance Contest'' *Proceedings of the International Conference on Pattern Recognition ICPR'2000*, Barcelona, Sept 3–8, 2000.

Baird, H. S. ''The Skew Angle of Printed Documents'' *Proceedings of the Conference of the Society of Photographic Scientists and Engineers. SPIE*. Bellingham, WA, 1987.

Ballard, D. H., ''Generalizing the Hough Transform to Detect Arbitrary Shapes.'' *Pattern Recognition* 13, no. 2 (1981): 111–122.

Bayer, T. A., and U. Kressel. ''Cut Classification for Segmentation.'' *Second International Conference on Document Analysis and Recognition ICDAR*. Tsukuba, Japan (1993): 565–568.

Black, D. *The Theory of Committees and Elections*. Cambridge: Cambridge University Press, 1958.

de Borda, J. *Memoire sur les Elections au Scrutin*. Paris: Histoire de l'Academie Royale des Sciences, 1781.

Brams, S. J., and P. C. Fishburn. *Approval Voting*. Boston: Birkhauser, 1983.

Bulis, A., R. Almog, M. Gerner, and U. Shimony. ''Computerized Recognition of Hand-Written Music Notes.'' *Proceedings of the International Computer Music Conference*, 1992, 110–112.

Carpenter, G. A., and S. Grossberg. ''The Art of Adaptive Pattern Recognition by Self-Organizing Neural Network,'' *Computer* 21, no. 3 (1988): 77–88.

Claus, V., H. Ehrig, and G. Rozenberg (eds.). *Graph Grammars and Their Applications to Computer Science and Biology*. (Third International Workshop) Lecture Notes in Computer Science 72. Berlin: Springer-Verlag, 1978.

Duda, R. O., and P. E. Hart. ''Use of the Hough Transform to Detect Lines and Curves in Pictures.'' *Communications of the ACM* 15, no. 1 (1972): 11–15.

Ehrig, H., H. Kreowski, and G. Rozenberg (eds.). *Graph Grammars and Their Applications to Computer Science*. (Fourth International Workshop) Lecture Notes in Computer Science 291. Berlin: Springer-Verlag, 1900.

Enelow, J. M., and M. J. Hinich. *The Spatial Theory of Voting: An Introduction*. Cambridge: Cambridge University Press, 1984.

Fahmy, H., and D. Blostein. ''A Graph Grammar Programming Style for Recognition of Music Notation.'' *Machine Vision and Applications* 6 (1993): 83–89.

Farquharson, R. *Theory of Voting*. New Haven, CT: Yale University Press, 1969.

Fletcher, L. A., and R. Kasturi. ''A Robust Algorithm for Text String Separation from Mixed Text/Graphics Image.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10, no. 6 (1988): 910–918.

Freeman, J., and D. Skarpura. *Neural Networks — Algorithms, Applications, and Programming Techniques*. Reading, MA: Addison-Wesley, 1991.

Fukushima, K., S. Miyake, and T. Ito. ''Neocognitron: A Neural Model for a Mechanism of Pattern Recognition.'' *IEEE Transactions on Systems, Man, and Cybernetics* 13 (1983): 826–834.

Gonzalez, R. C., and R. E. Woods. *Digital Image Processing*. Reading, MA: Addison-Wesley, 1992.

Hashizume, A., P. S. Yeh, and A. Rosenfeld. ''A Method of Detecting the Orientation of Aligned Components.'' *Pattern Recognition Letters* 4 (1986): 125–132.

Hinds, S. C., J. L. Fischer, and D. P. D'Amato. ''A Document Skew Detection Method Using Run Length Encoding and the Hough Transform.'' *Proceedings of the 10th International Conference on Pattern Recognition*. Atlantic City, 1990: 464–468.

Ho, T. K., J. J. Hull, and S. N. Srihari. ''Decision Combination in Multiple Classifier Systems.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16 (1994).

Holt, C. M., A. Stewart, M. Clint, and R. Perrot. ''An Improved Parallel Thinning Algorithm.'' *Communications of the ACM* 30, no. 2 (1987): 156–160.

Hough, P. V. C. ''Method and Means for Recognizing Complex Patterns.'' *U.S. Patent #3,069,654* (1962).

Huang, T. S. ''A Fast Two Dimensional Median Filtering Algorithm.'' *IEEE Transactions on Acoustics, Speech, and Signal Processing* 27 (1979): 13–18.

Kato, H., and S. Inokuchi. ''A Recognition System for Printed Piano Music Using Musical Knowledge and Constraints.'' *Structured Document Image Analysis*, ed. Baird et al. Berlin: Springer-Verlag, 1992.

Kimura, F., and M. Shridhar. ''Handwritten Numeral Recognition Based on Multiple Algorithms.'' *Pattern Recognition* 24 (1991).

Liang, S., M. Ahmadi, and M. Shridhan. ''Segmentation of Touching Characters in Printed Document Recognition.'' *Second International Conference on Document Analysis and Recognition*. Tsukuba, Japan, 1993: 569–572.

Lu, Y. ''On the Segmentation of Touching Characters.'' *Second International Conference on Document Analysis and Recognition*. Tsukuba, Japan, 1993: 440–443.

Masters, T. *Advanced Algorithms for Neural Networks — A C++ Sourcebook*. New York: John Wiley & Sons, 1995.

Mui, L., A. Agarwal, and P. S. P. Wang. ''An Adaptive Modular Neural Network with Application to Unconstrained Character Recognition.'' *International Journal of Pattern Recognition* 8, no. 5 (1994): 1189.

O'Gorman, L. ''Image and Document Processing Techniques for the RightPages Electronic Library System.'' *Proceedings of the International Conference on Pattern Recognition*. Los Alimitos, CA, 1992: 260–263.

Parker, J. R. *Practical Computer Vision Using C*. New York, NY: John Wiley & Sons, 1994.

Parker, J. R. ''Recognition of Hand Printed Digits Using Multiple Parallel Methods.'' *Third Golden West International Conference on Intelligent Systems*. Las Vegas, June 6–9, 1994.

Reed, T. ''Optical Music Recognition.'' MSc thesis, University of Calgary Department of Computer Science, 1995.

Shang, C., and K. Brown. ''Principal Features Based Texture Classification with Neural Networks.'' *Pattern Recognition* 27 (1994): 675–687.

Shridhar, M., and A. Badrelin. ''Recognition of Isolated and Simply Connected Handwritten Numerals.'' *Pattern Recognition* 19, no. 1 (1986).

Shridhar, M., and A. Badrelin. ''Context-Directed Segmentation Algorithm for Handwritten Numeral Strings.'' *Image and Vision Computing* 5, no. 1 (1987).

Straffin, P. D. Jr. *Topics in the Theory of Voting*. Boston: Birkhauser, 1980.

Touretzky, D., (ed.) *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan-Kaufmann, 1989.

Tsujimoto, S., and H. Asada. ''Resolving Ambiguity in Segmenting Touching Characters.'' *First International Conference on Document Analysis and Recognition*. San Malo, CA, 1991: 701–709.

Wilkinson, T. and J. Goodman. ''Slope Histogram Detection of Forged Signatures.'' *SPIE Conference on High Speed Inspection, Barcoding, and Character Recognition* 1384 (1990): 293–304.

Xu, L., A. Krzyzak, and C. Y. Suen. ''Methods of Combining Multiple Classifiers and Their Application to Handwriting Recognition,'' *IEEE Transactions on Systems, Man, and Cybernetics* 22, no. 3 (1992).

Yong, Y. ''Handprinted Chinese Character Recognition via Neural Network,'' *Pattern Recognition Letters* 7 (1988): 19–25.

Zhang, Y. Y. and C. Y. Suen. ''A Fast Parallel Algorithm for Thinning Digital Patterns,'' *Communications of the ACM* 27, no. 3 (1985): 236–239.

CHAPTER

10

Content-Based Search—Finding Images by Example

10.1 Searching Images

Google has made a reputation by allowing users to search the Internet using text. A simple query, a few words typed into a box in a window, sets in motion complex algorithmic wheels that result in hundreds or thousands of relevant matches—web pages that not only contain the words but that reflect the meaning of those words in their content. Searching for text is a simple thing, though. One finds exact matches to the text or matches to slight variations in the words. Searching for ''house'' results in matches to ''souses,'' too, and probably to ''homes.'' One would simply relate a word to a list of words that are related except in tense or plural or quantity. Searching for images is harder.

There are two ways to search for images: to specify a set of words (text) that describes the desired image, or to give an example image and ask for others like it. An example of the first case is Google Images. A search for ''radio'' using Google Images results in various pictures of radios; as of this writing, of the 20 images on the first page of results, 18 are radios, 1 is a microphone, and 1 is a cartoon of a radio announcer. These images have been categorized by people and given text labels, and the text labels are what is matched in the search. In other words, the process is neither automatic nor spontaneous. For a new image to be recognized by this system, it would need to be classified by a person first. The number of new images each day on the Internet would argue against this being a successful strategy in the long run, and it would not be useful for individual collections of photographs on home computers.

Searching for images by example is quite different in utility and in process. In this case, a query consists of an image (or images), and the question is, ''*Can you show me more images like this one?*'' The implication is two-fold: first, that the system can find similar images using the nature of the image contents; second, that the system can determine what is meant by ''*like this one.*'' In a landscape image, what is it that the person is searching for? Mountains? Sky? The black car parked by the power pole? This is rarely well defined by the image, and is why multiple images might form a better query. Figure 10.1 shows a typical example of what shall be called *query by example* (QBE) but is also called *content-based image retrieval* (CBIR). An image, the one labelled ''target,'' is given to the system. The remaining images are samples of those in the set being searched. A measure of similarity between the target and all other images is computed, and the images that are the most similar to the target are presented to the user as possible matches.

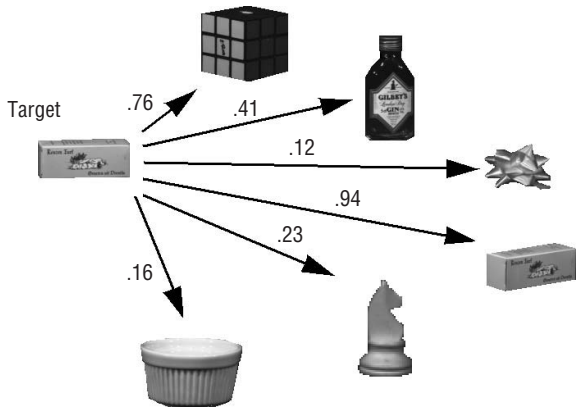


Figure 10.1: Query by example. The target image is evaluated as similar to others based on a set of features (measurements). The most similar ones are matches. Note that the best match is still not perfect.
Images from ALOI data set [Geusebroek, 2005]

So, searching images using content involves using computer vision techniques and basic statistics to characterize images. Images are similar if they have a lot of these characterizations in common, or nearly so. Which of these characteristics is best for image matching? It depends on the images.

10.2 Maintaining Collections of Images

A practical issue for QBE systems is the question of how the images are stored, labelled, and measured. An essential aspect of a working QBE is a capability to scan through directories looking for image files, because examining gigabytes of data manually is really impossible. To make a query, an image is specified

as the target, and this image can be anywhere. It makes no sense to search the entire computer for images that may match the target every time a query is made. A better (more efficient = faster) idea is to maintain a collection of images that can be searched, and to keep them in a fixed, known location.

The collection presented here is presumed to be located in the PC directory `C:\AIPCV\imagedata`, in sub-directories named `1` through `1000`. This file organization is arbitrary and can easily be changed to suit a particular image set or application. Each sub-directory contains a set of images that are related to each other. Sets of image data can be downloaded from the Internet, and the set that is used here as an example is the ALOI (Amsterdam Library of Object Images) data set [Geusebroek, 2005], which is downloadable at no cost from `http://staff.science.uva.nl/~aloi/`. In this set, each sub-directory contains images of the same object, rotated by a known angle. The backgrounds are fairly uniform and are black, unusual for family photos but good for testing algorithms because there is no background clutter. Collecting large numbers of images for testing is a time-consuming and boring process, so downloading an existing collection is suggested instead.

Once the collection has been downloaded and installed, it is a good idea to build a master data file that contains path names to all files in the collection and data connected with the files, such as values of features to be used in matching. Doing so ensures that image files will not be involved in the initial search. The master file is text and can be edited manually, if desired. Individual files can be added as needed, or the entire set can be processed automatically.

So, what is needed is a simple program that can (1) examine all files in a directory and sub-directories and build a master text file, and (2) a program that can build a master feature file containing all file path names and features connected with them. The structure of a directory is complicated and is not generally machine-independent, but what is most needed from that structure is actually pretty simple.

1. Open a directory. The call `opendir` does this, such as:

   ```
   dir = opendir ("c:\\AIPCV\\");
   ```

   If the name passed as a parameter is not a directory, then the function will return an error (`dir == NULL`); otherwise, the variable `dir` will be a file descriptor that points to the open directory.

2. The function `readdir` returns a structure that indicates certain useful facts about the next directory entry; this could be a file or another directory. For example:

   ```
   ent = readdir (dir);
   ```

   returns the next entry; `ent==NULL` means all entries have been read. Otherwise, it points to a structure within which the field `d_name` is the name of the entry (file).

3. Like a file, the directory needs to be closed. The function `cloedir(dir)`, where `dir` is the variable returned by the call to `opendir` previously, will do this.

So, that's all that is needed. As an example that uses this basic scheme, the following is a simple piece of code that prints all the image files within the `AIPCV` directory:

```
dir = opendir ("c:\\AIPCV\\");
if (dir != NULL) {

/* print all the files and directories within directory */
 while ((ent = readdir (dir)) != NULL)
 {
   if (ent->d_name[strlen(ent->d_name)-4] == '.')
   {
     str = (char *)(&ent->d_name[strlen(ent->d_name)-3]);
     lowerCase (str);
     printf ("File type is %d ", ent->d_type);
     if ( strcmp(str, "jpg")==0 )
     printf ("File '%s' is JPEG.\n", ent->d_name);
     else if ( strcmp(str, "png")==0 )
       printf ("File '%s' is PNG.\n", ent->d_name);
     else if ( strcmp(str, "bmp")==0 )
       printf ("File '%s' is BMP.\n", ent->d_name);
     else if ( strcmp(str, "ppm")==0 )
       printf ("File '%s' is PPM.\n", ent->d_name);
     else if ( strcmp(str, "pgm")==0 )
       printf ("File '%s' is PGM.\n", ent->d_name);
. . .
    }
  }
  closedir (dir);
} else
{
  /* could not open directory */
  perror ("");
  return 1;
}
```

This code does not process images, but it is essential to a content-based search program. What we want is a file that contains the names of all the images that are to be searched. The alternative to scanning directories automatically is to build a list of image files by hand, looking through directories for images and typing the names into a single file that will be used to open them later. On a modern terabyte drive, this is impractical.

10.3 Features for Query By Example

The problem to be solved is to find images in a collection, the *search set*, that are similar to one or more images presented as examples. Objects can be combined in various ways to form an image, and in almost all cases it is the nature of the objects that is used to determine similarity between images. Sadly, other uncontrollable factors also affect similarity; the illumination, orientation, scale, and noise can all have a serious impact on how the same set of objects appear in an image. So, how can an image query be conducted? The answer is, for the moment, rather crudely.

The commonly used techniques at present usually involve extracting features from the target image and comparing the values of those features against those extracted from the images in the search set. This can be done quickly if the search set has been processed in advance by having all the features extracted and tabulated, because it means that only the target image has to be examined at the time of the query. This won't work for searching Web images, but is feasible for individual data sets and personal images.

A large part of the literature on QBE and content-based search suggests the use of color features. This is reasonable, since most images these days are color images. Modern digital cameras create color images, and their prevalence has resulted in the largest part of the explosion of image data seen over the past decade. So-called *black and white* photographs are now taken largely as artistic expression, and many archived black and white images have already been classified manually over the years. So, to begin a study of QBE, it makes sense to look at the use of color as a way to match images.

10.3.1 Color Image Features

There are many ways to use color as a measure of image similarity.

For example, histogram-based techniques have been used for other applications (e.g., [Parker, 1997]), and color histograms have been used in various ways for QBE in a lot of past work [Hafner, 1995; Niblack, 1993; Tico, 2000]. A simple idea is to use a quantization scheme to reduce the number of colors in the image, and then to count how often each color occurs (number of pixels of that color). This is a simple characterization of the image that can be compared with others.

Color reduction can be used most obviously to eliminate candidates from consideration. If the target image does not contain any green at all, then any images containing green cannot be a good match. Images can be matched

according to the amount of various colors present in the image. Histograms of red, green, blue, yellow, and a gamut of colors can be created for all images. An image will match itself perfectly according to this scheme, and will match other images less well according to color content. It is easily possible to have two images that match perfectly according to this scheme, but it would be rare.

Of course, some images do not have color, and so we may have to apply these histogram methods to simple grey levels, counting the number of pixels having each grey value. In most cases, color images can be converted into grey scale without the loss of region or shape information. Black and white movies and television were, after all, the state of the art many years ago, and it was always possible to see clearly what was happening; scenes and objects can be distinguished without using color.

10.3.1.1 Mean Color

It is a simple thing to determine the average of the R, G, and B value over all image pixels. It may not be a compelling measure of similarity between two images, and is only one numerical value, so it is likely to be crude. Still, it may be a good negative (allows the rejection of some obviously non-similar images) and is simple to calculate. The overall mean color value in the image is a feature, and it will be called the *mean* feature.

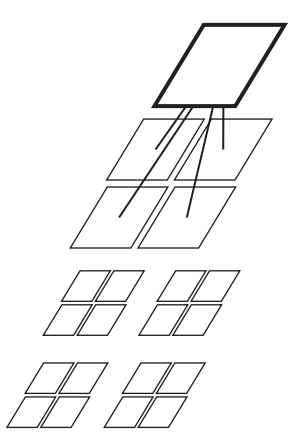
10.3.1.2 Color Quad Tree

A *quad tree* is a tree data structure in two dimensions. Starting at the first level, the one representing the entire image, the tree has one node. The next level down the tree has four nodes, one for each of the four equal-sized quadrants of the image. The next layer splits each of those quadrants into four more parts, and so on, until the final level of the tree, at which point the individual pixels are represented by each node. Figure 10.2 shows a diagram of a quad tree, including what various levels of the tree look like as images. Level 0 is the root of the tree and contains the average color over all quadrants/pixels. Level 1 has four parts, level 2 has 16, and so on.

There are many ways that a quad tree can be used to compare images for similarity. For example, the trees of the images being compared could be examined to determine how many levels deep an agreement could be seen, in terms of color or some other features. At some parts of the image, the trees would match better than others, perhaps, indicating a spatial property in the similarity. However, much simpler is to build a vector of the colors seen at a particular level and use that as a feature. The zero level has one color, and that will be the overall image mean. At level 1 there are four sub-images, each with

an average color. This gives a vector of four colors, or 12 components, and will be referred to as the *quad* feature.

