Still Image and Video Compression with MATLAB®

K.S. THYAGARAJAN

WILEY

STILL IMAGE AND VIDEO COMPRESSION WITH MATLAB

STILL IMAGE AND VIDEO COMPRESSION WITH MATLAB

K. S. Thyagarajan

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Library of Congress Cataloging-in-Publication Data:

Thyagarajan, K. S.
  Still image and video compression with MATLAB / K.S. Thyagarajan.
    p. cm.
  ISBN 978-0-470-48416-6 (hardback)
 1. Image compression. 2. Video compression. 3. MATLAB. I. Title.
  TA1638.T48 2010
  006.6′96–dc22

2010013922

Printed in Singapore

oBook ISBN: 978-0-470-88692-2
ePDF ISBN: 978-0-470-88691-5

10 9 8 7 6 5 4 3 2 1

To my wife Vasu,
who is the inspiration behind this book

CONTENTS

PREFACE

The term “video compression” is now a common household name. The field of still image and video compression has matured to the point that it is possible to watch movies on a laptop computer. Such is the rapidity at which technology in various fields has advanced and is advancing. However, this creates a need for some to obtain at least a simple understanding behind all this. This book attempts to do just that, to explain the theory behind still image and video compression methods in an easily understandable manner. The readers are expected to have an introductory knowledge in college-level mathematics and systems theory.

The properties of a still image are similar to those of a video, yet different. A still image is a spatial distribution of light intensity, while a video consists of a sequence of such still images. Thus, a video has an additional dimension—the temporal dimension. These properties are exploited in several different ways to achieve data compression. A particular image compression method depends on how the image properties are manipulated.

Due to the availability of efficient, high-speed central processing units (CPUs), many Internet-based applications offer software solutions to displaying video in real time. However, decompressing and displaying high-resolution video in real time, such as high-definition television (HDTV), requires special hardware processors. Several such real-time video processors are currently available off the shelf. One can appreciate the availability of a variety of platforms that can decompress and display video in real time from the data received from a single source. This is possible because of the existence of video compression standards such as Moving Picture Experts Group (MPEG).

This book first describes the methodologies behind still image and video compression in a manner that is easy to comprehend and then describes the most popular standards such as Joint Photographic Experts Group (JPEG), MPEG, and advanced video coding. In explaining the basics of image compression, care has been taken to keep the mathematical derivations to a minimum so that students as well as practicing professionals can follow the theme easily. It is very important to use simpler mathematical notations so that the reader would not be lost in a maze. Therefore, a sincere attempt has been made to enable the reader to easily follow the steps in the book without losing sight of the goal. At the end of each chapter, problems are offered so that the readers can extend their knowledge further by solving them.

Whether one is a student, a professional, or an academician, it is not enough to just follow the mathematical derivations. For longer retention of the concepts learnt, one must have hands-on experience. The second goal of this book, therefore, is to work out real-world examples through computer software. Although many computer programming languages such as C, C++, Java, and so on are available, I chose MATLAB as the tool to develop the codes in this book. Using MATLAB to develop source code in order to solve a compression problem is very simple, yet it covers all grounds. Readers do not have to be experts in writing cleaver codes. MATLAB has many built-in functions especially for image and video processing that one can employ wherever needed. Another advantage of MATLAB is that it is similar to the C language. Furthermore, MATLAB SIMULINK is a very useful tool for actual simulation of different video compression algorithms, including JPEG and MPEG.

The organization of the book is as follows.

Chapter 1 makes an argument in favor of compression and goes on to introduce the terminologies of still image and video compression.

However, one cannot process an image or a video before acquiring data that is dealt within Chapter 2. Chapter 2 explains the image sampling theory, which relates pixel density to the power to resolve the smallest detail in an image. It further elucidates the design of uniform and nonuniform quantizers used in image acquisition devices. The topic of sampling using nonrectangular grids—such as hexagonal sampling grids—is not found in most textbooks on image or video compression. The hexagonal sampling grids are used in machine vision and biomedicine. Chapter 2 also briefly demonstrates such a sampling technique with an example using MATLAB code to convert an image from rectangular to a hexagonal grid and vice versa.

Image transforms such as the discrete cosine transform and wavelet transform are the compression vehicles used in JPEG and MPEG standards. Therefore, unitary image transforms are introduced in Chapter 3. Chapter 3 illustrates the useful properties of such unitary transforms and explains their compression potential using several examples. Many of the examples presented also include analytical solutions.