The next level down, level 2, has 16 colors, or 48 components. This may give a better match, and so will be called the *hex* feature. Going to too many levels ultimately makes the match too specific, and so there is a point of diminishing returns reached at this level or the next.



```
struct quadtree
{
        struct quadtree *next[4];
        int uli, ulj, lri, lrj;
        float r, g, b;
};
```

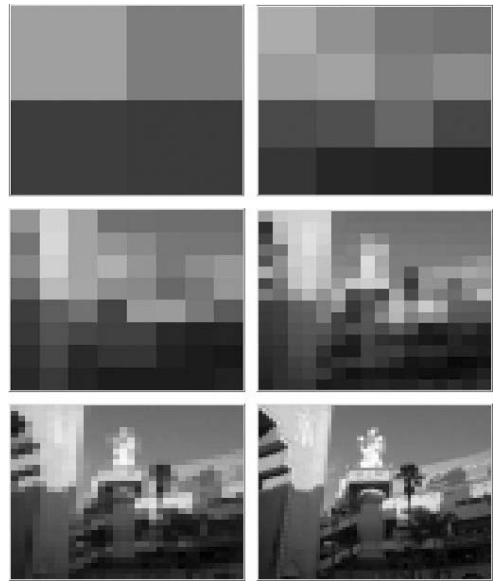


Figure 10.2: A quad tree breaks an image into four parts, and then breaks those parts into four parts, and so on, down to the individual pixel level. The images on the right show the quad tree of an image of downtown Hollywood rendered as images, starting at level 1 (4 parts) to level 6.

Most of this is beside the main point, which is that even a ''function'' as simple as a simple numeric constant can be used as a basis for a transformation of this type. The idea of using wavelets as a basis is not really new, although there has been a lot of interest recently in using them as applied to images, especially for compression. A *wavelet* is simply a function that, unlike the Fourier transform, not only has a *frequency* associated with it, but also a *scale*.

10.3.1.3 Hue and Intensity Histograms

Using quad trees as color features connects color values with spatial coordinates in a general, if crude, way. Histograms are purely a measure of the color content over an image, and they can be made more accurate. Each bin in the

histogram represents a range of colors, and the bins can be made as large or small as needed. RGB values are awkward for histogramming purposes, since they form a three-dimensional vector. A scalar is much better, and so hue might work; so might the intensity (grey level).

The OpenCV function `cvCvtColor` can convert an RGB image into an HSV image. Hue will be the first value in the three part `CvScalar` pixel value of such an image, and takes on a value between 0 and 179 (half of its full range of 0-360 degrees). Building a hue histogram `H` is a simple matter of finding the hue component `k` of each pixel and incrementing that bin: `H[k] += 1`. The histogram must be normalized or images of different sizes cannot be compared. Normalization involves dividing each bin by the sum of all bins `S`, which in most cases will be the same as the number of pixels.

Computing a hue histogram takes only about a dozen lines: convert the image to HSV, zero the histogram, visit all pixels and increment the bins, and divide by the total number of pixels. The following code also ignores grey pixels, which have little or no hue:

```
cvCvtColor( img, hsv, CV_BGR2HSV );

for (i=0; i<180; i++) hhisto[i] = 0;
for (i=0; i<hsv->height; i++)
       for (j=0; j<hsv->width; j++)
       {
             s = cvGet2D (hsv, i, j);
             if (isGrey(s)) continue;  // no hue
             k = (int)s.val[0];
             hhisto[k] += 1; n++;
       }
for (i=0; i<180; i++) hhisto[i] /= n;
```

Now all that is needed is a way to compare two histograms to each other.

10.3.1.4 Comparing Histograms

Two histograms, H1 and H2, are equal if H1[i] = H2[i] for all legal values of i. If they are not equal, how near to being equal are they? And, more importantly, which one is most similar to a target histogram H3?

Histograms are *N*-dimensional arrays, and can be treated like vectors. It is possible to compute a Euclidean distance between two histograms in the same way that a norm is computed for a vector. This is actually how a lot of software computes histogram similarity — as an *N*-dimensional distance between the histograms, calculated bin by bin. For example, the following code is a typical way to implement this:

```
for (i=0; i<N; i++) sum += (H1[i]-H2[i])*(H1[i]-H2[i]);
d = sqrt(sum);
```

It often works, too, but misses a key point: bins near each other in a histogram represent data in the image that is similar. Adjacent bins represent similar colors or grey levels. In a typical vector, adjacent ''bins'' represent orthogonal dimensions, and are unrelated.

What would be a better way to compare histograms? Similarity in nearby bins needs to be acknowledged, while differences need to be penalized. One way is to compare *cumulative* histograms, in which the value of a bin is the sum of the bins in the regular histogram to that point. If H is a histogram, then a cumulative histogram C is determined as follows:

```
C[0] = H[0];
for (i=1; i<N; i++) C[i] = C[i-1] + H[i];
```

In a normalized histogram, the sum over all bins will be 1. In a cumulative histogram, the bin values approach 1 as the bins are examined in increasing order. A difference between two histograms early on (i.e., low bin indices) sums repeatedly over successive bins until it equalizes, meaning that differences in distant bins have a greater impact than differences in nearby bins. This is what is wanted.

Does Euclidean distance on cumulative histograms (*hueC*) work better (i.e., higher success rate) than using simple histograms (*hue*)? A simple experiment using the code on the website (`searchCM.c`) says ''yes,'' but not too much. Searching for a sample of 1801 images in the ALOI set of over 700,000 using both methods yields:

- **Simple histogram** — 65.9% success
- **Cumulative histogram** — 69.1% success

10.3.1.5 Requantization

All color images consist of pixels having red, green, and blue components, but obviously images contain more colors than just those three. Consider the rainbow; Newton labelled his basic colors red, orange, yellow, green, blue, indigo and violet. Adding white and black for a total of nine *prototype* colors allows a reasonable range of colors for a matching process. Any image can be processed so as to contain only these colors, and in such a way that pixels take on the value of the prototype that is closest to their own color. Section 1 was an initial discussion of this concept in the context of segmentation. Here the discussion, and hence the prototypes, centers about separating matching images from non-matching ones.

Newton's colors are interesting historically, but they contain too much blue and do not take into account distances between colors. The prototypes

should be, as far as is possible, equal distances from each other. The ones used here will be:

Red	(170, 0, 0)
Orange	(170, 85, 0)
Yellow	(170, 170, 0)
Green	(0, 170, 0)
Blue	(0, 0, 170)
Purple	(85, 0, 170)
Pea	(85, 170, 0)
Black	(25, 25, 25);
White	(240, 240, 240)
Grey	(128, 128, 128)

When using these prototypes, the colors in the image are replaced and a histogram of the ten colors is constructed and normalized. This is compared with other such histograms in the data collection, and the nearest one in the Euclidean sense is the best match. This will be referred to as the *proto* feature.

10.3.1.6 Results from Simple Color Features

All these features, computed for the entire ALOI data set, were calculated and stored in one large text file, `master3.txt`. Then target images were selected from the data set at random and a query was launched: which images in the set were the best matches for the target? Of course, the image itself is in the set, and should be a perfect match, but which others would match the target? An important result is how often an image in the same class was a match; all images of the same object reside within the same directory, so it is easy to tell when such a match occurs. Any images in the same directory as the target will be considered a match.

The results are given in Table 10.1. This can be thought of as a basic comparison of the color features for effectiveness in queries. The overall results are encouraging, but some provisions should be mentioned. The ALOI data set has 1000 objects viewed on a black background. Each object is imaged from 72 distinct points on a circle with the object at the center (every five degrees). The consequences are that there will be a few images that are very similar to any target in the set because a five degree orientation change is rarely going to make a huge difference in what can be seen. Also, the large

amount of black is a blessing and a curse: the backgrounds are constant, but the large number of black pixels can make scale a feature by default. That is, small objects will have more black in them, and large objects will have less. This does not represent a true property of the object, since the size within the image can be changed when the image is captured. Small objects will match other small objects because of the number of black pixels in the histograms and other color measures. Thus, in the histograms the color black is not included.

Table 10.1: Results for Searching Experiments Using Color Features

FEATURE	0	1	2	3	4	5	6	7	8	9	10	TOTAL	%
Mean	0	168	156	197	130	145	127	151	122	163	442	1005	55.8
Quad	0	42	40	144	97	141	124	140	146	143	784	1337	74.2
Hex	0	5	3	48	42	96	88	84	126	102	1207	1607	89.2
Proto	5	592	203	178	174	145	144	123	108	129	373	877	48.7
Hue	0	95	124	129	141	126	114	130	129	173	640	1186	65.9
HueC	0	78	106	119	123	134	115	132	135	176	686	1244	81.0

A vote was taken over the first five methods for each trial to get an overall result. A majority vote would need $\frac{3}{5}$ for a winner. The results, out of five methods over all images, is:

Votes	0	1	2	3	4	5
	41	92	209	327	486	646
	Total correct = 1459 (81.0%)					

In the Table 10.1, 1801 images have been selected from the total set to be targets for a search. The class of those targets is known, and the numbers in each column indicate the number of times a search yielded the correct target: an image from the same class (directory). Each search selected the best ten matches to a target image and uses those to determine the class of the target. Each column indicates how many times a specific number of matches were obtained, 0 through 10. In a majority vote, any more than 5 matches that agreed with each other would be a definitive class indicator, so the success column shows how many times that happened, and what that is as a percentage of the total number of trials. This can be called the *success rate*, as before.

The success rate is more often called *precision* in information retrieval circles. It is defined as the number of relevant documents retrieved by a search

(successes) divided by the total number of documents retrieved by that search. Another measure used in information retrieval is *recall*; this is defined as the number of successful retrievals divided by the total number of existing relevant documents (which should have been retrieved). The trials being done here only perform ten retrievals, so recall cannot be measured.

Recall is a measure of completeness, whereas precision can be seen as a measure of accuracy or exactness. 100% precision means that the result of every search was relevant. 100% recall means that every relevant document was retrieved.

A single number that combines both recall and precision is the *F-score*, which is defined as:

$$F = 2.0^{*} (\text{recall}^{*}\text{precision})/(\text{recall} + \text{precision})$$

This is called the *F1 measure* because others exist that weight recall differently from precision. In the literature, recall and precision are shown as a graph, giving the relationship between the two. It is sometimes hard to compare graphs, so F1 can be useful in these cases.

A scheme that is not common but that is well understood by non-specialists is the *search engine* evaluation scheme. When a web search engine, such as Google, is given a query, the resulting responses are ranked according to relevance, and are returned and displayed in that order. The number of responses on the page is q, frequently 10 or so, but $q = 30$ is not too unusual. There are many ways of reporting success in this kind of enterprise. We are suggesting that success is *the percentage of relevant responses on the first page of a typical query*. This is certainly a measure of success that would be quickly understood by anyone who uses web search engines frequently. Out of the ten responses reported on the first page of a response to a query, how many of them are really a match? When asked this question of text-based queries, the average person would accept 3 successes (30%), which they think of as typical; this is based on a casual poll of students and university staff.

Each data set has limitations with respect to its use in testing of retrieval methods that should be understood. In photos of scenery and portraits, the background is actually a part of what is being searched. Sky = blue, grass = green, and these will appear prominently in outdoor images. Sometimes the match will be to these things rather than to, for instance, the car in the middle of the picture. That issue is not present in this (ALOI) data set, and some may say it is ''cheating'' to do things this way. It is cheating only if it is not pointed out. In addition, some software developers will point out that in outdoor scenes it is reasonable that matches occur to other outdoor scenes at random, as the content is ''outdoors.'' It is important to understand the limitations of any method, and to understand why certain matches take place the way they do. Perfection in such searches can be achieved only by observing the results of the various algorithms and adjusting the code and parameters to

improve the results. It is natural that each person will have a different set of images, and that a different mix of methods would work for that set than for others.

10.3.1.7 Other Color-Based Methods

A technique described in the literature [Tico, 2000] for creating color histograms has the rather desirable property that it disregards achromatic information, often included as noise in other types of color histogram. This is accomplished by calculating the standard deviation of the red, green, and blue components of a color pixel and then normalizing to the range [0,1]. As a reminder, the standard deviation is:

$$\sigma = \sqrt{\sum_{i=1}^{n} \frac{(x - \overline{x})^2}{n}} \qquad \text{(EQ 10.1)}$$

In fact, any measure of variation between the R, G, and B components would probably work (e.g., average difference, bitwise AND), but standard deviation is well known and easy to calculate.

Now the *chrominance* of a pixel is found using a piecewise linear function:

$$\mu(\sigma) = \begin{cases} 0 & \text{if } 0 \le \sigma < a \\ 2\left(\dfrac{\sigma - a}{b - a}\right)^2 & \text{if } a \le \sigma < \dfrac{a+b}{2} \\ 1 - \left(\dfrac{\sigma - b}{b - a}\right)^2 & \text{if } \left(\dfrac{a+b}{2} \le \sigma < b\right) \\ 1 & \text{if } b \le \sigma < 1 \end{cases} \qquad \text{(EQ 10.2)}$$

where a and b are constants between 0 and 1, and $a < b$. In past experiments that were conducted, effective values of a and b were determined to be $a = 0.05$ and $b = 0.8$. These values are empirical, and it may be that further work is needed to do better. The chrominance values were calculated for each pixel in the region and were used to construct a color histogram with 16 bins.

The use of the mean color value worked better than expected (55%) and perhaps more could be made of the fundamental idea. An underlying principle is that of *moments*, which are polynomials of increasing order that describe the shape of a statistical distribution. The first-order moment is:

$$\sum_{i=1}^{n} (x - \overline{x})^1 \qquad \text{(EQ 10.3)}$$

The exponent ''1'' is left in to emphasize the fact that the order of the moment is indicated by this exponent. This is nearly the formula for the mean; the mean

is the first-order moment divided by n, or the amount of first-order moment per pixel.

The second-order moment would be:

$$\sum_{i=1}^{n}(x-\overline{x})^2 \qquad \text{(EQ 10.4)}$$

which is the numerator of the variance formula. The variance is the second-order moment per pixel. There is a third-order moment (the skewness), fourth (kurtosis), and so on. The point is, if the mean (first-order moment) is useful in searching for images, then why not the others? One idea is to simply use the mean, standard deviation, and skewness values of the color components over the image as a nine-component feature vector [Stricker, 1995]. A simple experiment (`searchCM` on the website) uses a weighted feature vector having these nine components, but because the mean has a larger scale than standard deviation, which in turn is larger than skewness, the three parts are normalized differently before combining them into a vector. The result, over a run of 654 random searches, is 586 successful retrievals (89.6%). This is better than any of the histogram-based methods on this data set!

All important content-based searching systems use color as a key feature, and almost all use some variation on color histograms. Such methods are at the heart of working systems today, although success is still somewhat less that is to be desired.

10.3.2 Grey-Level Image Features

Searching in sets of grey-level images is not that important from the perspective of adding functionality to a system. Most users would not search for grey images. It is important because color is simply not enough information when conducting a search. A search for red Mustang automobiles would not produce any blue or white ones, and it seems obvious that what is being searched for is the kind of car, not its color. Certainly some objects are clearly classified by color: the sky is normally blue, trees are green, snow is white. On the other hand, dogs, cats, cars, houses, and a huge variety of other objects come in many colors. Can we search for these?

There are methods that can be used for grey-level images, and these should generalize to color images, too. The RGB components of a color image can be averaged to give a grey level. Thus, techniques discussed here can be added to those that use color to yield a more robust composite scheme for content-based searching.

The website includes a directory, `c:\aipcv\grey`, where grey images can be placed for searching, and the code provided with this book looks there for the data set. The ALOI collection has sets of grey images that can be used for testing, and can be downloaded from `http://staff.science.uva.nl/ ~aloi/`.

10.3.2.1 Grey Histograms

A grey-level histogram is a simple modification of a color histogram, and the methods used for comparing and using color histograms all apply here. Because grey level is only a single measure (unlike color, which is a 3D vector), it might not be as discriminatory. Also, a slight shift of all grey levels in an image (i.e., brightening or a linear contrast shift) will not change the image content but will significantly affect the average level and the histogram.

A basic grey-level histogram over the entire image would have 256 bins, although it is possible that decreasing the number of bins could be accomplished while not affecting the results too much. A quad tree could be used to build a set of grey levels over hierarchical sub-images, as was done with color quad trees. Using the first level gives four levels to be compared, and using the second level gives sixteen levels, as before. The results, expressed as a success rate (how often the correct image class was matched to a target) are:

- **Level 1 quad tree** — $292/654 = 44.6\%$
- **Level 2 quad tree** — $564/654 = 86.2\%$
- **Basic grey histogram** — $586/654 = 89.6\%$
- **Cumulative grey histogram** — $586/654 = 89.6\%$

10.3.2.2 Grey Sigma – Moments

Sigma (σ) or standard deviation has been described as a simple texture metric, but really it is just a measure of the variability of the brightness as captured in the pixel values. The intensity variation across a region is determined by calculating the standard deviation of the intensity values of all the pixels. As discussed in Section 10.3.1.7, this is the second-order moment; a vector feature consisting of three or four moments can be built quite easily and used as a means of measuring image similarity.

A basic system that used mean, standard deviation, and skewness and calculated a weighted Euclidean distance to determine similarity gave the correct image class 113/654 times, or 17.3% of the time. This corresponds to one or two hits per web search, for example — not too bad for such a simple calculation. As a basis for comparison, using the ALOI data set (720,000 images in 1000 categories) the random level of success is 1/1000, or 0.1% success.

10.3.2.3 Edge Density – Boundaries Between Objects

Edge density is a simple geometric measure based on the strength of the edges in an image region or in the whole image. It can be found by first using a standard edge detector (e.g., Sobel, Section 2) to enhance the pixels that belong to edges and boundaries. The result is a set of pixels whose values represent

the strength of the edge at that point. Pixels far from an edge are 0, and those near an edge increase to a maximum value. The edge density measure is calculated as the mean pixel value of the edge enhanced image.

Why is this a valid feature for comparison? Because edges represent the boundary between objects and the background, or between objects. In either case, they are located where objects exist in the image, and so are connected with content. This feature represents a measure of how ''busy'' the image is.

10.3.2.4 Edge Direction

Many edge detectors, including the Sobel edge detector, can be implemented as a convolution of a small (e.g., 3x3) image mask, and often multiple masks, each with a directional bias. This allows a crude estimate of edge direction to be made. In particular, for a typical 3x3 region in an image, the two Sobel masks are:

$$s_x = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad s_y = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

Given that the response to each mask represents a vector in the X or Y direction, the direction associated with the pixels in the region can be established using simple trigonometry:

$$\theta = \arctan(S_y/S_x)$$

This edge-direction value is used to compute an overall estimate of the direction of the edges in a region by calculating a resultant vector over all pixels in the region. When similarity between images is calculated, it is done using differences between these region-based resultants.

10.3.2.5 Boolean Edge Density

This is, on the face of it, very similar to the edge density method above. After the edge detector has been applied to the image, the image is thresholded so that what could be called ''edge pixels'' are white (1) and non-edge pixels are black. The measure returns the proportion of white (edge) pixels in the region. The difference between this and standard edge density is seen in images with noise. Boolean edge density sometimes allows noise to be minimized; both noise and edges are high frequency information, and the thresholding process in Boolean edge density tends to reduce the effect of noise.

This is the least successful feature seen so far, at least when applied to an entire image. It classifies an image into the correct group only 3/654 times. However, it serves as a useful segue into the next section: when Boolean edge density is measured over 25 equally sized sub-regions of an image

(5x5 grid imposed over the image) and the resulting 25 Boolean edge density measurements are made into a feature vector and used to determine similarity, the results are much better: 361/654 successes, or 55.2%. So, perhaps it is time to look at ways to incorporate spatial concerns into the search.

10.4 Spatial Considerations

Based on prior experimentation (e.g., [Rao, 1999]), it is generally appreciated that statistical measures based on entire images are often less successful in characterizing the image in a search or matching context than using the same measurements based on subdivisions of the same image. This makes a great deal of sense, not the least because in many images the objects of greatest interest are near the center of the image. Simply giving a higher priority to pixels near the center might improve searches, and omitting pixels on the boundaries would certainly make them faster.

When using sub-regions, one or more of the features is measured on each defined region and collected into a large set of sets of features. Based on the work of Rao, five distinct ways of defining sub-regions are apparent: *overall*, *rectangular*, *angular*, *circular*, and *hybrid*. Figure 10.3 illustrates the shape of these regions.

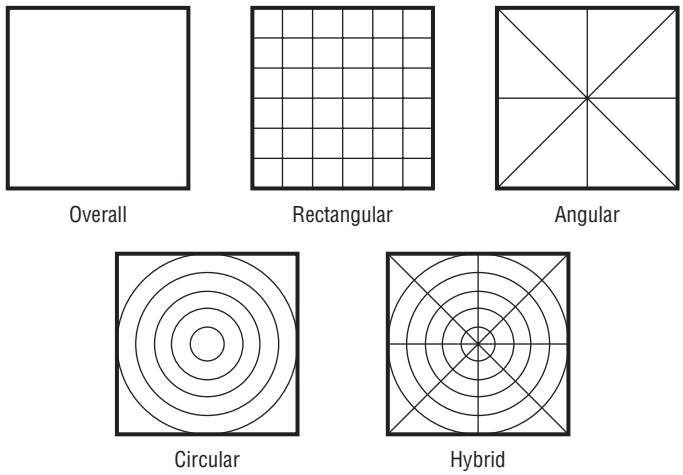


Figure 10.3: The five types of sub-region used in the similarity search process.

10.4.1 Overall Regions

This is the *trivial* region, the entire image considered as a single region. This corresponds with the usual global techniques that have been discussed and evaluated to this point.

10.4.2 Rectangular Regions

This is a first step towards spatial compartmentalization of an image, and is simple to implement because an image is rectangular, and so are the regions. The image is broken into multiple vertical and horizontal parts, and then features are extracted from each of the regions in the grid. For example, if the image is cut vertically four times and horizontally four times, then there will be 25 sub-regions. If this image is 250x500 pixels, then each region is 50x100 pixels, if there is no overlap between the regions. Allowing regions to overlap has the advantage of blurring the boundaries between the regions and allowing some natural variation in position of objects.

- **Advantages** — Fast, easy to implement.
- **Disadvantages** — Not designed with rotation in mind.

Going back to the grey-level edge features, recall that the results for the overall image were lousy: edge density was successful 25/654 times (3.8%), and Boolean edge density was successful only 3/654 times (0.46%). If these features are used over sub-regions, the results improve dramatically. Using 25 edge-density measures in a vector, one for each of a 5x5 set of image sub-regions, edge density is successful 557/654 times (85.2%), edge direction works 583/654 times (89.1%), and even edge density is successful 361/654 times (55.2%). All trials used the ALOI grey-level image set.

These features are based on geometry, not color or grey level, and so can be combined with other histogram-based features to create a highly successful image search engine.

10.4.3 Angular Regions

Angular regions are wedge-shaped sections of the image radiating from the geometric center of the image (not the centroid, which is pixel-based). An angular differential of 45 degrees is common, creating eight angular regions.

- **Advantages** — All regions have some center and some outlying pixels; nice rotational properties.
- **Disadvantages** — More difficult (expensive) to implement than rectangular regions. Some angles are more accurately sampled than others.

This is the most difficult spatial sampling scheme to implement. There are two good ideas on this subject, the first being to build triangles containing the portions of the image to be used, and then doing a standard point-in-polygon test to see if pixels lie within that polygon. The second idea uses a simple angle created by any pixel and the center of the image. It is this latter idea that will be described.

In OpenCV terms, the geometric center of an image, x, is simply:

$$(ci, cj) = (x->\text{height}/2,\ x->\text{width}/2)$$

Each pixel has *(i,j)* coordinates, too, and the angle from *(i,j)* to *(ci, cj)* to $(ci, cj + 100)$ can be thought of as the angle made by the pixel relative to the horizontal. Sampling angular regions is a matter of determining which angular region a pixel belongs to (i.e., between 0 and 45 degrees) and associating that with a histogram bin or other feature set.

A function called `angle_2pt` is provided that takes *ci*, *cj*, *i*, and *j* as parameters and returns an angle between 0 and 360 degrees. This turns out to be a spectacularly useful function in general; it can be used for finding angle-distance signatures, projections, convex hulls, and a score of geometric measures. In particular, it is used here for angular region sampling. Figure 10.4 shows the code for the function, as well as some graphical illustrations of the regions it finds for 45-degree increments.

```
float angle_2pt (int r1, int c1,
                 int r2, int c2)
{
    double atan(), fabs();
    double x, dr, dc, conv;


    conv = 180.0/3.1415926535;
    dr = (double)(r2-r1);
    dc = (double)(c2-c1);
/*  Compute the raw angle from Drow, Dcolumn   */
    if (dr==0 && dc == 0) x = 0.0;
     else if (dc == 0) x = 90.0;
     else
     {
        x = fabs(atan (dr/dc));
        x = x * conv;
     }
/* Adjust the angle according to the quadrant */
      if (dr <= 0) {//upper 2 quadrants
      if (dc < 0) x = 180.0 - x; // Left
     }
      else if (dr > 0) // Lower 2 quadrants
      {
        if (dc < 0) x = x + 180.0; // Left
          else x = 360.0-x;         // Right
      }
      return (float)x;
      }
```

Figure 10.4: (left) Code for the function *angle_2pt*, valuable for many situations involving angles. (right) Angular regions found using this function: regions 0, 1, 2, and 3.

10.4.4 Circular Regions

Circular regions consist of concentric circles or rings beginning at the geometric center of the image, as nearly as possible. We used five rings for our experiments. The radius of the last ring is equal to the maximum of the largest row and column index.

- **Advantages** — Rotationally invariant; using more rings is not really more costly. It is easy to weight the rings so that the image center is more important.
- **Disadvantages** — Expensive to implement. Pixels near the boundary are arbitrarily assigned to regions.

Sampling is done by determining distance to the image center. If there are to be five regions, then first compute the radius of the largest possible circle: r4 = MIN(height, width)/2. Divide this by Nregions-1 (= 4 in this case) giving the distance between two consecutive circles dr. Now subtract dr from R4 to give r3, and repeat twice more to find all radii. For example, if the image is 256x256 pixels and five regions are desired, then r4 = 256, dr = 64, r3 = 192, r2 = 128, and r1 = 64. Any pixel less than r1 in distance from the center pixels (128,128) is in region 1; pixels less than 128 in distance but greater than 64 are in region 2; and so on.

10.4.5 Hybrid Regions

Hybrid regions represent a logical combination of angular and circular regions, as defined above. This is really just a two-dimensional polar coordinate system. Both the concentric rings seen in circular regions and the radial segments of the angular regions are superimposed. Finding which region a pixel belongs to is easy if the circular and angular schemes have already been implemented: a pixel will belong to an angular region and a circular one, and these can be indexed in any convenient way.

For example, if 8 angular regions and 5 circular regions are used, there will be 40 hybrid regions. The groupings are: between 0 and 45 degrees and d > r4, between 0 and 45 degrees and (d > r3 and d < r4), and so on.

- **Advantages** — Combines the results of all the region-based methods, so should provide a better result over a wide class of images.
- **Disadvantages** — Expensive to implement because all the methods have to be calculated.

10.4.6 Test of Spatial Sampling

Given a variety of features, color and grey, and a variety of spatial sampling methods, there are a huge number of combinations of methods to try out. A composite system, one that uses multiple algorithms, that uses a selection of

color-based and simple shape feature-based techniques, could be put together quickly. Not only are multiple algorithms being used, but they are being applied to sub-regions of the images, determined in multiple ways.

Five algorithms were selected for the implementation: grey sigma, edge density, Boolean edge density, edge direction, and color histograms. For each algorithm, the five methods of defining regions on the image were used. The results were compared to each other using each similarity algorithm, and were compared as well as possible against some methods published in the technical literature.

A new experimental database was selected, just to try new data with different properties. The *Corel* data set has many thousands of images, and is distributed on CDs. Corel is a well-known software company, famous for products like Paint Shop Pro, WordPerfect and CorelDRAW Graphics Suite X5. They distribute image data with their products, and it has been used frequently for testing of image processing and retrieval algorithms in academic settings. For this project, 782 images were chosen from 8 classes. Figure 10.5 shows representative samples of each class. Seven of the classes had 100 images, while the final one had only 82. The *accuracy*, A, for each class is calculated as $A = 100c/qn$, where c is the number of correct (in-class) retrievals, n is the number of images in that class, and q represents the number of results that were returned [Seidl, 2001].

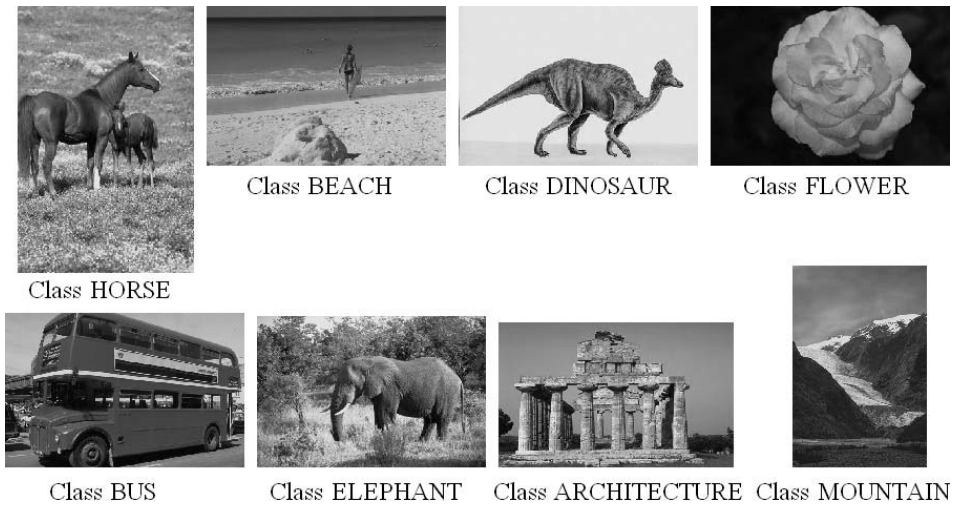


Figure 10.5: Sample images from classes used from the Corel data set.

Given this measure of success, the overall system was initially tested on the 782 images. Eight tables, one for each class, are needed to convey all the information resulting from the trials. Each image is queried against the database, and a table of success percentage with similarity algorithms occupying columns and region drawing methods as rows. A sample table, the one for the ''Beach'' class, is shown as Table 10.2; it is plain to see that there is

a significant variability in success across the features and sampling methods tested. It is also true that the method and region scheme that works best for one class does not necessarily work the best for some other. What is wanted, naturally, is a scheme that works best for all classes.

Table 10.2: Region/Feature Accuracies for Select Classes: "Beach" (%)

SPATIAL SAMPLING METHOD	GREY SIGMA	EDGE DIRECTION	EDGE DENSITY	BOOLEAN EDGE DENSITY	HUE HISTOGRAM	INTENSITY HISTOGRAM
OVERALL	12.8	18.3	9.6	11.6	23.7	18.5
RECTANGULAR	13.2	20.7	9.6	10.2	22.1	12.7
ANGULAR	12.7	15.1	9.6	9.6	23.7	14.9
CIRCULAR	20.0	12.5	13.1	15.2	23.4	20.1
HYBRID	22.1	21.4	5.8	14.3	25.5	12.4

Source: J.R.Parker, 2007

Figure 10.5 shows representative samples of each class. Unlike the ALOI data set, the backgrounds are an important aspect of most images.

The experiments involved searching for each image among all the other images, using every combination of similarity measure and region splitting method described above. This means that a similarity value has been calculated for all image pairs in the data collection. These can be sorted into a ranked list for each algorithm, in which the first image is the best (highest similarity value, most likely match). We can use these ranked lists as a means to vote for the best match. The method we have used in the past to do this is called the *Borda count* and is described in Section 8.5.5.

Other voting methods were attempted, such as the simple majority vote, weighted Borda count, and so on, but the simple Borda count appeared to provide the most robust solution. The overall results, using this set of algorithm combination methods, are shown in Table 10.3.

This means that, in a web search having ten results per page, the first page would have five to six correct (directly relevant) matches to the query, on the average. This is better than our informal poll suggests is acceptable, and better than the same poll is being achieved now on text-based queries.

In the methods presented, the work of Rao [1999] and that of Tico [2000] formed a central component, and so it seemed appropriate to compare the results above against their published work. We implemented both methods using the original papers as the definitive description of the method; this means that there is a chance that the resulting program generates results that are somewhat different from the one used by the original authors. The results were computed in the same way as for the previous experiment, and are

tabulated in Table 10.4. In all cases but one, the composite system here gives better success rates than any of the others.

Table 10.3: Search Results for Image Classes

IMAGE CLASS	RESULT
Horse	86.37%
Beach	28.78%
Dinosaur	98.97%
Flower	81.67%
Bus	58.78%
Elephant	39.30%
Architecture	40.43%
Mountain	26.90%
Overall	56.95%

Table 10.4: Comparison Between Retrieval Methods on Corel Data Set (%)

IMAGE CLASS	RAO %	TICO %	THIS BOOK
Beach	27.7	25.6	28.8
Horse	89.0	68.3	86.4
Dinosaur	42.0	72.6	99.0
Elephant	20.0	24.7	39.3
Flower	46.4	51.3	81.7
Architecture	27.0	24.2	30.4
Bus	36.0	33.7	58.8
Mountain	26.0	19.9	26.9
Overall	**39.5**	**40.4**	**57.0**

Source: J.R.Parker, 2007

10.5 Additional Considerations

Many aspects of content-based search methods have not been discussed yet. Only a brief summary of other useful techniques can be mentioned here, but it is interesting to note that most have been described in some other context in Chapters 2-6.

10.5.1 Texture

The texture measures based on grey-level co-occurrence have been used with some success to determine similarity between images. In particular, energy, entropy (Section 5.3.5), contrast (Section 5.3.3), and homogeneity (Section 5.3.4) have been used with some success [Ohanian, 1993]. An evaluation in the literature showed that homogeneity was the best of these features, at about 12% [Howarth, 2004]. Grey-level co-occurrence does typically take longer to calculate than most of the measures that have been discussed, though, and this may be a significant factor that argues against its use.

Texture features developed by Tamura [1978] are based on human perception rather than mathematical principles, and of the six he tested, three of them have some value for image queries based on texture: coarseness (the largest size at which a texture exists), contrast (dynamic range, related to statistical moments), and directionality (an impression of directionality over a region). These are less computationally intense than co-coherence matrices and give success rates in the 9–12% range. These measures are frequently referenced and used in real systems. For example, QBIC [Niblack, 1993] and MARS [Huang, 1996; Ortega, 1997] both use variations of Tamura features.

10.5.2 Objects, Contours, Boundaries

In the long run, it will be the ability to recognize objects within an image that will lead to highly successful content-based retrieval. At this time, the state of the art is relatively poor, but prospects are good. The first step in an object-based scheme would be segmentation, where the primary object could be distinguished from the background using colors, texture, edges, and similar methods. This is followed by either a statistical characterization of the object's shape or a structural analysis and comparison against known objects in a data collection.

Segmentation has been covered in general in Chapters 2, 4, and 5. The point of segmentation is simply to locate regions of the image that are homogeneous and contiguous; such regions often represent recognizable objects. These objects can be measured for shape. Features such as circularity, moments, Fourier descriptors, and a host of others can be measured and made into feature vectors, which, in turn, are compared for similarity against classified sets in the data collection. Mehtre [1997] has compared a large set of shape features for success in content-based searches, and notes that classical invariant moments appear to work very well. Vassilieva [2009] describes shape-based searches in more detail and gives some results.

10.5.3 Data Sets

Data sets on family computers tend to have some similarities. The family camera acquires images of a particular size and resolution, and will have relatively

stable color properties. However, people share photos, acquire some from the Internet and websites, and crop and edit photos. The result is that data sets in the wild have images with widely varying sizes, scales, and other properties.

In order for a content-based query system to provide the best results in these cases, it is probable that measurements should be made using fixed-size images. The users's images should not be altered, of course, but they should be scaled down to a constant fixed size (say, 256x256) before features are measured. Target images are likewise scaled before their features are extracted, making it more likely that matches will represent visually similar images.

Similarly, it may be better to modify the contrast of images to have intensity ranges that follow a known pattern. Images should have a similar contrast and should fill the intensity range before having features measured, at least in many cases. Images of a baseball game should be identified as the same regardless of the mean grey level over the image, and a copy of an image should not be thought of as different just because it is darker than the original.

It has been said before, and bears repeating, that objects in the center of an image are more important than those in outlying areas. Pixels near the center can be weighted more highly simply by multiplying them by a number greater than 1. Using a fixed-size image as a template, it is a simple matter to construct an image of weights where the center has, perhaps, a value of 2.0 and the outlying edges have a value of 0.2. Weighting pixels with these values is simple, and normalizing with respect to the weights is simple, too. The pixel values are multiplied by the weights before being used, and normalizing means dividing by the sum of the weights in any specified region.

Finally, the nature of the image set is important in some cases. Some data sets are highly homogeneous. Medical image data, for example, tend to be, as are more industrial data sets. It may be best to identify specific features in homogeneous data sets that permit fine degrees of variation to be detected, or that look in particular places in the image for some kinds of features. It is not cheating to take advantage of prior knowledge of the data, so long as the method is applied only to the data for which it was designed.

10.6 Website Files

A good listing of image ''databases'' can be found at `www-i6.informatik.rwth-aachen.de/dagmdb/index.php/Content-Based_Image_Retrieval`.