The theory of wavelet transform has matured to such an extent that it is deemed necessary to devote a complete chapter to it. Thus, Chapter 4 describes the essentials of discrete wavelet transform (DWT), its type such as orthogonal and biorthogonal transforms, efficient implementation of the DWT via subband coding, and so on. The idea of decomposing an image into a multilevel DWT using octave-band splitting is developed in Chapter 4 along with examples using real images.

After introducing the useful compression vehicles, the method of achieving mathematically lossless image compression is discussed in Chapter 5. Actually the chapter starts with an introduction to information theory, which is essential to gauge the performance of the various lossless (and lossy) compression techniques. Both Huffman coding and arithmetic coding techniques are described with examples illustrating the methods of generating such codes. The reason for introducing lossless compression early on is that all lossy compression schemes employ lossless coding as a means to convert symbols into codes for transmission or storage as well as to gain additional compression.

The first type of lossy compression, namely, the predictive coding, is introduced in Chapter 6. It explains both one-dimensional and two-dimensional predictive coding methods followed by the calculation of the predictor performance gain with examples. This chapter also deals with the design of both nonadaptive and adaptive differential pulse code modulations, again with several examples.

Transform coding technique and its performance are next discussed in Chapter 7. It also explains the compression part of the JPEG standard with an example using MATLAB.

The wavelet domain image compression topic is treated extensively in Chapter 8. Examples are provided to show the effectiveness of both orthogonal and biorthogonal DWTs in compressing an image. The chapter also discusses the JPEG2000 standard, which is based on wavelet transform.

Moving on to video, Chapter 9 introduces the philosophy behind compressing video sequences. The idea of motion estimation and compensation is explained along with subpixel accurate motion estimation and compensation. Efficient techniques such as hierarchical and pyramidal search procedures to estimate block motion are introduced along with MATLAB codes to implement them. Chapter 9 also introduces stereo image compression. The two images of a stereo image pair are similar yet different. By using motion compensated prediction, the correlation between the stereo image pair is reduced and hence compression achieved. A MATLAB-based example illustrates this idea clearly.

The concluding chapter, Chapter 10, describes the video compression part of the MPEG-1, -2, -4, and H.264 standards. It also includes examples using MATLAB codes to illustrate the standards' compression mechanism. Each chapter includes problems of increasing difficulty to help the students grasp the ideas discussed.

I thank Dr. Kesh Bakhru and Mr. Steve Morley for reviewing the initial book proposal and giving me continued support. My special thanks to Dr. Vinay Sathe of Multirate Systems for reviewing the manuscript and providing me with valuable comments and suggestions. I also thank Mr. Arjun Jain of Micro USA, Inc., for providing me encouragement in writing this book. I wish to acknowledge the generous and continued support provided by The MathWorks in the form of MATLAB software. This book would not have materialized but for my wife Vasu, who is solely responsible for motivating me and persuading me to write this book. My heart-felt gratitude goes to her for patiently being there during the whole period of writing this book. She is truly the inspiration behind this work.

K. S. Thyagarajan

San Diego, CA

1 INTRODUCTION

This book is all about image and video compression. Chapter 1 simply introduces the overall ideas behind data compression by way of pictorial and graphical examples to motivate the readers. Detailed discussions on various compression schemes appear in subsequent chapters. One of the goals of this book is to present the basic principles behind image and video compression in a clear and concise manner and develop the necessary mathematical equations for a better understanding of the ideas. A further goal is to introduce the popular video compression standards such as Joint Photographic Experts Group (JPEG) and Moving Picture Experts Group (MPEG) and explain the compression tools used by these standards. Discussions on semantics and data transportation aspects of the standards will be kept to a minimum. Although the readers are expected to have an introductory knowledge in college-level mathematics and systems theory, clear explanations of the mathematical equations will be given where necessary for easy understanding. At the end of each chapter, problems are given in an increasing order of difficulty to make the understanding firm and lasting.

In order for the readers of this book to benefit further, MATLAB codes for several examples are included. To run the M-files on your computers, you should install MATLAB software. Although there are other software tools such as C++ and Python to use, MATLAB appears to be more readily usable because it has a lot of built-in functions in various areas such as signal processing, image and video processing, wavelet transform, and so on, as well as simulation tools such as MATLAB Simulink. Moreover, the main purpose of this book is to motivate the readers to learn and get hands on experience in video compression techniques with easy-to-use software tools, which does not require a whole lot of programming skills. In the remainder of the chapter, we will briefly describe various compression techniques with some examples.