`angular.c`	Samples an image using the angular method and displays the sub-regions.
`check.c`	Confirms the correctness of a binary file of features against the text file.

`convert.c`	Converts a master text feature file into a binary file; binaries are faster and easier to read.
`display.c`	Displays the images in the data set directory.
`listGreyFiles.c`	Creates a master file of grey-level image names.
`makeCM.c`	Makes a file of color-central moments.
`makeMaster.c`	Creates a master file of images by scanning a directory.
`makeMaster2.c`	Creates a master text file of image features using the images in the master file.
`makeMasterGrey.c`	Creates a master feature file (binary only) for grey-level image data set.
`quadtree.c`	Builds and displays a color-based quad tree.
`search1.c`	Searches for an image, reads the image file name from the console, and searches the data set for others like it.
`search2.c`	Tests extracted features using already collected features from the data set to conduct searches and collect statistics.
`searchCM.c`	Tests a collection of color features on a set of color images.
`searchGrey.c`	Tests a collection of grey level-features by conducting 654 searches using pre-computed features and by collecting statistics.
`searchGreyMoments.c`	Use grey-level moments to search. Conducts more than 600 searches and collects statistics.
`searchGreyQuad.c`	Use grey-level histograms, including quad trees, to search. Conducts 654 searches and collects statistics.

10.7 References

Agrawal, R., K. Lin, H. S. Sawhney, and K. Shim. ''Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases.'' *Proceedings of the 21st International Conference on Very Large Data Bases*. Zurich, Switzerland (1995): 490–501.

Ankerst M., B. Braunmüller, H. P. Kriegel, T. Seidl. ''Improving Adaptable Similarity Query Processing by Using Approximations.'' *Proceedings of the*

24th International Conference on Very Large Data Bases. New York, NY (1998): 206–217.

Ankerst, M., G. Kastenmüller, H. P. Kriegel, T. Seidl. ''3D Shape Histograms for Similarity Search and Classification in Spatial Databases.'' *Advances in Spatial Databases, 6th International Symposium SSD* 99. Hong Kong, China (July 20–23, 1999): 207–226.

Ankerst M., H. P. Kriegel, T. Seidl. ''A Multistep Approach for Shape Similarity Search in Image Databases.'' *IEEE Transactions on Knowledge and Data Engineering* 10, no. 6 (1998): 996–1004.

Beckmann N., H. P. Kriegel, R. Schneider, B. Seeger. ''The R*-tree: An Efficient and Robust Access Method for Points and Rectangles.'' *Proceedings ACM SIGMOD International Conference on Management of Data*. Atlantic City, 1990: 322–331.

Berchtold S., C. Böhm, B. Braunmüller, D. Keim, H. P. Kriegel. ''Fast Parallel Similarity Search in Multimedia Databases.'' *Proceedings ACM SIGMOD International Conference on Management of Data*. Tucson, 1997: 1–12.

Berchtold S., C. Böhm, D. Keim, H. P. Kriegel. ''A Cost Model for Nearest Neighbor Search in High-Dimensional Data Spaces.'' *Proceedings of the 16th ACM Symposium on Principles of Database Systems*. Tucson, 1997: 78–86.

Berchtold S., C. Böhm, D. Keim, H. P. Kriegel. ''The X-tree: An Index Structure for High-Dimensional Data.'' *Proceedings of the 22nd International Conference on Very Large Data Bases*. Bombay (Mumbai), India: Morgan Kaufmann, 1996, 28–39.

Berchtold S., H. P. Kriegel. ''S3: Similarity Search in CAD Database Systems.'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Tucson, AZ: ACM Press, (1997): 564–567.

Brinkhoff, T., H. P. Kriegel, R. Schneider. ''Comparison of Approximations of Complex Objects Used for Approximation-based Query Processing in Spatial Database Systems.'' *Proceedings of the 9th International Conference on Data Engineering [ICDE]*. Vienna, Austria: IEEE Computer Society, (1993): 40–49.

del Bimbo, A., *Visual Information Retrieval*. San Francisco: Morgan Kaufmann, 1999.

de Borda, Jean-Charles. *Memoire sur les Elections au Scrutin*. Paris, France: Histoire de l'Academie Royale des Sciences, 1781.

Faloutsos, C., R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, W. Equitz. ''Efficient and Effective Querying by Image Content.'' *Journal of Intelligent Information Systems* 3 (1994): 231–262.

Faloutsos, C., K. I. Lin. ''FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Data.'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*. San Jose, CA: ACM Press, 1995, 163–174.

Faloutsos, C., M. Ranganathan, Y. Manolopoulos. ''Fast Subsequence Matching in Time-Series Databases.'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Minneapolis, MN: ACM Press, 1994, 419–429.

Gaede, V. and O. Günther. ''Multidimensional Access Methods.'' *ACM Computing Surveys* 30, no. 2 (1998): 170–231.

Geusebroek, J. M., G. J. Burghouts, and A. W. M. Smeulders. ''The Amsterdam Library of Object Images.'' *International Journal of Computer Vision* 61, no. 1, (2005): 103–112 (http://staff.science.uva.nl/~aloi/).

Guttman, A. ''R-trees: A Dynamic Index Structure for Spatial Searching.'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Boston: ACM Press, 1984, 47–57.

Hafner, J., H. S. Sawhney, W. Equity, M. Flickner, and W. Niblack. ''Efficient Color Histogram Indexing for Quadratic Form Distance Functions,'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17, no. 7 (1995): 729–736.

Holm, L., and C. Sander. ''Touring Protein Fold Space with Dali/FSSP.'' *Nucleic Acids Research* 26 (1998): 316–319.

Howarth, P. and S. Rüger. ''Evaluation of Texture Features for Content-Based Image Retrieval.'' *Third International Conference on Image and Video Retrieval. Lecture Notes in Computer Science* 3115 (2004): 326–334.

Korn, F., N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. ''Fast and Effective Retrieval of Medical Tumor Shapes.'' *IEEE Transactions on Knowledge and Data Engineering* 10, 6 (1998): 889–904.

Kriegel, H. P., T. Seidl. ''Approximation-Based Similarity Search for 3-D Surface Segments.'' *GeoInformatica* 2, no. 2 (1998): 113–147.

Kruskal, J. B., and M. Wish. *Multidimensional Scaling*. Beverly Hills, CA: SAGE Publications, 1978.

Mehtre, B. M., M. S. Kankanhallib, and W. F. Lee. ''Shape Measures for Content Based Image Retrieval: A Comparison.'' *Information Processing and Management* 33, no. 3 (1997): 319–337.

Muller, H., S. Marchand-Mailler, and T. Pun. ''The Truth About Corel Evaluation in Image Retrieval.'' *Proceedings of the International Conference on Image and Video Retrieval*. Berlin: Springer-Verlag, 2002, 38–49.

Niblack, W. ''The QBIC Project: Querying Images by Content Using Color, Texture, and Shape.'' *Proceedings of SPIE* 1908, no. 1 (1993): 173–187.

Ohanian, P.. R. Dubes, ''Performance Evaluation for Four Classes of Textural Features.'' *Pattern Recognition* 25 (1992): 819–833.

Parker, J. R. ''Voting Methods for Multiple Autonomous Agents.'' *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems*. ANZIIS-95. Perth, Australia (1995): 128–133.

Parker, J. R., and B. Behm. ''Composite Algorithms in Image Content Searches.'' *International Journal of Software Engineering and Knowledge Engineering, World Scientific* 17, no. 4 (2007): 451–463.

Rao, A., R. K. Srihari, and Z. Zhang. ''Spatial Color Histograms for Content-based Image Retrieval.'' *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*. Chicago, 1999, 183–186.

Sawhney H. and J. Hafner. ''Efficient Color Histogram Indexing.'' *Proceedings International Conference on Image Processing*. Austin, TX: IEEE Computer Society 1994, 66–70.

Seidl, T. ''Adaptable Similarity Search in 3-D Spatial Database Systems.'' (Ph.D. thesis, University of Munich, 1997). Munich, Germany: Herbert Utz Publishers, 1998.

Seidl, T. and H. P. Kriegel. ''Efficient User-Adaptable Similarity Search in Large Multimedia Databases.'' *Proceedings of the 23rd International Conferences on Very Large Data Bases*. Athens, Greece: Morgan Kaufmann 1997, 506–515.

Seidl, T. and H. P. Kriegel. ''Optimal Multi-Step k-Nearest Neighbor Search.'' *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Seattle, WA: ACM Press 1998, 154–165.

Seidl, T. and H.-P. Kriegel. ''Adaptable Similarity Search In Large Image Databases,'' in *State-of-the-Art in Content-Based Image and Video Retrieval*, ed. R. Veltkamp, H. Burkhardt and H.-P. Kriegel. Boston: Kluwer Academic Publishers, 2001, 297–317.

Sellis, T., N. Roussopoulos, and C. Faloutsos. ''The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.'' *Proceedings of the13th International Conference on Very Large Data Bases*. Brighton, England: Morgan Kaufmann 1987, 507–518.

Stricker, M. and M. Orengo. ''Similarity of Color Images.'' *Proceedings of the SPIE Conference on Storage and Retrieval for Image and Video Databases III*. 2420 (1995): 381–392.

Swain, M. and D. Ballard. ''Color Indexing,'' *International Journal of Computer Vision*, 7 (1991): 11–32.

Tamura, H., S. Mori, and T. Yamawaki. ''Textural Features Corresponding to Visual Perception.'' *IEEE Transactions on Systems, Man, and Cybernetics* 8 (1978): 460–472.

Tico, M., T. Haverinen, and P. Kuosmanen. ''A Method of Color Histogram Creation for Image Retrieval.'' *Proceedings of the Nordic Signal Processing Workshop NORSIG2000*. Kolmården, Sweden, 2000, 157–160.

Vassilieva, N. S. ''Content-based Image Retrieval Methods.'' *Programming and Computer Software* 35, no. 3 (2009): 158–180.

White D. A., and R. Jain. ''Similarity Indexing with the SS-tree.'' *Proceedings of the 12th International Conference on Data Engineering*. New Orleans, IEEE Computer Society, 1996, 516–523.

Yen, C.-Y. and K. J. Cios. ''Image recognition system based on novel measures of image similarity and cluster validity.'' *Neurocomputing* 72, no. 1–3 (2008): 401–412.

Zhang, D. and G. Lu. ''Content-Based Shape Retrieval Using Different Shape Descriptors: A Comparative Study.'' *Proceedings of the IEEE International Conference on Multimedia and Expo*, Tokyo: IEEE Computer Society 2001, 317–320.

10.7.1 Systems

Faloutsos, C., W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. ''Efficient and Effective Querying by Image Content.'' *Journal of Intelligent Information Systems*, 3. Kulwer Academic Publishers, 1994, 231–262.

Huang, T. S., S. Mehrotra, and K. Ramchandran. ''Multimedia Analysis and Retrieval System (MARS) Project.'' *Proceedings of the 33rd Annual Clinic on Library Application of Data Processing — Digital Image Access and Retrieval*, Urbana-Champaign, IL: PUBLISHER 1996, PAGES.

Ma, W. Y. and B. S. Manjunath. ''NetRa, A Toolbox for Navigating Large Image Databases.'' *Proceedings of the International Conference on Image Processing*, Santa Barbara, CA: IEEE Signal Processing Society Vol. 1, 1997, 568–571.

Ortega, M., Y. Rui, K. Chakrabarti, S. Mehrotra, and T. S. Huang. ''Supporting Similarity Queries in MARS.'' *Proceedings of the Fifth ACM International Conference on Multimedia*, Seattle: ACM Press, 1997, PAGES.

Smith, J. and S. F. Chang. ''VisualSEEK: A Fully Automated Content-based Image Query System.'' *Proceedings of the Fourth ACM International Conference on Multimedia*, Boston: ACM Press, 1996, 87–98.

CHAPTER

11

High-Performance Computing for Vision and Image Processing

Gordon E. Moore, the founder of Intel, once observed that the number of transistors that can be placed on an integrated circuit doubled every 20 months. The implication is that the amount of computing power that could be purchased for a fixed price (*bang for the buck*) also doubled at the same rate. This was accomplished for many years by making CPUs smaller and faster; however, since perhaps 2007, speed increases have been as much the result of using parallel computing (multiple processors) as it has been as result of faster single CPUs. In 2010, it is rare to find a PC running a single processor faster than 3.5 GHz; it is relatively common to find PCs with 3 or 4 processors, each faster than 3 GHz. These systems are less expensive than single processor systems of 2005 and can do a lot more work.

A multiple CPU system can do more computation per time unit than a single CPU system in situations where the problem itself can be split into parts, and the data can too. A two-dimensional Fourier transform is just such a problem. Each row is transformed independently, which can be done by distinct processors, followed by the columns. Sorting, on the other hand, is not such a problem, and it is difficult to gain a computational advantage by using multiple processors to sort a collection of numbers. Most image-processing tasks can be applied to parts of an image and then merged later into an image again, meaning that parallel computing can be used for a performance improvement. It is also true that with the increase in typical image sizes over the past decade and the algorithmic complexity of many image processing methods, there is a good reason to consider it.

Unfortunately, an overhead is associated with breaking a problem into parts that affects the speed improvement and, in fact, limits the gains that can be achieved. There is also a programming cost, and frequently a cost connected

with the equipment and software needed. Let's see if there are some relatively simple and inexpensive ways to achieve some parallelism.

11.1 Paradigms for Multiple-Processor Computation

There are multiple paradigms for performing computations in parallel, and a few ways to implement each. A *vector computer* performs multiple computations on multiple data elements using special hardware. For example, such a machine can multiply each element of a vector by another number in the time needed to multiply two numbers, speeding up the computation time by a factor of N (the size of the vector). A problem is that vector computers are special-purpose machines and expensive. Another problem is that the N-times speedup depends on a setup (requiring time), so a gain is possible only if a lot of calculating is going on. The pump needs to be primed, as it were.

Distributed computing involves sending parts of a problem to various processors (or computers, on a network). Each processor does its share of the computation and sends the results back to some central repository. This style of computing needs little or no special hardware, but some time is needed to send the data to each processor and then for the processor to return the answer. Input/output takes a lot more real time than does computation or memory access. This means that a distributed computation is faster than a simple one only if the time needed to send the data to another processor is shorter than the time needed to do a calculation and receive the results

11.1.1 Shared Memory

A shared-memory multiprocessor possesses a collection of CPUs that are near enough to each other physically that they can use the same memory. The nearness increases speed but means that the hardware is specialized and difficult to expand, in general. Multi-core CPUs on PCs are essentially shared memory multiprocessors, as are many vector machines.

The nature of these parallel computers creates some programming difficulties. Although many processors can read the same memory location at the same time, writing to memory requires some sort of protection. Without it, the results of calculations can be unpredictable. For example, consider a program that executes on two computers simultaneously. The pixel value at (i,j) is RGB = (128, 84, 200).

```
Processor 1             Processor 2
get pixel p at i,j                          p=(128, 84, 200)
g = green value at p        . . .           g = 84
r = red value at p      get pixel p at i,j  r=128; p=(128, 84, 200)
b = blue value at p     g = green value at p  b=200,   g=84
```

```
k = (r+b+g)/3          r = red value at p      k=137;   r=128
p = (k, 0, 0)          b = blue value at p     p=(137,0,0) b = 0
. . .                  k = (r+b+g)/3           k = 137/3 = 46
                       p = (k, 0, 0)           p = (46,0,0)
      . . .
```

What happens when two identical programs run using the same data but slightly out of synchronization is two different results. On one processor, the code above averages the color values of pixel p, giving a grey value of (137,0,0). When two processes use the same memory locations simultaneously, the result is quite different because the processes are modifying the memory at the same time. In the preceding situation, the result is a level of (46,0,0), but the result changes depending on how the interaction takes place.

When using shared memory, only one process can be allowed to access shared memory at a time. This is done by stopping one process temporarily, and then allowing it to proceed when it is safe. Code that modifies a set of shared memory locations is called a *critical section* and can be protected using locking or semaphores. More details can be found in the references. What is important is that software for parallel programming offers such protection, and that programmers are aware of the results of inappropriate access to shared data.

11.1.2 Message Passing

In this paradigm for distributed processing, a program is written as a set of independent processes that communicate by sending packages of data between them. Data packages, or messages, can have different types, as can send integers, or floats as data to each other. There are many geometries for sending messages, each useful for a different kind of processing. A process can send a message with different data to a set of other processes, such as sending a row of a matrix to a Fourier transform program. Similarly, a process can send a message to a large group of processes, such as sending a threshold to processes representing parts of images. Processes can even be organized into ''loops,'' where A sends data to B, who sends to C, who sends to A.

Processes can execute on one computer, on a multi-core CPU, or on many computers over a network. There is a great degree of flexibility in a message-passing system. One program can take advantage of a large number of processors or run to completion on just one. And there are many tools available for writing distributed programs using message passing. Section 11.3 discusses one such example — the Message-Passing Interface (MPI) system.

11.2 Execution Timing

Because the goal of parallel computation is to produce the results in a shorter time, it is important to know how to measure the time taken for a program to execute. Of course, the nature of the results is the most important thing,

and if a faster program produces answers that are wrong or inaccurate, the faster program is useless, so the slow, correct one should be used. However, all results being equal, the faster program is to be preferred. How can the execution time be measured and the speedup quantified?

The commonly used scheme for timing C programs is to insert timing code into the source. The clock on a computer is like any other, in that it advances second by second. Sampling the clock before the program starts and then when it is done yields two times, and the difference is the real time needed to perform the computation. This is the important thing — the actual time needed to calculate the result. The CPU time, or the number of cycles used by the processor multiplied by the time for one cycle, is an abstract thing and could require vastly more real time to complete.

Computers these days are quick, and measuring computation time requires a clock with a good resolution. A not especially high-powered PC can evaluate a sine function a million times in 0.05 seconds. (The same calculation would have taken 0.875 seconds when the first edition of this book was written in 1997.) This means that a clock used to measure execution time needs to be accurate to under a millisecond, and preferably to under a microsecond.

11.2.1 Using *clock()*

An obvious way to measure time is to use the function `clock()`. It is relatively generic, being found on Unix, Linux, (in the GNU libraries at least), and Windows systems (needing the include file `windows.h`). It is easy to use, too.

`clock()` takes no arguments and returns a value of type `clock_t`, which is an *unsigned int* or *unsigned long*, depending on the system. In either case, timing code is a simple matter. `clock()` returns a time since reboot (or similar), so record the clock time before and after the calculation, and then subtract them. The function returns the time in an arbitrary unit, clock ticks, so to convert to seconds, the measured time is multiplied by a system-dependent constant, `CLOCKS_PER_SEC`, which is the number of ticks in a second. Now the execution time is known. It is often more useful to have this time in milliseconds, so divide by 1000 to get this.

Here's an example program that times a simple calculation. In order to get a significant value for execution time, a loop that computes a sine function a large number of times is instrumented, as we say, with timing based on `clock()`. The basic inner loop computes sine 1024 times. An outer loop runs the inner loop `niter` times, and `niter` doubles for each trial. The result is a set of times, each representing twice the computational effort of the one previous. The program is:

```
/* example of the use of clock() to time code */
#include "mpi.h"
```

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <time.h>

#define BUFSIZE 1024
#define ITERS   1000

int main(int argc, char *argv[])
{
    int i, j, niter=0;
       double xtime, dat[BUFSIZE], y, z;
       clock_t cstart, cstop;

       for (niter=ITERS; niter < ITERS*1000; niter *= 2)
       {
               cstart = clock();
               for (j=0; j<niter; j++)
               {
                     for (i=0; i<BUFSIZE; i++)
                     {
                           dat[i] = sin (3.1415926535 * i/1024.0);
                     }
               }

               xtime = (double)(clock() - cstart)*CLK_TCK/1000.0;
               printf ("Iterations %d  Clock time is %lf = %lf\n",
                           niter, xtime, xtime/niter);
       }
       printf ("Returning.\n");
    return 0;
}
```

Each time value measured should be twice the time value of the previous measurement, but this is not quite the case. A sample output is:

```
Iterations 1000  Clock time is 46.000000 = 0.046000
Iterations 2000  Clock time is 94.000000 = 0.047000
Iterations 4000  Clock time is 188.000000 = 0.047000
Iterations 8000  Clock time is 359.000000 = 0.044875
Iterations 16000  Clock time is 750.000000 = 0.046875
Iterations 32000  Clock time is 1531.000000 = 0.047844
Iterations 64000  Clock time is 2938.000000 = 0.045906
Iterations 128000  Clock time is 5875.000000 = 0.045898
Iterations 256000  Clock time is 11781.000000 = 0.046020
Iterations 512000  Clock time is 23531.000000 = 0.045959
Returning.
```

Note that the measurements for small numbers of iterations seem to be less accurate than those for larger numbers, as indicated by the difference between consecutive timings on a per-iteration basis. In fact, time is the issue; the longer the program runs, the more accurate the timing is when using the `clock()`

function. The last number in each output line is the number of milliseconds per inner loop (1000 sine calculations). As more and more such loops are timed, the accuracy seems to improve. The last two trials take a long time, but they appear to produce times that are good to two significant figures, or perhaps to 0.00005 milliseconds.

The lesson is that to get good timings, the program has to run for a long time. A typical way to do this is to perform the same calculations many times, as was done in the example, but on real algorithms and real data. So, when timing an edge-detection algorithm, simply repeat it a hundred times and divide the results by 100 to get an accurate answer.

By the way, don't expect the timings to reproduce exactly on two different trials. A PC is doing other things in addition to executing the program being timed. The operating system interferes with the timing process, sometimes swapping out the program in favor of another one, or Windows can decide to do a disk-temperature calibration or poll its devices. This adds to the time that is measured during the program's execution. More accurate timings can be obtained by repeating the measurements many times and using the average.

11.2.2 Using QueryPerformanceCounter

Microsoft Windows operating systems (7 and XP, at least) provide a system-specific, high-resolution timer named `QueryPerformanceCounter`. It is supposed to be accurate to within nanoseconds and uses a 64-bit representation for time. Although it is somewhat complex to use, it is a lot like the simple `clock` function and is used in almost exactly the same way.

`QueryPerformanceCounter` takes a single argument, which is a pointer to a `LARGE_INTEGER`. A `LARGE_INTEGER` is actually a structure that represents a 64-bit number, needed for high-resolution times, and which can be ported across systems (but is probably used only on Windows). As before, the timing function `QueryPerformanceCounter` is called before and after the code to be timed, but passing the variable in which to store the time as a parameter. The number part of the `LARGE_INTEGER` structure is called `QuadPart`; if `stop` and `start` are `LARGE_INTEGER` variables used to store the time at the start and end of the code execution, respectively, then the duration is `stop.QuadPart - start.QuadPart`, and can be cast as `double` without losing much. The code timer seen previously can be written to use `QueryPerformanceCounter` as:

```
/* Timing C code using QueryPerformanceCounter */
#include "mpi.h"
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <time.h>
```

```
#define BUFSIZE 1024
#define ITERS   1000

int main(int argc, char *argv[])
{
    int i, j, niter=1000;
      LARGE_INTEGER start, stop, quantum;
      double xtime, dat[BUFSIZE];


      for (niter=ITERS; niter<ITERS*1000; niter *= 2)
      {
          QueryPerformanceCounter(&start);
          for (j=0; j<niter; j++)
          {
                 for (i=0; i<BUFSIZE; i++)
                 {
                        dat[i] = sin (3.1415926535 * i/1024.0);
                 }
          }
          QueryPerformanceCounter(&stop);
          QueryPerformanceFrequency( &quantum ) ;
          xtime = (double)(stop.QuadPart -
              start.QuadPart)/(double)(quantum.QuadPart);
          printf ("Done. Time = %lf    Iterations %d  per iteration %lf\n",
          xtime, niter, xtime/niter * 1000);
      }

      printf ("Returning.\n");
    return 0;
}
```

There are two important things to note. First, the `include` file `windows.h` is needed. Second, the resolution of the timer is returned by the function `QueryPerformanceFrequency`, and the time difference divided by this value gives the time in seconds. The output of the preceding program is:

```
Done. Time = 0.045933    Iterations 1000  per iteration 0.045933
Done. Time = 0.091912    Iterations 2000  per iteration 0.045956
Done. Time = 0.184248    Iterations 4000  per iteration 0.046062
Done. Time = 0.367418    Iterations 8000  per iteration 0.045927
Done. Time = 0.736176    Iterations 16000  per iteration 0.046011
Done. Time = 1.527363    Iterations 32000  per iteration 0.047730
Done. Time = 2.996671    Iterations 64000  per iteration 0.046823
Done. Time = 5.883134    Iterations 128000  per iteration 0.045962
Done. Time = 11.779256   Iterations 256000  per iteration 0.046013
Done. Time = 23.556424   Iterations 512000  per iteration 0.046009
Returning.
```

Note that this high-resolution timer produces essentially the same results on a PC as does the simple `clock` function. For 16,000 iterations and above, the

two times differ in the fourth decimal place; since the times are milliseconds, the differences are microseconds. In the worst case, they differ in the third place, and that's pretty good.

In either case, there is now code available to find accurate timing of image analysis and classification code. The next section discusses how to use this for parallel systems. In particular, shared memory and message-passing systems will be examined to see how difficult they are to use, whether accuracy is affected, and how much improvement in execution time can be achieved.

11.3 The Message-Passing Interface System

The Message-Passing Interface (MPI) system is a well-tested and freely available system for implementing message passing on PCs. The basic principle used by MPI is the sending of messages, or packages of information, to computers on a network. Each message can contain data and commands for the computer to process; it does this and sends a message back with the results. A central computer usually deals out the data to the satellite computers and sorts the returning data packages into their correct place, meaning that there is a potential bottleneck. MPI is a standard message-passing protocol that is well implemented and easy to install and use.

11.3.1 Installing MPI

MPI requires that special calls be added to the source code that implements the image-processing operations, so it needs to be installed so that it works with a convenient compiler. The programs in this book all work with Microsoft Visual C++ 2008 Express Edition, and fortunately MPI works with this system, and many others.

1. Download the MPI executable from `www.mcs.anl.gov/research/projects/mpich2/`.
2. Run the install script `mpich2-1.2.1p1-win-ia32.msi`.

   MPI will be installed, usually into `C:\Program Files\MPICH2`. Documentation can be found online at `www.mcs.anl.gov/research/projects/mpich2/documentation/index.php?s=docs`.
3. Set up the compiler. Running MVC++ 8:
   - In Project ➪ Properties ➪ Linker ➪ Input, add `mpi.lib` to Additional Dependencies.
   - In Project ➪ Properties ➪ C/C++ ➪ General, add `C:\Program Files\MPICH2\include` to Additional Include Directories. This path is normal

but may change, depending on how MPI is installed. In all cases, use `X\include`, where `X` is the full path name of the directory where MPI is installed.

- In Project ➪ Properties ➪ Linker ➪ General, add `C:\Program Files\MPICH2\lib` to Additional Library Directories. This path is normal but may change, depending on how MPI is installed. In all cases, use `X\library`, where `X` is the full path name of the directory where MPI is installed.

Now programs can be compiled, but the MPI system needs to be set up.

4. Run `wmpiregister` and enter the username and password that you use for your PC.
5. Run `wmpiconfig` and select configuration options.

MPI is now set up on one computer.

11.3.2 Using MPI

A simple example of MPI will show the initialization phase — the way parallelism is seen by the program and the way data is sent to and from processes. Programs compiled for MPI must include `mpi.h`, which contains the definition of relevant types and constants. The main program must initialize the MPI system before any other operations are performed; the call to `MPI_Init(0, 0)` does this.

When the program starts, a number of processors to be used, let's say *N* of them, is specified either on the command line or in a box in a `WMPIEXEC` window (more on this later). The call to `MPI_Init` results in *N*-1 processes being created; the creating process is the first and is numbered 0. The other processes are numbered in ascending order from this to *N*-1. The number assigned to the process, called its *rank*, can be gotten from MPI by using the call:

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

The procedure will return a value in the parameter `my_rank` that will be a value between 0 and *N*-1 inclusive. A value of 0 means that this is the root or main process, and the other value is a unique integer identifying the process. The total number of processes created can be found by calling:

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

The value *N*-1 is returned in `size`.

A common way to implement a parallel program using MPI is to have the main process (i.e., 0, or *master*) read or acquire the data and distribute it to the other processes (called *slaves*), which compute the results. The slave processes

then send the data back to the master, which collates it and save or prints it. This is simply described by the following code:

```
 if (my_rank ==0)                // Master
        master (size);
else                             // Slave
        slave (my_rank, size);
```

This divides the program quite logically into the two parts, and all the MPI programs in this book can be written in this way.

Finally, the MPI system needs to be terminated, thus freeing system resources and destroying the processes. This is done using:

```
MPI_Finalize();
```

What has been shown is the skeleton of almost any MPI-based parallel system for processing image or, in fact, most other kinds of data.

11.3.3 Inter-Process Communication

Inter-process communication is at the heart of any message-passing system and is sufficiently complex that entire volumes can be written about it. The code on the website will use a basic scheme, allowing others to be examined as wanted or needed. The most important part of the system is the ability to send and receive *messages* — a message being simply a collection of data with a header. This seems like a simple idea, but sending and receiving of messages improperly causes much of the trouble that programmers have with MPI.

A message can be sent using:

```
MPI_Send (data, count, data_type, destination,
                              tag, communicator)
```

The variable `data` is a pointer to the information to be sent, and it will be of type `data_type`. The variable `count` is the number of elements of that type being sent. The `destination` is an integer representing the number of the process that is to receive the data, and `tag` is an integer that can represent whatever the user wishes. For example, it generally indicates a message type: `1` could be for image data, `2` could be for parameters to the algorithm, and so on. The type of message sent should generally synchronize with what is being expected by the receiver. The `communicator` is a handle indicating details of the communication. Most usually, and what will be used here, is the standard `MPI_COMM_WORLD` handle.

This function does what is called a *blocking* send. This means that the process that is calling `send` will wait (or goes *blocked*, in operating systems terminology) until the message has been received before continuing to execute. This is one

cause of problems for programmers: if the message is never received, then the process blocks forever, or *hangs* as they say. If enough processes hang, or if the main program does, then the program will never terminate. Avoiding these deadlock situations is critical to correct coding of parallel programs.

An example of a send operation would be the master process sending a part of an image to a slave process — for example, process 2. The image data is copied to an array called `data`, which contains grey-level pixels. This means the data type is `unsigned char`. If three rows of the image are to be sent, and each row is `NC` pixels wide, then the call would be:

```
MPI_Send (data, NC*3, MPI_UNSIGNED_CHAR, 2, 1, MPI_COMM_WORLD)
```

Each data type defined by C or C++ has a corresponding constant that indicates it to `MPI_Send`: `MPI_UNSIGNED_CHAR` is what is needed here, but the other choices are obvious, such as `MPI_INT`, `MPI_FLOAT`, and so on. The tag value is not important unless the programmer assigns some importance to it; it is ignored by the system in most (but not all) regards. The tag value of `1` here means ''data.''

On the other end of a send operation is a receive, where a process accepts information from another. The form of the call is:

```
MPI_Recv(data, size, MPI_UNSIGNED_CHAR, 0, MPI_ANY_TAG,
      MPI_COMM_WORLD, &stat)
```

The first three parameters are pretty clear. The fourth, `0`, means that this call expects a message from process 0, or the master process. It is possible to receive from any process, or even to specify that the send is not known. The constant `MPI_ANY_TAG` means that the receive can accept any tag sent. This is the one place where the tag matters. If a specific tag is given, then the tag on the message that was sent should match.

The final parameter is a returned status, and from it the size of the message that was sent, the actual tag sent, error codes, and so on can be retrieved. The status code is a structure holding the following fields:

- `MPI_SOURCE` — The process number of the sender
- `MPI_TAG` — The tag that was sent
- `MPI_ERROR` — Any error codes associate with the message

A receive operation that corresponds with the preceding send would be:

```
MPI_Recv (data, NC*3, MPI_UNSIGNED_CHAR, 0, 1,
                                  MPI_COMM_WORLD, &status)
```

If the size of the data being received is not known, a parameter that specifies the largest size that is to be expected will work. This allows the system to allocate enough buffer space.

At this point, a simple example can be compiled, executed, and timed.

11.3.4 Running MPI Programs

The program `mpiTest1.c` on the website is a simple test of the MPI system. It simulates the execution of some arbitrary workload to be executed in parallel, and can demonstrate the way that send and receive calls are used, how programs are executed, and that parallel code is faster (in real time) than regular code.

After the compiler has been set up properly, this program should compile to `mpiTest1.exe`; a copy of this executable can be found on the website, too. It can be executed in two ways: First, and to be suggested under Windows, is to use `wmpiexec`, which is a part of the Windows installation. Figure 11.1 shows the `wmpiexec` window that was used to execute `mpiTest1`.

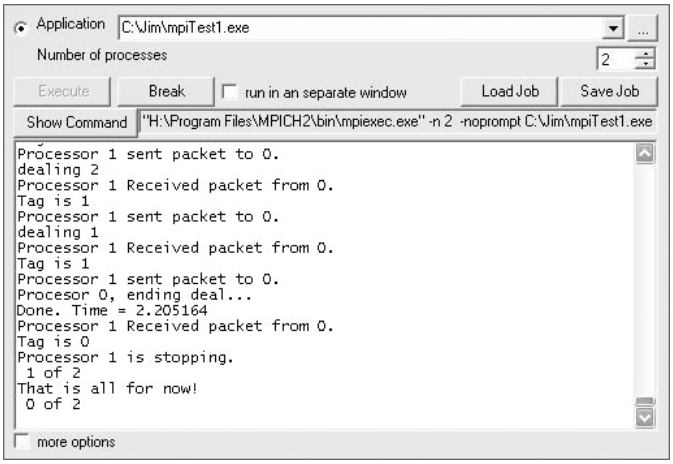


Figure 11.1: Using `wmpiexec` to run an MPI program.

The name of the executable file must be typed into the Application field (or use the browse key). In this example, the number on the upper-right part of the window above the Save Job button is 2 and specifies the number of processes to create. When the Execute button is pressed, the program starts to run with two processes (in this case), and the output appears in the large white area below. Note the output line that says:

```
Done. Time = 2.205164
```

The other way to run the program is from the command line. Execute the Windows Command Prompt program, change to the directory where the file `mpiTest1.exe` is located, and then type the following command:

```
"C:\Program Files\MPICH2\bin\mpiexec.exe" -n 2 -noprompt mpiTest1.exe
```

This command assumes that the MPI system has been installed in the normal place (`C:\Program Files\MPICH2`). It also specifies that two processes are to be

created (the `-n 2` parameters). Adding the string `> out.txt` to the end of the preceding command will result in the output from the program being placed into a file called `out.txt` instead of being displayed on the screen. Sometimes the output is needed later.

Although this program does not do any useful work, it does require a lot of time to execute because it performs a calculation in place of real work. It's a dummy load and allows the code to be timed as described in Section 11.2.2

The computer on which this program is running is a quad-core Intel Q8200, each processor being 2.33 Ghz. This means that four simultaneous processes can be executed: the master (0) and three slaves. After that, any further processes will run on one of the four processors, in addition to some other processes, and not much, if any, speed improvement can be expected. This was tested by running the same program repeatedly, specifying 2, 3, 4, 5, and 6 processes and examining the elapsed time, which was measured using `QueryPerformanceCounter` in the master process. The times are:

PROCESSORS	TIME
2	2.02077
3	1.06104
4	0.775137
5	0.622573
6	0.741093

The fact that the times get smaller until 5 processors are used is actual proof that the system is taking advantage of parallelism to compute the result. That the speed improves a little after the addition of processor 5 implies that one of the processors has some extra spare time in addition to what is used for the basic computation.

11.3.5 Real Image Computations

The program `mpiTest1` is a dummy, as it spends CPU time without doing any useful work. Let's build a parallel program that does something. Using the same framework, it should be possible to implement a time-consuming operation. A *median filter* comes to mind.

A median filter is intended to smooth noise in an image by replacing each pixel in the image by the medial value of the pixels in a neighborhood around that pixel. A 13-point median filter, for instance, takes the 13 nearest pixels to the target, computes the median, and uses that value to replace the target value. It is computationally expensive because finding a median means sorting

the 13 pixels and selecting the seventh in the sorted list as the median. Thus, each such filter involves Rows x Columns sorts of 13 numbers.

The master process sends two messages to each slave process. The first message contains the size of the image that will be sent to it, as number of rows and columns. The size will vary according to the number of processes. If there is one slave, the whole image will be sent; two slaves means that half will be sent to each; and so on.

The second message sent will contain the pixel data. A pointer to the first row is passed as the buffer pointer, and the specified number of rows is the count. The code is:

```
nr = im->info->nr; nc = im->info->nc;
     if (size-1 > 1)
          n = (nr+(size-1)*4)/(size-1);  // How many rows per processor
     else n = nr;
     j = 0;
     for (partner = 1; partner < size; partner++)
     {
          rstart = j; rend = j+n+2; if (rend>=nr) rend = nr-1;
          b1[0] = (rend-rstart+1);
          MPI_Send (b1, 2, MPI_INT, partner, 1, MPI_COMM_WORLD);
          MPI_Send (im->data[rstart],
               (rend-rstart+1)*nc, MPI_UNSIGNED_CHAR,
               partner, 1, MPI_COMM_WORLD);
          j = rend-3;
     }
```

When the slaves (I call them *partners* in the code; it's friendlier but non-standard terminology) receive the message, they calculate five passes of a median filter. They have to receive both messages from the master, in order:

```
MPI_Recv (p, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
     im1 = newimage (p[0], p[1]);
// p[0] is Nrows, p[1] is Ncols.
     im2 = newimage (p[0], p[1]);
     MPI_Recv(im1->data[0], (4+p[0])*p[1], MPI_UNSIGNED_CHAR, 0, MPI_ANY_TAG,
          MPI_COMM_WORLD, &stat);
```

The calculation is made and then the data is returned:

```
MPI_Send(im2->data[0], p[0]*p[1], MPI_UNSIGNED_CHAR, 0,1,MPI_COMM_WORLD);
```

The master receives the data from the partners, which move it directly into the image array:

```
j = 0;
for (partner = 1; partner < size; partner++)
{
     rstart = j; rend = j+n+2; if (rend>=nr) rend = nr-1;
     MPI_Recv (im->data[rstart], (rend-rstart+1)*nc,
         MPI_UNSIGNED_CHAR, partner, 1, MPI_COMM_WORLD, &stat);
```

```
    j = rend-3;
    sprintf  (name,"H:\\AIPCV\\stage%1d.pgm", partner);
    Output_PBM (im, name);
}
```

The master saves a copy of each partner's work in an image file. The results for a four-processor run are shown in Figure 11.2. The execution timings show that adding a fourth processor did not help, but that there is a speedup of 25% for using three CPUs. The timings were done four times and the average was taken. This is standard practice; the average of N measurements is a more accurate estimate of the actual value than is any one of them. The timings are available in Table 11.1.

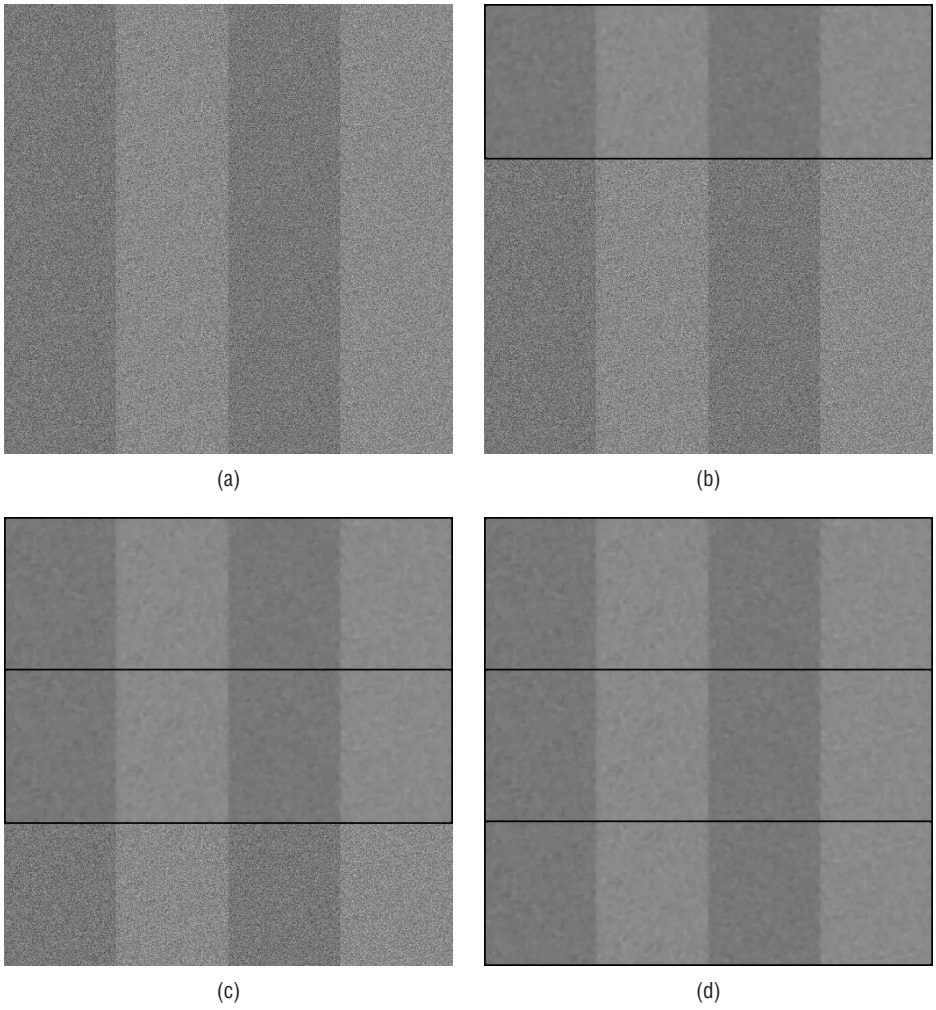


Figure 11.2: The result of a parallel computation of a 13-point median filter. (a) Original. (b) After processor 2 is done. (c) After processor 3. (d) After processor 4.

Table 11.1: Execution Time for Four-Pass, 13-Point Median Filter (Seconds)

CPUS	TIME 1	TIME 2	TIME 3	TIME 4	AVERAGE
2	3.172	3.183	3.163	3.164	3.171
3	2.365	2.412	2.374	2.364	2.378
4	2.474	2.458	2.460	2.460	2.463

A problem was discovered while building the median filter program. OpenCV seems to be incompatible with MPI (or vice versa), so programs using both will fail to run properly. This may be fixed in a future release, or a workaround may be found.

11.3.6 Using a Computer Network—Cluster Computing

So far, the performance improvement has been a result of using more than one processor on a specific computer. These processors are very near each other, and high performance can be expected due to the fact that they share a high-speed bus; so, data transfers can take place at memory access speeds. MPI was designed originally to work across Ethernet networks, where a collection of PCs (a *cluster*) would work on the same problem using messages passed back and forth over the network. It's easy to do this using Linux or Unix workstations, but on Windows there are a lot of layers of software on the way. PCs are consumer items, and Windows is meant for consumers; Unix and Linux originally were, and are still mainly, built for use by programmers and other adepts, and have fewer automatic safeguards.

Using MPI in a cluster setting does not require any changes to the source program. The configuration of the computers involved, on the other hand, has to be carefully set up. The high-level changes to the PCs being used as slave processors are:

1. Turn off the firewall. Other security software might have to be shut down, too, depending on the system.
2. The executable program must be copied to and reside at the same location on all computers. It may be a good idea to create a directory specifically for this purpose, perhaps `C:\mpiprocs`. The complete path must be identical on all machines, and the directory `C:\mpiprocs` must be shared; right-click the directory and select the Sharing tab (see Figure 11.3).
3. Install MPI on each of the slave processors, in the same way it is installed on the development and master computer.
4. Install the C++ Express compiler on all slave machines. The compiler does not need to be set up to compile code, but the local copy of the MPI program needs library files connected with the compiler.

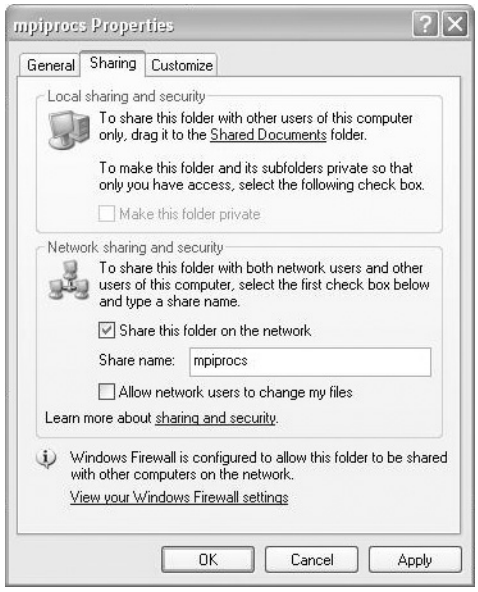


Figure 11.3: Turning on sharing for a directory containing the MPI executable.

5. Create a user account on all computers, one that has the same username and password. For this discussion, let the username be ''mpi'' and the password be ''mpi.'' Just to be safe, make sure that this account has administrator privileges.
6. Execute the MPI program named `wmpiregister.exe` (installed with the distribution) and register the username ''mpi'' and password ''mpi,'' as shown in Figure 11.4. Do this on each computer in the cluster.

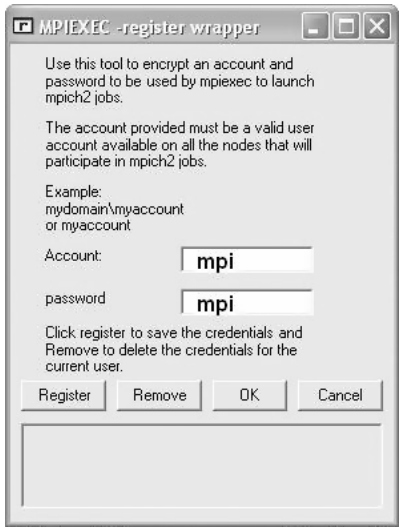


Figure 11.4: Registering the MPI account and password.

7. Run the MPI program `wmpiconfig.exe`. Adjust the settings as follows:
   - In the Hosts section, list the network names of all computers to be used as slaves. In Figure 11.5 there are two computers: ''hauptmann'' and ''studio02.''
   - Set a location and maximum size of the error log file. This log can be useful if something goes wrong during a run.
   - Set a time-out. This is the maximum time that a program will be allowed to execute, so the value should be much larger than the longest time the program is expected to take.
   - Set the phrase to the password used for the MPI user accounts. This makes it easy to recall.

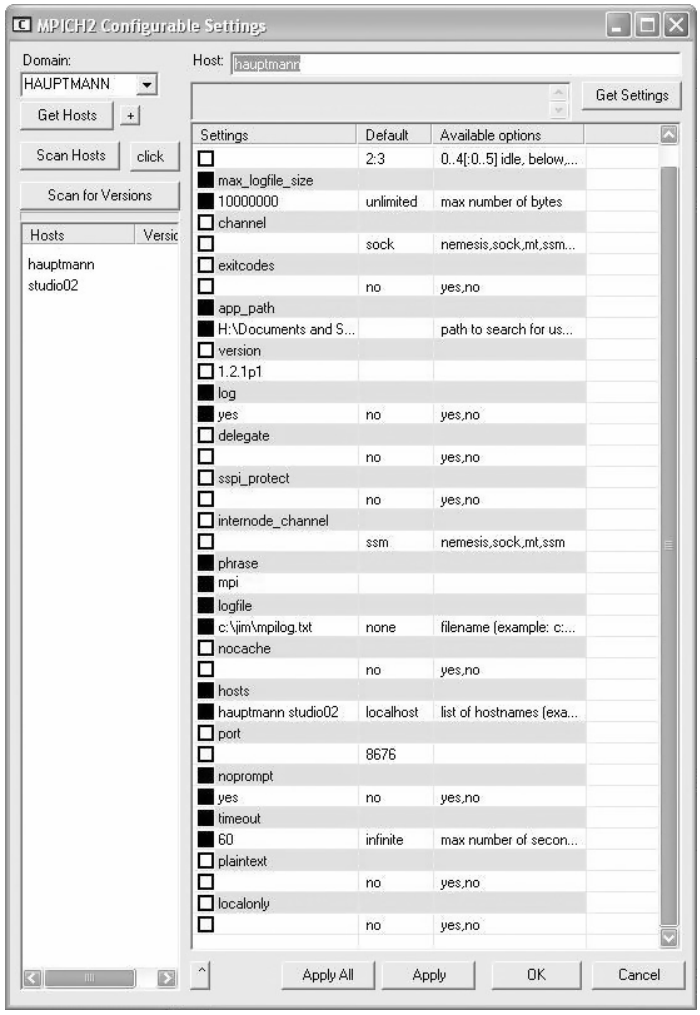


Figure 11.5: Using `wmpiconfig` to set up the MPI parameters.

Now it should be set up and ready to go. Running the program as before, using `wmpiexec`, will result in the system attempting to connect with the slave processors and executing the program on them. Errors are possible, perhaps likely; every PC system on the planet is different from the rest, or nearly so, and it's impossible to predict all interactions between software.

This is a lot of work, but some parts, such as moving the executable to a fixed location, can be automated.

Using a cluster produces unpredictable execution times. Sending data across the Ethernet and back takes a significant amount of time: an entire 1024x1024 image would need 0.08 seconds using a 100 MB network, and 0.08 seconds to return the processed pixels. This is on an idle network, and Ethernets can saturate quickly — at between 30% and 40% of max capacity, as a rule. This means that it would not be surprising to find that it actually took 0.16 seconds each way on a real network, or 0.32 seconds both ways. To show a profit in terms of real execution time, the time needed on one processor needs to be about three times the transmission time. That's if an improvement is to be made using two processors. For three processors, the needed execution time would be six times the transmission time, and in general 3*N times for N processors. This is a rule of thumb only, but it means that a cluster is effective only for very computationally intense problems.

So, how do you tell that ''studio02'' is actually running code? By watching the performance window in the Task Manager, as shown in Figure 11.6. The computer named ''studio02'' is pretty slow (a 1.2 Ghz Celeron), used simply to show the setup and operation of a small Windows-based cluster, and it usually operates at about 2% CPU usage when idle. When `mpiTest1` is executed from the master machine (''hauptmann''), the usage can be observed to rise to 100% for a moment while the computation is active, and then it drops off again.

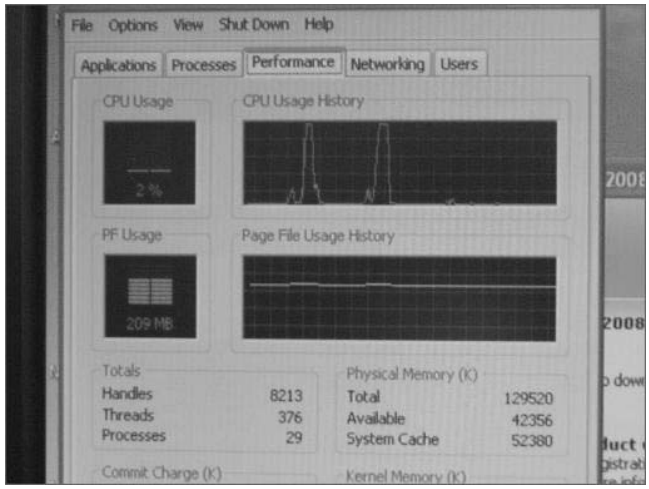


Figure 11.6: The performance meter of a slave processor while running an MPI task from the network.

11.4 A Shared Memory System—Using the PC Graphics Processor

Modern graphics processors are really remote descendents of the original PC graphics card of the 1990s. These days, the graphics processing unit (GPU) is a parallel processor with greater speed and memory bandwidth than the main CPU. Why? Because modern video games require a large degree of sophistication, a lot of memory, and a large degree of speed and power to be able to render 60 frames of high-resolution graphics each second. That's what a video game requires. A GPU costs between $100 and $600 and has the power of a supercomputer of a decade ago.

It has recently occurred to medical and scientific workers that a GPU can be used as a computer, and can process their data as fast or faster than can the CPU. It is true, but the fact remains that a GPU was designed to render three-dimensional computer models into two-dimensional images. The implication of the special purpose nature of a GPU is that in order to use it for more general computation, the problem must be described in terms of graphical operations. So, for example, it is normal for an image that is to be processed using a GPU to be saved as a texture, in graphics terms, and for the operations performed on it to be typical of those performed on a texture. So GPUs can be programmed with relative ease to perform interpolations and convolutions. In fact, as vector computers, GPUs can be programmed to do most things that require the same thing to be done on many different data objects (e.g., pixels) simultaneously. They implement a form of *data parallelism*. Sorting and searching would not be among those things that a GPU could do well.

Writing code for a GPU is painful. The paradigm is not a natural one for someone trained as a programmer of PCs and Sun workstations, in that the essence is one of the applications of a software pattern to many or all data points concurrently. Loops can't be eliminated, but are fewer and smaller. Operations and data flow are sometimes implicit. As a result, whole volumes are written about it, and only a taste can be given here. The hope is that the references provide enough detail to fill in the many gaps left by the explanations and examples that follow.

11.4.1 GLSL

The OpenGL Shading Language (*GLSL*) was introduced with OpenGL 2.0 in 2004. OpenGL itself is an application programming interface (API) for computer graphics that functions across computing platforms and graphics hardware manufacturers. It is used to render graphics on computers of all types, and is commonly used in applications from scientific visualization to

computer games. Most computers come with an up-to-date OpenGL library, whether the users know it or not.

The basic nature and operation of OpenGL will be discussed here briefly for two reasons. First, OpenGL is the framework within which GLSL operates — the envelope, if you will. Calls to the shader are made though OpenGL, and OpenGL initialization is an essential first step. Thus, its mode of operation, built-in constants, and basic syntax must be known. Second, the shader can only perform operations that are intended for use in rendering polygons. Any image-processing or vision algorithms implemented using GLSL must be expressed in terms of some graphical primitive, so it is important to know what those are and how to access them.

11.4.2 OpenGL Fundamentals

OpenGL will render objects that are defined as polygons. Complex objects such as cars and teapots are built from triangles or quadrilaterals linked together in three dimensions and then shaded so as to simulate a curved surface. Graphics processors these days still describe performance on the basis of the number of polygons they can process per second. There are multiple ways to draw polygons in OpenGL, and multiple ways to specify objects that consist of polygons. The basic scheme encloses a set of polygon descriptions, coded as procedure calls, between a `begin` and an `end` procedure call. Specifically, we could draw a triangle in the following manner (see also Figure 11.7):

```
glBegin (GL_TRIANGLES);
glVertex3f (x0, y0, z0);
glVertex3f (x1, y1, z1);
glVertex3f (x2, y2, z2);
glEnd();
```

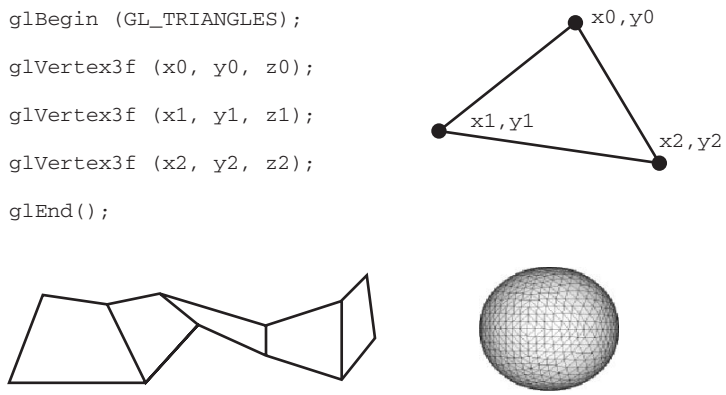


Figure 11.7: Drawing a triangle iusing OpenGL. All the actual rendering is done between glBegin and glEnd. Complex objects can be drawn as connected triangles.

The constant `GL_TRIANGLES` indicates that an independent set of triangles will be specified as vertices, between the `glBegin` and `glEnd` call. The calls to `glVertex3f` specify the coordinates of the vertices as three floating-point Cartesian coordinates: `x`, `y`, and `z`. Each three consecutive calls defines one triangle here because that is what `GL_TRIANGLES` means. Color is specified by calling, as one possibility, `glColor3f`, passing the R, G, and B values desired. This effectively sets the *current* color, and everything is drawn in that color until it is changed. The color values are between 0.0 and 1.0, so `glColor3f (1.0, 1.0, 1.0)` sets the current color to white, and `glColor3f (1.0, 0.0, 0.0)` sets it to red.

OpenGL maintains a stack of transformation matrices that define the viewing geometry, and is always currently using the one on top. Specifying modifications to the current transformation matrix is easy. Creating and pushing a new one has to be requested specially. Also, there are multiple stacks, one for each type of matrix: viewing, model transformations, and projection transformations. When manipulating the matrices, enter the correct mode using the `glMatrixMode` function: either `glMatrixMode(GL_MODELVIEW)` or `glMatrixMode(GL_PROJECTION)`. In `MODELVIEW` mode, we can rotate, scale, or translate our polygonal models before plotting them. In `PROJECTION` mode, we can change the way the scene looks.

The viewing operations involve defining the transformation, specifying the viewpoint, and specifying the clipping volume. For example, the use of an orthographic viewing projection can be specified by:

```
gluOrtho2D (0.0, N, N, 0.0);
```

which sets up a two-dimensional viewing transformation with the min (left) and max (right) X coordinates being the first two parameters (`0.0` and `N`), and the bottom and top coordinates being the final two. In this case, bottom > top, so Y runs in ascending values from the top of the image to the bottom.

Establishing a perspective viewing transformation is a matter of creating and pushing a perspective transformation onto the projection stack. The whole process is:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (90.0, 1.0, 1.0, 350.0);
```

The call to `glLoadIdentity()` pushes an identity matrix onto the current stack (projection stack, in this case). This new matrix is modified according to the parameters given to the `gluPerspective` call. Specifically, the first parameter is the field of view in the vertical direction, in degrees; the second parameter is the *aspect ratio*, the ratio of the x extend to the y extend in the

field of view; the last two parameters are the near and far clipping planes specified as Z coordinate, where `0` is the nearest possible, and positive values are increasing in distance.