1.1 WHAT IS SOURCE CODING?

Images and videos are moved around the World Wide Web by millions of users almost in a nonstop fashion, and then, there is television (TV) transmission round the clock. Analog TV has been phased out since February 2009 and digital TV has taken over. Now we have the cell phone era. As the proverb *a picture is worth a thousand words* goes, the transmission of these visual media in digital form alone will require far more bandwidth than what is available for the Internet, TV, or wireless networks. Therefore, one must find ways to format the visual media data in such a way that it can be transmitted over the bandwidth-limited TV, Internet, and wireless channels in real time. This process of reducing the image and video data so that it fits into the available limited bandwidth or storage space is termed *data compression*. It is also called *source coding* in the communications field. When compressed audio/video data is actually transmitted through a transmission channel, extra bits are added to it to counter the effect of noise in the channel so that errors in the received data, if present, could be detected and/or corrected. This process of adding additional data bits to the compressed data stream before transmission is called *channel coding*. Observe that the effect of reducing the original source data in source coding is offset to a small extent by the channel coding, which adds data rather than reducing it. However, the added bits by the channel coder are very small compared with the amount of data removed by source coding. Thus, there is a clear advantage of compressing data.

We illustrate the processes of compressing and transmitting or storing a video source to a destination in Figure 1.1. The source of raw video may come from a video camera or from a previously stored video data. The source encoder compresses the raw data to a desired amount, which depends on the type of compression scheme chosen. There are essentially two categories of compression—*lossless* and *lossy*. In a lossless compression scheme, the original image or video data can be recovered exactly. In a lossy compression, there is always a loss of some information about the

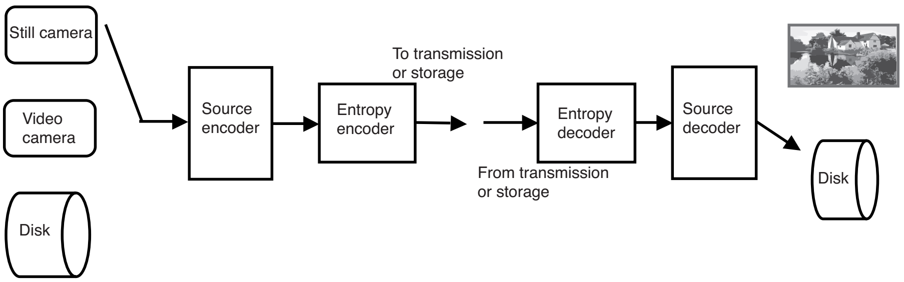


Figure 1.1 Source coding/decoding of video data for storage or transmission.

original data and so the recovered image or video data suffers from some form of distortion, which may or may not be noticeable depending on the type of compression used. After source encoding, the quantized data is encoded losslessly for transmission or storage. If the compressed data is to be transmitted, then channel encoder is used to add redundant or extra data bits and fed to the digital modulator. The digital modulator converts the input data into an RF signal suitable for transmission through a communications channel.

The communications receiver performs the operations of demodulation and channel decoding. The channel decoded data is fed to the entropy decoder followed by source decoder and is finally delivered to the sink or stored. If no transmission is used, then the stored compressed data is entropy decoded followed by source decoding as shown on the right-hand side of Figure 1.1.

1.2 WHY IS COMPRESSION NECESSARY?

An image or still image to be precise is represented in a computer as an array of numbers, integers to be more specific. An image stored in a computer is called a digital image. However, we will use the term image to mean a digital image. The image array is usually two dimensional (2D) if it is black and white (BW) and three dimensional (3D) if it is a color image. Each number in the array represents an intensity value at a particular location in the image and is called a picture element or pixel, for short. The pixel values are usually positive integers and can range between 0 and 255. This means that each pixel of a BW image occupies 1 byte in a computer memory. In other words, we say that the image has a grayscale resolution of 8 bits per pixel (bpp). On the other hand, a color image has a triplet of values for each pixel: one each for the red, green, and blue primary colors. Hence, it will need 3 bytes of storage space for each pixel. The captured images are rectangular in shape. The ratio of width to height of an image is called the aspect ratio. In standard-definition television (SDTV) the aspect ratio is 4:3, while it is 16:9 in a high-definition television (HDTV). The two aspect ratios are illustrated in Figure 1.2, where Figure 1.2a corresponds to an aspect ratio of 4:3 while Figure 1.2b corresponds to the same picture with an aspect ratio of 16:9. In both pictures, the height in inches remains the same,