This sets up the basic viewing transformation, but that's not usually enough. Where is the center of projection, or the place from which we are looking? In what direction are we looking? These can be specified by using the call:

```
gluLookAt (MyX, MyY, MyZ, dirx, diry, 0.0, 0.0,0.0, 1.0);
```

where `MyX`, `MyY`, `MyZ` are the current three-dimensional coordinates of the place from where we are looking; `dirx`, `diry`, `0.0` represent the direction we are looking, or the *viewing direction*, as defined in Figure 11.8; and the last three parameters represent the vector that identifies the vertical — in this case the vector (`0,0,1`), or the Z axis. This call will create a matrix that will be multiplied by the current one (i.e., top of stack) to give a resultant viewing matrix.

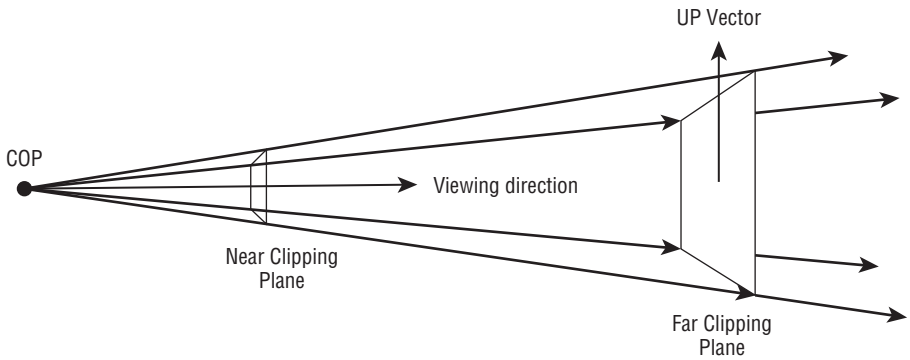


Figure 11.8: OpenGL clipping and viewing geometry.

Now we should define the clipping volume. Clipping can be turned on using:

```
glViewport (0.0, 0.0, 700.0, 700.0)
```

which specifies the coordinates, in two dimensions, of the part of the projected scene that can be seen. If you want to actually clip in three dimensions, then call:

```
glFrustrum (left, right, bottom, top, near, far)
```

where all the parameters are `double`. This specifies the six clipping planes as the relevant coordinate only; if Z is distance, for example, then we need only specify Z for the *near* and *far* clipping planes.

11.4.3 Practical Textures in OpenGL

Using textures is very important because images have to be treated like textures in order to process them on the GPU. Texture mapping in OpenGL involves three basic steps:

1. First, we have to read the texture into memory; textures are images and are stored in an image file format such as BMP or JPEG.
2. Next, we map the vertices of the polygon onto texture coordinates ''by hand.'' That is, every polygon vertex that we plot must be associated with a row and column coordinate in the texture image. Remember, the polygons are in three-dimensional space and the texture is in two-dimensional space.
3. Finally, we draw the polygons, covering them in the process with portions of the texture.

Once we have defined the mapping, as in step two, this part is almost automatic, taking place when we draw the polygons because OpenGL kept track of the polygons that had textures and what they were. So, really, texture mapping is all about setting up things so that it can be done automatically.

OpenGL thinks of textures as objects, and the image used to map onto a polygon is merely one property or parameter of the texture. Textures are referred to by names, which are in fact unique integers. OpenGL assigns these using:

```
glGenTextures (N , &textureObject)
```

which returns the name of a texture; if N=1, it returns the next integer we can use to define a new texture. In general, this function returns N such names. It (they) get(s) returned as the second parameter `textureObject`. This merely gives a unique ''name.'' A new texture object is given properties using two functions, and this means that the name is associated with a texture image, pixel color scheme and implementation, size, and so on.

First we must *bind* the name to a *target*. The official documentation on this subject seems to me to be a bit fuzzy, especially for beginners. Please bear with me or skip ahead if you are an advanced user, but I think the following is essential:

- **Only one target is important** — That target is called `GL_TEXTURE_2D`. It represents the two-dimensional texture that is being used at the present time. There are one- and three-dimensional targets that we will never use.
- **Only one texture is active at a time** — That is the texture currently bound to the target `GL_TEXTURE_2D`. If you map a texture, it will be the one bound

to GL_TEXTURE_2D. If you modify a texture parameter, it will be to the texture bound to GL_TEXTURE_2D.

An OpenGL function call binds a texture to a target:

```
glBindTexture (GL_TEXTURE_2D, textureObject)
```

Binding means to connect or to link. So, from now on, the name in textureObject, which is an integer, will be associated with a particular set of parameters referring to a texture. The name is really more like a *handle* than a name.

- **The binding can be redone** — Every time that glBindTexture is called, another texture object, specified by name, becomes the one currently active; it becomes bound to the target GL_TEXTURE_2D. If we want to use multiple textures, a very common thing to do, we must bind the texture we want to use before we plot the polygons to be texture mapped.
- **We can't change the binding while drawing** — Inside of a glBegin-glEnd block, a call to glBindTexture will cause an error. The error will not cause the program to halt, but the desired texture mapping will not occur.

Once we have a name bound to a texture object, we can give that object properties. Specifying the actual texture image is a simple matter (with complicated parameters):

```
glTexImage2D (GL_TEXTURE_2D, 0, 3, width, height, borderWidth,
                          pixelFormat, pixelType, image)
```

The first parameter is the kind of texture, always GL_TEXTURE_2D. The second parameter is called the *level of detail*, the usual level being 0. The third parameter specifies the number of color components in a texture pixel. This can be 1, 2, 3, or 4; the 3 specifier above actually means three components: r, g, and b.

The width and height parameters refer to the size of the image itself, and there are size limitations. Important fact: *Both must be a power of two plus the total size of a specified border*. The size of this border is the sixth parameter, borderWidth; it can be 0 or 1. The pixelFormat can be one of a few values, specified as constants, and represents the way pixels are represented. Usual for me is GL_RGB or GL_RGBA. The pixelType specifies how the pixel values are stored: reals, ints, chars, and so on. GL_UNSIGNED_BYTE is typical. Finally, image is a pointer to the pixels themselves, stored in the form described by the parameters.

This process assumes that the properties of the texture image are known (i.e., it has been read into memory). The call to glTexImage2D does not really do anything except note what the values of certain image parameters are and where the image can be found. The image is actually at least four parameters (image, location, sizes, format), and this one call sets them all. Setting other parameters is done more simply by calling:

```
glTexParameter (GL_TEXTURE_2D, pname, pvalue)
```

There are 16 texture parameters, but not all of them are equally important and the default values are often okay.

Now for the final step: drawing a texture mapped polygon. This involves specifying the texture coordinates of each vertex as it is drawn — that's all! In practice, here's how we draw a quad that is mapped to a texture:

```
glTexCoord2f (1.0, 0.0); glVertex3f (x0, y0, z0);
glTexCoord2f (1.0, 1.0); glVertex3f (x1, y1, z1);
glTexCoord2f (0.0, 1.0); glVertex3f (x2, y2, z2);
glTexCoord2f (0.0, 0.0); glVertex3f (x3, y3, z3);
```

Each texture coordinate, specified as an (x, y) index into the texture image, is associated with the vertex coordinate in three dimensions that follows. It's that simple.

So, the complete (more or less) code needed to map a texture onto parts of a cube is as follows:

```
/* Read the texture image */
    readIcon (&hedgeIcon, "hedge1.ppm");
/* Assign a name */
    glGenTextures (1, &hedgeTex);
/* Make it the current texture */
    glBindTexture (GL_TEXTURE_2D, hedgeTex);
/* Assign an image to the current texture */
    glTexImage2D (GL_TEXTURE_2D, 0, 3, 128, 128, 0, GL_RGB,
         GL_UNSIGNED_BYTE, &hedgeIcon[2]);
/* Filter for varying viewing distance */
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
         GL_NEAREST);
/* The environment setting - MODULATE */
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
         GL_MODULATE);
/* Switch on texture mapping */
    glEnable (GL_TEXTURE_2D);
    glBegin (GL_QUADS);                  /* START DRAWING */
/* East face of the cube */
    glTexCoord2f (1.0, 0.0); glVertex3f ((x0, y0,z0);
    glTexCoord2f (1.0, 1.0); glVertex3f (x1, y1, z1;
    glTexCoord2f (0.0, 1.0); glVertex3f (x2, y2, z2);
    glTexCoord2f (0.0, 0.0); glVertex3f (x3, y3, z3);
/* West face */
    glTexCoord2f (1.0, 0.0); glVertex3f (x0+WEST, y0, z0);
    glTexCoord2f (1.0, 1.0); glVertex3f (x1+WEST, y1, z1);
    glTexCoord2f (0.0, 1.0); glVertex3f (x2+WEST, y2, z2);
    glTexCoord2f (0.0, 0.0); glVertex3f (x3+WEST, y3, z3);
    glEnd();
```

This code appears to be the minimum needed to perform texture mapping in OpenGL; however, three things have not been mentioned yet:

1. We need to switch on texture mapping using a single procedure call, to `glEnable(GL_TEXTURE_2D)`.

2. There is an environment in which texture mapping takes place, and there are global parameters that specify how it is done. Essential parameters specify how the texture is placed on the surface: `GL_DECAL` pastes a texture over whatever is there; `GL_MODULATE` multiplies the background with the texture, so that they both appear to some extent. The function `glTexEnvf` allows us to alter environment parameters.
3. There are parameters that are filters, specifying how the texture changes scale as we get nearer or farther from it. The parameter `GL_TEXTURE_MIN_FILTER` specifies how a pixel being textured is mapped onto an area bigger than one texture element. In the preceding code, the call:

   ```
   glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   ```

   specifies that we use the value of the texture element that is nearest physically to the center of the pixel being mapped onto. This has to be specified or the mapping would not take place; perhaps the default is not appropriate.

11.4.4 Shader Programming Basics

OpenGL programs are like other C programs. Specifically, they make calls to the OpenGL library; otherwise, the flow of control is that of traditional C code, because that's what it is. Shader programs are not like C or other traditional programs. Yes, they can have variables and `if` statements and loops, but the way that information is sent to a shader program is a little strange, and there are many aspects of shader execution that are not clearly outlined by the documentation but that are not common programming practice and that must be understood. For example, the following are key aspects of shader code:

- **A shader program executes on all data elements in parallel** — Many of the programs that have been developed as examples in this book operate on the target image by examining each pixel in turn. Code that does this uses the following loop:

  ```
  for (i=0; i<height; i++)
            for (j=0; j<width; j++)
                 do something to a pixel.
  ```

  The equivalent shader program would be:

  ```
  do something to a pixel.
  ```

  There is no need to write code to traverse all pixels, because that's implicit; it's what the graphics processor was built for. And, of course, this is where the parallelism resides.
- **There are two important shader types** — The *vertex shader* operates on vertices, which can be thought of as pixels or as one of the corners of

a polygon, and executes on the vertex processor. The *fragment shader* operates on polygons and small sections of an image.

Each type of shader can have a program written for it in shader language; each has different characteristics, but both vertex and fragment shaders follow the same syntax. The two shaders are used for different sorts of graphic operations and are connected to the OpenGL framework through function calls. The vertex shader is used to transform vertices (e.g., brightness, color), manipulate normal vectors, generate and transform texture coordinates, and apply per vertex lighting. It knows about only one vertex at a time. The fragment shader executes on the fragment processor, and can do interpolations, apply textures, determine fog and blur, and determine transparency, among other things. It can use multiple pixels in the same neighborhood, and so is especially valuable in image processing because it can perform convolutions, distance computations, and connectivity determinations. In addition, the vertex shader can send data to the fragment shader, since vertices are the building blocks of polygons.

- **Shader programs are strings that are compiled when the host OpenGL program executes** — The user program treats the shader code as a character string, passing it to the shader ''compiler'' as a parameter. Of course, all computer programs are a string in some sense, but here it is one explicitly. Once the shader code is compiled, it can be executed repeatedly by using the already compiled version, but only until the program is complete. The next time the main program runs, the shader program will be recompiled.
- **Arguments are passed to shader programs by calling functions, which store them in fixed locations** — In this sense, a shader program is a bit like a device driver, where parameters are placed in fixed addresses in memory, and then the device is started by setting a specific bit someplace. The shader knows where to look for the value being passed and accesses it using another simple function call.

11.4.4.1 Vertex and Fragment Shaders

As mentioned in the preceding section, two shaders in the OpenGL/GLSL scheme correspond to the physical hardware on the GPU. The first shader in the data pipeline is the *vertex shader*, which computes the position of each vertex within the defined coordinate system (the space defined by the graphics clipping volume). The basic vertex shader looks like this:

```
void main()
{
     gl_Position = ftransform ();
}
```

where the `ftransform` function carries out the geometric viewing transformation on the vertex coordinates. This transformation is defined by the OpenGL matrix stack built by the programmer. An identical way to do this transform is:

```
void main()
{
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

GLSL's built-in `gl_Position` variable can be used to pass the transformed coordinates to the next stage in the pipeline, the next shader. A vertex shader without a reference to `gl_position` will not compile.

The next stage is the *fragment shader*, which computes the ultimate color or grey level, and sometimes depth for hidden surface removal, of a fragment. GLSL's built-in `gl_FragColor` variable can be used to define the final color. A trivial example of a fragment shader is:

```
void main()
{
     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

This will write a fully saturated red color (R=1.0; other components are 0) into the pixel. Colors are real values ranging between 0.0 and 1.0. A fragment shader can do much more, but the basic purpose is to assign a color to a point in the rendered image.

Both shaders can be passed variables; a vertex shader can pass information to a fragment shader; and both shaders can share global information, especially OpenGL variables (such as `gl_ModelViewProjectionMatrix`).

11.4.4.2 Required GLSL Initializations

The code for setting up a shader consists of the following seven steps, which are easily kept distinct from the other OpenGL code:

1. First, create a shader object for each shader that is involved. For most image-processing applications, there will be a vertex shader and a fragment shader. Creating a shader object is like declaring a variable — it simply marks an intention to use a shader and creates some space for one. It is accomplished by calling `glCreateShader`:

   ```
   GLuint vShader, fShader;
   vShader = glCreateShader(GL_VERTEX_SHADER);
   fShader = glCreateShader(GL_FRAGMENT_SHADER);
   ```

2. Now the source code must be read from a file and saved as a string. The strings are passed to the system by calling `glShaderSource`:

   ```
   vSource = readShader("C:\\AIPCV\\chap11\\convolve.vert");
   glShaderSource(vShader, 1, (const GLchar **)(&vSource), NULL);
   ```

3. At this point, the vertex shader object has source code. It is compiled using `glCompileShader`:

```
glCompileShader(vShader);
```

4. Any compilation errors are saved in a log. Any programmer knows how rare it is to get a program to compile correctly after a modification, so it's a good idea to check this and print out the log:

```
glGetShaderiv (vShader, GL_COMPILE_STATUS, &flag);
if(flag == GL_FALSE)
{
     printf ("Vertex shader program failed to compile.\n");
     log = (char *)malloc (2048);
     glGetShaderInfoLog (vShader, 2048, &siz, log);
     printf ("LOG: %s", log);
     free(log);
     exit(0);
}
```

5. Once the program(s) is compiled, a `program` object is created. This is a way to specify all the shader objects (programs) that are going to be used. There will ultimately be one `program` (the object being created in this step) that holds all shader objects. So, create the `program` object (just one):

```
program = glCreateProgram();
```

   and attach all the shader objects to it:

```
glAttachShader(program, fShader);
glAttachShader(program, vShader);
```

6. Now link the shaders to create a single executable program for the GPU:

```
glLinkProgram(program);
```

7. Before these programs are executed, they must be identified as the active shader programs. Many programs can be compiled and linked, but only one is active, and that's the one most recently specified by:

```
glUseProgram(program);
```

The main C/C++ program can thus switch between shader programs at execution time, even under user control.

11.4.5 Reading and Converting the Image

The image to be processed must be read in from a file, which is a relatively simple matter, and then must be sent to a storage place where the shaders can access it. Images must be stored as OpenGL textures. OpenGL uses textures to map onto objects so that they look like real-world objects. For example, mapping a brick texture onto a polygon wall makes it look like a brick wall.

The shader was designed to do a set of operations like texture mapping, so it is necessary to subvert these intentions and make the shader work on the target image.

OpenCV can be used to read an image from a file, but the pixel data is in the wrong form. It's a simple matter to fix it, though. OpenCV images have the colors in reverse order from that which OpenGL would like, and so they must be reversed. Also, some OpenGL textures are RGB format and others are RGBA, which includes an alpha (transparency) channel.

The steps in the process are as follows.

1. Read in an image:

```
image = cvLoadImage(name, 1);
Width = image->width; Height = image->height;
```

2. Allocate a block of memory for the texture pixels:

```
targetImage = (GLubyte *)malloc (image->width*image->height*4);
p = targetImage;
```

3. For all pixels, reverse the color bytes and copy them into this new buffer:

```
for (i = 0; i < image->height; i++)
          for (j = 0; j < image->width; j++)
          {
               s = cvGet2D (image, i, j);
               *p++ = (GLubyte) s.val[2];
               *p++ = (GLubyte) s.val[1];
               *p++ = (GLubyte) s.val[0];
//             *p++ = (GLubyte) s.val[4]; // For RGBA textures
          }
```

   At this point, the array `targetImage` holds the pixels in a way that is acceptable by OpenGL. Using `image->imageData` directly will work, but the reds and blues will be reversed. Now this buffer is set up as a texture, using the OpenGL functions already described.

4. Create a texture ID:

```
GLuint textn;
glGenTextures(1, &textn);
glBindTexture(GL_TEXTURE_2D, textn);
```

5. Set the parameters of this texture: size, nature, and data buffer:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, height, width, 0, GL_RGB,
             GL_UNSIGNED_BYTE, (const GLvoid *) targetImage);
```

6. Describe the graphical properties of the texture:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Now the image is ready to be mapped onto an object (polygon or set of polygons) in OpenGL.