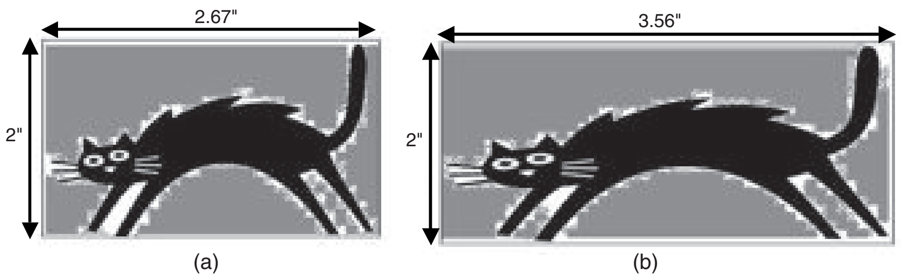


Figure 1.2 Aspect ratio: (a) 4:3 and (b) 16:9. The height is the same in both the pictures.

which means that the number of rows remains the same. So, if an image has 480 rows, then the number of pixels in each row will be 480 × 4/3 = 640 for an aspect ratio of 4:3. For HDTV, there are 1080 rows and so the number of pixels in each row will be 1080 × 16/9 = 1920. Thus, a single SD color image with 24 bpp will require 640 × 480 × 3 = 921,600 bytes of memory space, while an HD color image with the same pixel depth will require 1920 × 1080 × 3 = 6,220,800 bytes. A video source may produce 30 or more frames per second, in which case the raw data rate will be 221,184,000 bits per second for SDTV and 1,492,992,000 bits per second for HDTV. If this raw data has to be transmitted in real time through an ideal communications channel, which will require 1 Hz of bandwidth for every 2 bits of data, then the required bandwidth will be 110,592,000 Hz for SDTV and 746,496,000 Hz for HDTV. There are no such practical channels in existence that will allow for such a huge transmission bandwidth. Note that dedicated channels such as HDMI capable of transferring uncompressed data at this high rate over a short distance do exist, but we are only referring to long-distance transmission here. It is very clear that efficient data compression schemes are required to bring down the huge raw video data rates to manageable values so that practical communications channels may be employed to carry the data to the desired destinations in real time.

1.3 IMAGE AND VIDEO COMPRESSION TECHNIQUES

1.3.1 Still Image Compression

Let us first see the difference between *data compression* and *bandwidth compression*. Data compression refers to the process of reducing the digital source data to a desired level. On the other hand, bandwidth compression refers to the process of reducing the analog bandwidth of the analog source. What do we really mean by these terms? Here is an example. Consider the conventional wire line telephony. A subscriber's voice is filtered by a lowpass filter to limit the bandwidth to a nominal value of 4 kHz. So, the channel bandwidth is 4 kHz. Suppose that it is converted to digital data for long-distance transmission. As we will see later, in order to reconstruct the original analog signal that is band limited to 4 kHz exactly, sampling theory dictates that one should have at least 8000 samples per second. Additionally, for digital transmission each analog sample must be converted to a digital value. In telephony, each analog voice sample is converted to an 8-bit digital number using *pulse code modulation* (PCM). Therefore, the voice data rate that a subscriber originates is 64,000 bits per second. As we mentioned above, in an ideal case this digital source will require 32 kHz of bandwidth for transmission. Even if we employ some form of data compression to reduce the source rate to say, 16 kilobits per second, it will still require at least 8 kHz of channel bandwidth for real-time transmission. Hence, data compression does not necessarily reduce the analog bandwidth. Note that the original analog voice requires only 4 kHz of bandwidth. If we want to compress bandwidth, we can simply filter the analog signal by a suitable filter with a specified cutoff frequency to limit the bandwidth occupied by the analog signal.

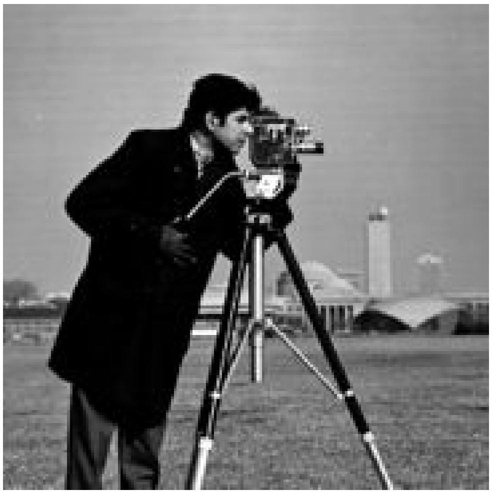


Figure 1.3 Original cameraman picture.

Having clarified the terms data compression and bandwidth compression, let us look into some basic data compression techniques known to us. Henceforth, we will use the terms compression and data compression interchangeably. All image and video sources have redundancies. In a still image, each pixel in a row may have a value very nearly equal to a neighboring pixel value. As an example, consider the cameraman picture shown in Figure 1.3. Figure 1.4 shows the profile (top figure)

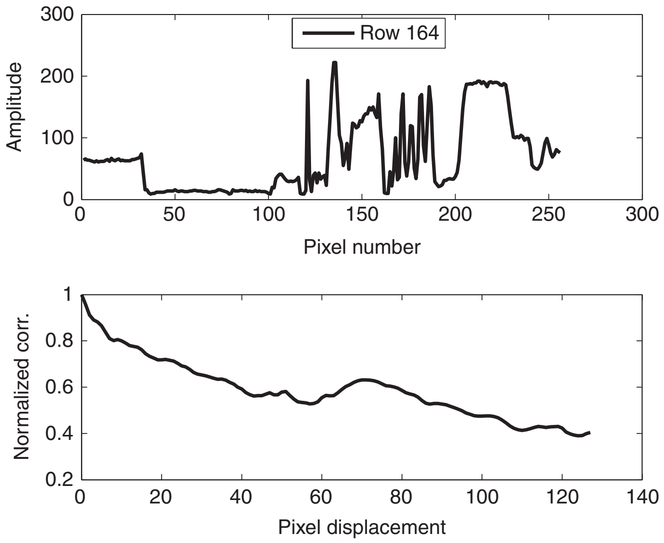


Figure 1.4 Profile of cameraman image along row number 164. The top graph shows pixel intensity, and the bottom graph shows corresponding normalized correlation over 128 pixel displacements.

and the corresponding correlation (bottom figure) of the cameraman picture along row 164. The MATLAB M-file for generating Figure 1.4 is listed below. Observe that the pixel values are very nearly the same over a large number of neighboring pixels and so is the pixel correlation. In other words, pixels in a row have a high correlation. Similarly, pixels may also have a high correlation along the columns. Thus, pixel redundancies translate to pixel correlation. The basic principle behind image data compression is to *decorrelate* the pixels and encode the resulting decorrelated image for transmission or storage. A specific compression scheme will depend on the method by which the pixel correlations are removed.

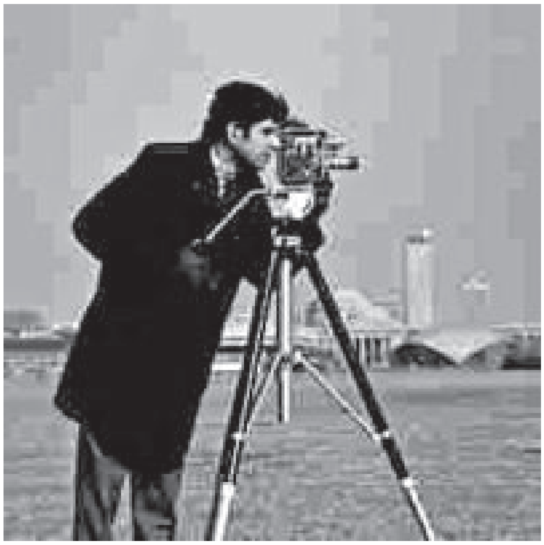
```
Figure1_4.m
%   Plots the image intensity profile and pixel correlation
%   along a specified row.
clear
close all
I = imread('cameraman.tif');
figure,imshow(I),title('Input image')
%
Row = 164; % row number of image profile
x = double(I(Row,:));
Col = size(I,2);
%
MaxN = 128; % number of correlation points to calculate
Cor = zeros(1,MaxN); % array to store correlation values
for k = 1:MaxN
    l = length(k:Col);
    Cor(k) = sum(x(k:Col) .* x(1:Col-k+1))/l;
end
MaxCor = max(Cor);
Cor = Cor/MaxCor;
figure,subplot(2,1,1),plot(1:Col,x,'k','LineWidth',2)
xlabel('Pixel number'), ylabel('Amplitude')
legend(['Row' ' ' num2str(Row)],0)
subplot(2,1,2),plot(0:MaxN-1,Cor,'k','LineWidth',2)
xlabel('Pixel displacement'), ylabel('Normalized corr.')
```