11.4.6 Passing Parameters to Shader Programs

Once a shader program has been read in and compiled, all the variables declared within it are easy to find; the program is, after all, just a character string. Certain kinds of variables, noticeably `uniform` variables, are defined to be input arguments to the program. A value is placed into such a variable by the main C program before the shader program executes.

A simple example (from the `convolve2.frag` program on the website) passes the size of an image to a convolution program. The beginning of the program is:

```
varying vec2 TexCoord;

#define KERNEL_SIZE 9
float kernel [KERNEL_SIZE];

uniform sampler2D colorMap;
uniform float Width;
uniform float Height;
float dw, dh;
vec2 offset[KERNEL_SIZE];

void main (void)
{
             ....
```

The first declaration, for `TexCoord`, shows it to be `varying`; this means that it is being received from the vertex shader. A few lines below there are three `uniform` declarations: a `sampler2D`, which is a texture (an image), and two *floats*, `Width` and `Height`. These are being passed from the main C code.

The calling C program does the following to set up the passing of the float values:

```
k = glGetUniformLocation(shaderProgram, "Width");
```

This call gets the location in which to place the value for the `Width` parameter. Note that the name of the parameter is specified here; it is found in the program and its location is returned.

```
if (k<0) printf ("Error: Name 'Width' not found in shader program.\n", k);
```

If `-1` is returned, it means that the variable name was not found. Variables are not found if they are not actually used.

Finally, the following:

```
glUniform1f(k, (float)Width);
```

sets the value at the location `k` (i.e., where ''width'' is in the fragment shader) to the value of the `Width` variable, which is in fact `image->Width` for the image that was read in. Similarly, for `Height`:

```
k = glGetUniformLocation(shaderProgram, "Height");
printf ("Error: Name 'Height' is at %d in shader program.\n", k);
glUniform1f(k, (float)Height);
```

The variable of type `sampler2D` is a different kind of thing. A *sampler* is a handle (i.e., a pointer to a structure that describes some object) that allows access to a texture. Its implementation is mysterious and the details are not necessary to know, but knowing how to connect textures to samplers is important. Samplers are always input and cannot be modified within a shader. An image will be of type `sampler2D`, or a two-dimensional texture. The only use for a sampler inside of a fragment shader is to identify a texture to a function (a *texture lookup function*) that can look up pixel values; for example, the function `texture2D (A, B)` takes a `uniform2D` value named `A` and a pair of indices, `B`, (a two-dimensional vector giving row and column coordinates of a point in the texture) and returns the pixel value corresponding:

```
texLoc = glGetUniformLocation(shaderProgram, "colorMap");
printf ("Name 'colorMap' is at %d in shader program.\n", texLoc);
glUniform1i(texLoc, 0);
```

Passing a value of `0` here tells the shader to use `GL_TEXTURE0` as the texture being passed, and, in this case, this corresponds to the image being processed.

Many types can be passed to a shader, including vectors and matrices.

11.4.7 Putting It All Together

Let's finish this discussion of programming by writing a program to sharpen an image using a simple convolution filter. The GPU will convolve the input (color) image with the following 3 x 3 matrix:

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{matrix}$$

The advantage of this problem is that the code can be easily and quickly changed to perform nearly any convolution.

The main program, provided on the website as `shader2.c`, is based on the vertex and fragment shader programs above and is organized as follows:

1. Set up the OpenGL GLUT declarations to open windows and display images, or use another facility to display images. Recall that openCV

can't be used. This detail is covered elsewhere and is a common first step in any OpenGL program.

2. Initialize a utility called GLEW, which is used to manage the complex code found in the large array of OpenGL libraries. Think of it as a version management system. It is not absolutely necessary, but it simplifies the process for beginners.

   These two steps are cookbook code. Details can be found in the references if more information is wanted.

3. Read the image (see Section 11.4.1.6) and set it up as a texture. The program provided sets up two textures, as writing to the same buffer as one is reading from is a bad idea.
4. Read and initialize the shader programs (see Section 11.4.1.5)
5. Run the main loop (`glutMainLoop`), which results in the image being mapped as a texture on a rectangle and displayed in a window. When this is done, the shader programs operate on all pixels before the display occurs. The `render` function is a callback that is used to display the result. After setting up the viewing coordinate system, this function binds the texture (see Section 11.4.1.2), specifies the shader program using `glUseProgram` (see Section 11.4.1.5), and sets up the parameters to be passed (see Section 11.4.1.7). The rendering is done in between the `glBegin()` and `glEnd()` calls, where the polygon mapped with the image is drawn.

Figure 11.9 shows an example of this program's results. The original image is displayed in initial form, the CPU processed version is shown in a distinct window, and the GPU processed image is in a third window.

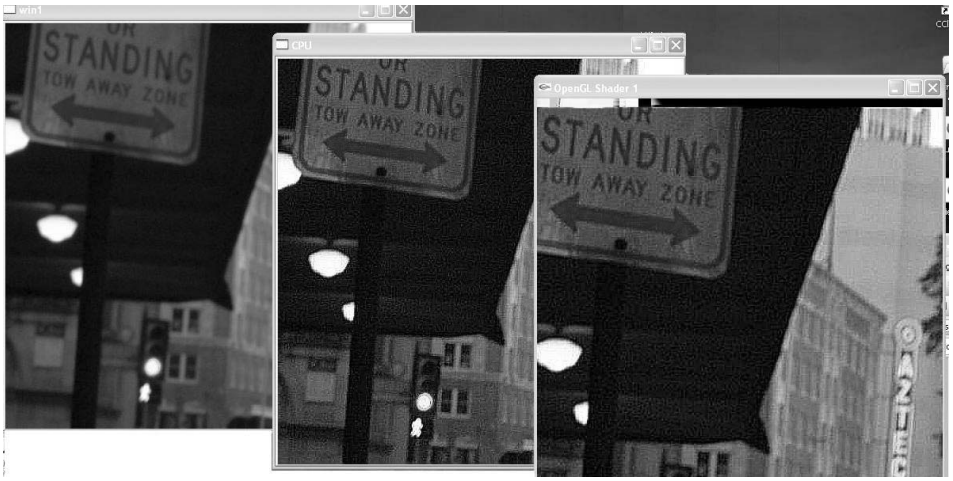


Figure 11.9: Image of a street in downtown San Antonio sharpened using a convolution mask (unsharp masking) left to right: by the CPU, and by the GPU, an ATI Radeon HD5670. The GPU is 200 times faster.

11.4.8 Speedup Using the GPU

The program `shader2` on the website is instrumented to show execution time. The CPU code used to implement the contract enhancement is timed using `QueryPerformanceCounter`, and so is the `render` function, which uses the shader code to accomplish the same task. Each time render is called the time is added, and when the program quits, the average over all executions — usually many thousands — is calculated and printed. The results are astonishing; over three trials the execution times are:

CPU execution time =	0.612	0.444	0.534 seconds
GPU execution time =	0.0020	0.0022	0.0015 seconds

It is plain to see why the extra time needed to program a shader might be worth it. In the worst case, the GPU was more than 200 times faster than the CPU for this task. There seems to be some sort of initialization effect showing in the GPU timings, as the first runs are slower than the others. Stopping the program and running it again 30 minutes later gives a longer execution time again. Over long runs, the time averages to well under 0.002 seconds.

The sample shader and C driver programs provided in this chapter still have a lot of magic in them for people unfamiliar with OpenGL, but they should provide a basis for experimentation. When in doubt about how the shader works, test it by writing a small program. Documentation for GLSL and many other sample programs exists on the Internet, but be aware that there are two main versions of GLSL out there, so don't get them confused. Programs with a lot of ARB suffixes represent a previous version than the one used here.

11.4.9 Developing and Testing Shader Code

It is difficult to write and test shader code, because it is compiled during the execution time of the main program. Compile errors are detected after an image is read in, maybe after it is displayed, and do not provide clear messages or runtime debugging. Fortunately, there are now shader software development kits (SDKs) that include an editor and runtime environment. Using one of these tools, code can be typed into a window and compiled at a keystroke, and compilation errors can be fixed within seconds. Graphics windows are usually part of such a tool. In the spirit of this book (that is to say, tools are free), one such tool, *ShaderDesigner* by Typhoon Labs (`www.opengl.org/sdk/tools/ShaderDesigner/`), is introduced briefly.

Figure 11.10 shows a screen typical of this system. Images (textures) can be loaded using a pop-up window and mouse selection, eliminating the need to write the tedious code required to load images from a path and convert them into the right form. Vertex and fragment shader programs can be typed in to the main text frame and saved as text files for use in the real application. The code can be compiled using a mouse click (or by pressing the F4 key), and the result of the code operating on the image is displayed on the left.

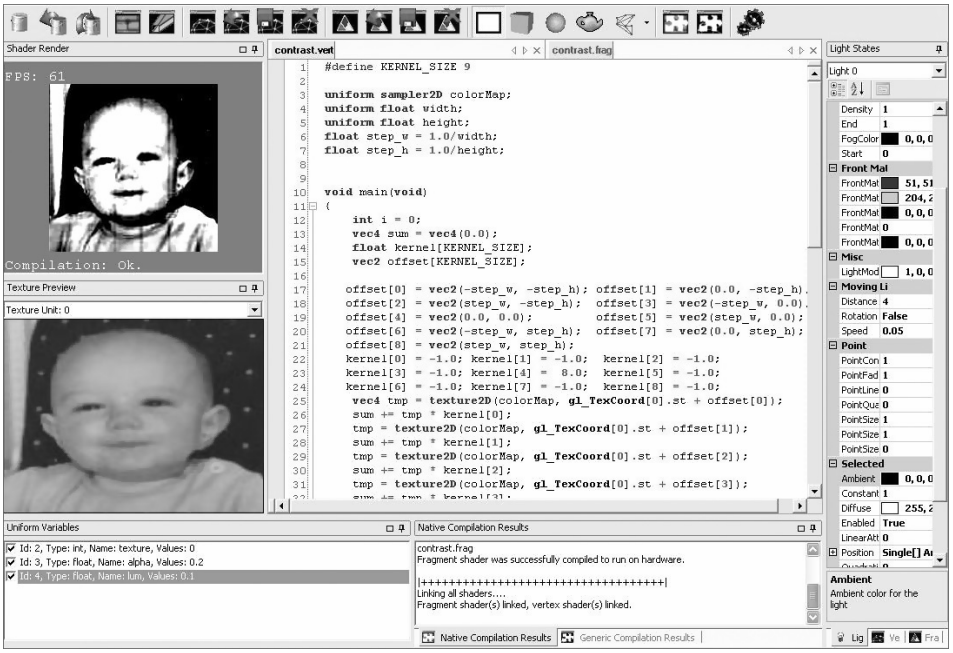


Figure 11.10: The ShaderDesigner main window, showing the basic display and controls.

Best of all, compilation error messages are listed at the lower right of the window, meaning that errors can be found and corrected quickly. This is almost like the Visual Studio Express SDK in basic function. The system can be downloaded and installed in a few minutes on Windows systems that have OpenGL already installed.

11.5 Finding the Needed Software

All the software discussed in this chapter can be found on the Internet and is free to download. This chapter has used a lot of new items, which need to be found, downloaded, installed, and the compiler needs to be set up to use it. A brief description of what is needed would be in order.

Installation of software depends on the system and the compiler. Directions should be available on the website for the package. In general, the include file directory and the library file directory for the package need to be added to the compiler search paths; the library (a `.lib` file) needs to be added to the library dependencies; and a `.dll` file or two (on a PC) needs to be placed in a Windows directory. On Linux and Mac systems, the details vary.

- **MPI** — Can be downloaded from `www.mcs.anl.gov/research/projects/mpich2/`. Refer to Section 11.3.1 for details about installing MPI.

- **OpenGL** — Is normally found on Windows PCs and Apple computers. A good source of documentation and tutorials is `www.opengl.org/`. Recent versions can be downloaded from the website that is specific for the graphics card (usually `http://developer.amd.com/pages/default.aspx` or `http://developer.nvidia.com/object/opengl_driver.html`).
- **GLUT** — A window system-independent toolkit for writing OpenGL programs, downloadable from `www.xmission.com/~nate/glut.html`.
- **GLEW** — A cross-platform open-source C/C++ extension-loading library. It provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. Downloadable from `http://glew.sourceforge.net/`.

11.6 Website Files

`clock1.c`	Shows the use of the `clock` function for timing execution
`clock2.c`	Uses `QueryPerformanceCounter` to time code execution
`mpiTest1.c`	Demo of MPI program, with timing
`mpiTest2.c`	A 13-point median filter using MPI to distribute the work over multiple processors
`shader1.c`	Parallel image processing using the GPU; convolution for image sharpening
`shader2.c`	Image processing with timing, passing image size as parameter
`mpiTest1.c`	Executable of `mpiTest1.c`
`convolve.vert`	Basic vertex shader program
`convolve1.frag`	Fragment shader, convolution of 512x512 image
`convolve2.frag`	Fragment shader, convolution, size passed as arguments
`pic3.jpg`	Image used for testing — a street scene in San Antonio, Texas

11.7 References

Andrews, G. R. *Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley, 2000.

Bailey, M. ''Intro to GLSL,'' teaching slides, Oregon State University, 2006.

Barney, B. ''OpenMP,'' Lawrence Livermore National Laboratory, last modified March 6, 2010, `https://computing.llnl.gov/tutorials/openMP/`.

Buck, I., and T. Purcell. ''A Toolkit for Computation on GPUs,'' *in GPU Gems*, ed. R. Fernando. Reading MA: Addison-Wesley, 2004, 621–636.

Dowd, K., and C. Severance. *High Performance Computing*, Second Edition. Cambridge: O'Reilly, 1998.

Fung, J. ''Computer Vision on the GPU,'' in *GPU Gems 2*, ed. M. Pharr. Reading, MA: Addison-Wesley, 2005, 651–667.

Fung, J., and S. Mann. ''Using Multiple Graphics Cards as a General Purpose Parallel Computer: Applications to Computer Vision,'' *Proceedings of the 17th International Conference on Pattern Recognition*, Cambridge, United Kingdom, 2004, 805–808.

Galloway, N. ''Getting OpenGL to Work With Visual C++,'' *Thoughts From My Life*, 2008, `http://thoughtsfrommylife.com/article-748-OpenGL_and_Visual_Studio_Express_2008`.

Kessenich, J. ''The OpenGL Shading Language,'' document revision 8, September 7, 2006, `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`.

Lindsay C. ''Intro to GPU Programming (OpenGL Shading Language),'' talk slides. Accessed May 2010, `http://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/gpu.pdf`.

Marroquim, R., and A. Maximo. ''Introduction to GPU Programming with GLSL,'' *Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, Rio de Janeiro, IOP Publishing (UK) Brazil: 2009.

Matsuoka, S., T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawam. ''GPU Accelerated Computing — From Hype to Mainstream, the Rebirth of Vector Computing,'' *Journal of Physics: Conference Series* 180 (2009).

McClelland, J. L., and D. E. Rummelhart. *Parallel Distributed Processing — Explorations in the Microstructures of Cognition. Volume 1: Foundations*. Cambridge, MA: The MIT Press, 1986.

McClelland, J. L., and D. E. Rummelhart. *Parallel Distributed Processing — Explorations in the Microstructures of Cognition. Volume 2: Psychological and Biological Models*. Cambridge, MA: The MIT Press, 1986.

Molaro, D., and J. R. Parker. ''A Distributed Thinning Algorithm on a Workstation Network,'' in *Parallel Programming and Applications*, ed. Fritzson and L. Finmo. Amsterdam, Netherlands: IOS Press, 1995, 195–202.

Molaro, D., and J. R. Parker. ''Distributed Programming Using Objects — A Case Study,'' *Third Golden West International Conference on Intelligent Systems*, Las Vegas, June 6–9, 1994.

O'Connor, K. ''Intro GLSL,'' teaching slides, GV2, University of Dublin, 2006.

Olano, M., and W. Heidrich. *Real-Time Shading*, ed. J. C. Hart, M. McCool. Natick MA: AK Peters, Ltd, 2002.

OpenGL SourceForge.net. The OpenGL Extension Wrangler Library (GLEW). Accessed May 20, 2010, `http://glew.sourceforge.net/index.html`.

OpenMP Architecture Review Board. ''Summary of OpenMP 3.0 C/C++ Syntax,'' 2008, http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf.

Parker J. R. ''A Multiple/Parallel System For Recognizing Handprinted Digits,'' *Vision Interface 1997*, Kelowna, B.C., Canadian Image Processing and Pattern Recognition Society May 20–22, 1997.

Parker, J. R. *Start Your Engines — Developing Driving and Racing Games*. Scottsdale,: Paraglyph Press, 2005.

Rost, R. *OpenGL Shading Language,* (''Orange Book''). London: Addison -Wesley, 2004.

Segal, M., and K. Akeley. ''The OpenGL Graphics System: A Specification (Version 2.0),'' 2004, http://www.opengl.org/documentation/glsl/2010/11/10 glspec20.pdf.

Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI — The Complete Reference,* Volume 1: *The MPI Core,* 2nd edition. Cambridge, MA: MIT Press, 1999.

Talton, J. ''OpenGL Shading Language,'' teaching slides, Stanford University, 2006.

van der Pas, R. ''An Introduction Into OpenMP,'' *IWOMP 2005*, Eugene, OR: University of Oregon, 2005, http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf.

Villar, J. R. ''Chapter 1 — Introduction to GLSL,'' *OpenGL Shading Language Course,* TyphoonLabs, 2009, http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_1.pdf.

Index

B

C

D

E

F

G

H

L

M

N

O

P

S

T

U

V

W

X

Z