One of the earliest and basic image compression techniques is known as the *differential pulse code modulation* (DPCM) [1]. If the pixel correlation along only one dimension (row or column) is removed, then the DPCM is called one-dimensional (1D) DPCM or row-by-row DPCM. If the correlations along both dimensions are removed, then the resulting DPCM is known as 2D DPCM. A DPCM removes pixel correlation and requantizes the residual pixel values for storage or transmission. The residual image has a variance much smaller than that of the original image.

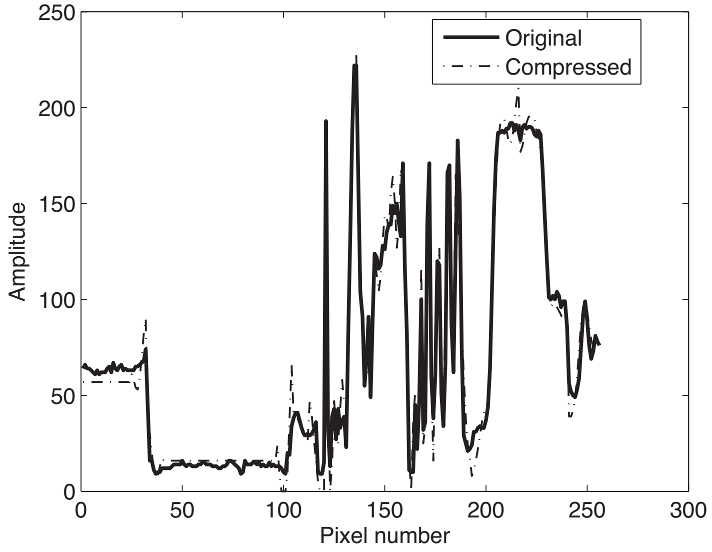
Further, the residual image has a probability density function, which is a double-sided exponential function. These give rise to compression.

The quantizer is fixed no matter how the decorrelated pixel values are. A variation on the theme is to use quantizers that adapt to changing input statistics, and therefore, the corresponding DPCM is called an adaptive DPCM. DPCM is very simple to implement, but the compression achievable is about 4:1. Due to limited bit width of the quantizer for the residual image, edges are not preserved well in the DPCM. It also exhibits occasional streaks across the image when channel error occurs. We will discuss DPCM in detail in a later chapter.

Another popular and more efficient compression scheme is known by the generic name *transform coding*. Remember that the idea is to reduce or remove pixel correlation to achieve compression. In transform coding, a block of image pixels is linearly transformed into another block of *transform coefficients* of the same size as the pixel block with the hope that only a few of the transform coefficients will be significant and the rest may be discarded. This implies that storage space is required to store only the significant transform coefficients, which are a fraction of the total number of coefficients and hence the compression. The original image can be reconstructed by performing the inverse transform of the reduced coefficient block. It must be pointed out that the inverse transform must exist for unique reconstruction. There are a number of such transforms available in the field to choose from, each having its own merits and demerits. The most efficient transform is one that uses the least number of transform coefficients to reconstruct the image for a given amount of distortion. Such a linear transform is known as the optimal transform where optimality is with respect to the minimum mean square error between the original and reconstructed images. This optimal image transform is known by the names *Karhunen–Loève transform* (KLT) or *Hotelling transform*. The disadvantage of the KLT is that the transform kernel depends on the actual image to be compressed, which requires a lot more side information for the receiver to reconstruct the original image from the compressed image than other *fixed* transforms. A highly popular fixed transform is the familiar *discrete cosine transform* (DCT). The DCT has very nearly the same *compression efficiency* as the KLT with the advantage that its *kernel* is fixed and so no side information is required by the receiver for the reconstruction. The DCT is used in the JPEG and MPEG video compression standards. The DCT is usually applied on nonoverlapping blocks of an image. Typical DCT blocks are of size 8×8 or 16×16. One of the disadvantages of image compression using the DCT is the *blocking* artifact. Because the DCT blocks are small compared with the image and because the average values of the blocks may be different, blocking artifacts appear when the zero-frequency (dc) DCT coefficients are quantized rather heavily. However, at low compression, blocking artifacts are almost unnoticeable. An example showing blocking artifacts due to compression using 8×8 DCT is shown in Figure 1.5a. Blockiness is clearly seen in flat areas—both low and high intensities as well as undershoot and overshoot along the sharp edges—see Figure 1.5b. A listing of M-file for Figures 1.5a,b is shown below.



(a)



(b)

Figure 1.5 (a) Cameraman image showing blocking artifacts due to quantization of the DCT coefficients. The DCT size is 8 × 8. (b) Intensity profile along row number 164 of the image in (a).

```
% Figure1_5.m
% Example to show blockiness in DCT compression
% Quantizes and dequantizes an intensity image using
% 8x8 DCT and JPEG quantization matrix
close all
clear
I = imread('cameraman.tif');
figure,imshow(I), title('Original Image')
%
fun = @dct2; % 2D DCT function
N = 8; % block size of 2D DCT
T = blkproc(I,[N N],fun);  % compute 2D DCT of image using NxN blocks
%
Scale = 4.0; % increasing Scale quntizes DCT coefficients heavily
% JPEG default quantization matrix
jpgQMat = [16 11 10 16  24  40  51  61;
           12 12 14 19  26  58  60  55;
           14 13 16 24  40  57  69  56;
           14 17 22 29  51  87  80  62;
           18 22 37 56  68 109 103  77;
           24 35 55 64  81 194 113  92;
           49 64 78 87 103 121 120 101;
           72 92 95 98 121 100 103  99];
Qstep = jpgQMat * Scale; % quantization step size
%   Quantize and dequantize the coefficients
for k = 1:N:size(I,1)
    for l = 1:N:size(I,2)
        T1(k:k+N-1,l:l+N-1) = round(T(k:k+N-1,l:l+N-1)./ Qstep).*Qstep;
    end
end
% do inverse 2D DCT
fun = @idct2;
y = blkproc(T1,[N N],fun);
y = uint8(round(y));
figure,imshow(y), title('DCT compressed Image')
% Plot image profiles before and after compression
ProfRow = 164;
figure,plot(1:size(I,2),I(ProfRow,:),'k','LineWidth',2)
hold on
plot(1:size(I,2),y(ProfRow,:),'-.k','LineWidth',1)
title(['Intensity profile of row ' num2str(ProfRow)])
xlabel('Pixel number'), ylabel('Amplitude')
%legend(['Row' ' ' num2str(ProfRow)],0)
legend('Original','Compressed',0)
```

A third and relatively recent compression method is based on *wavelet transform*. As we will see in a later chapter, wavelet transform captures both long-term and short-term changes in an image and offers a highly efficient compression mechanism. As a result, it is used in the latest versions of the JPEG standards as a compression tool. It is also adopted by the SMPTE (Society of Motion Pictures and Television Engineers). Even though the wavelet transform may be applied on blocks of an image like the DCT, it is generally applied on the full image and the various wavelet

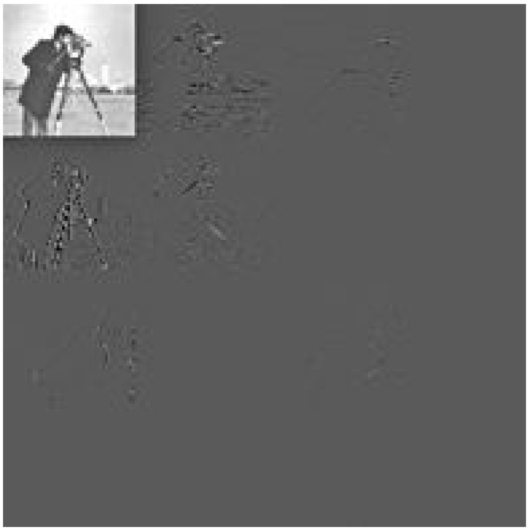
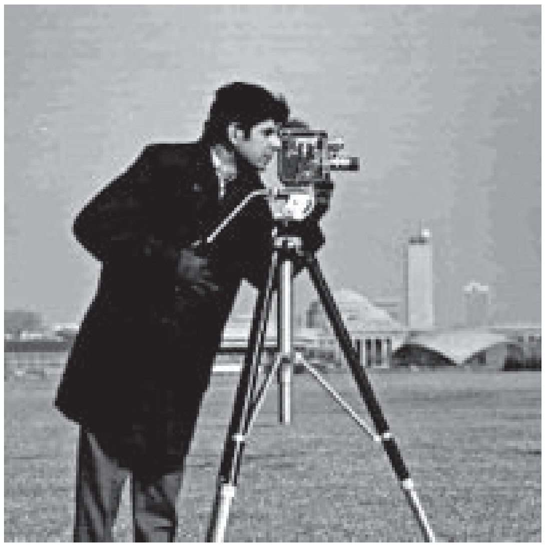
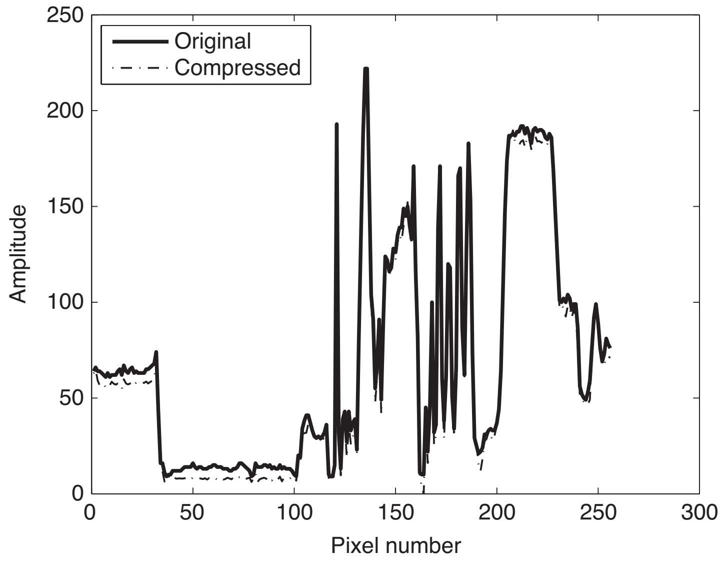


Figure 1.6 A two-level 2D DWT of cameraman image.

coefficients are quantized according to their types. A two-level discrete wavelet transform (DWT) of the cameraman image is shown in Figure 1.6 to illustrate how the 2D wavelet transform coefficients look like. Details pertaining to the levels and subbands of the DWT will be given in a later chapter. The M-file to implement multilevel 2D DWT that generates Figure 1.6 is listed below. As we will see in a later chapter, the 2D DWT decomposes an image into one approximation and many detail coefficients. The number of coefficient subimages corresponding to an L-level 2D DWT equals $3 \times L + 1$. Therefore, for a two-level 2D DWT, there are seven coefficient subimages. In the first level, there are three detail coefficient subimages, each of size $\frac{1}{4}$ the original image. The second level consists of four sets of DWT coefficients—one approximation and three details, each $\frac{1}{16}$ the original image. As the name implies the approximation coefficients are lower spatial resolution approximations to the original image. The detail coefficients capture the discontinuities or edges in the image with orientations in the horizontal, vertical, and diagonal directions. In order to compress, an image using 2D DWT we have to compute the 2D DWT of the image up to a given level and then quantize each coefficient subimage. The achievable quality and compression ratio depend on the chosen wavelets and quantization method. The visual effect of quantization distortion in DWT compression scheme is different from that in DCT-based scheme. Figure 1.7a is the cameraman image compressed using 2D DWT. The wavelet used is called Daubechies 2 (db2 in MATLAB) and the number of levels used is 1. We note that there are no blocking effects, but there are patches in the flat areas. We also see that the edges are reproduced faithfully as evidenced in the profile (Figure 1.7b). It must be pointed out that the amount of quantization applied in Figure 1.7a is not the same as that used for the DCT example and that the two examples are given only to show the differences in the artifacts introduced by the two schemes. An M-file listing to generate Figures 1.7a,b is shown below.



(a)



(b)

Figure 1.7 (a) Cameraman image compressed using one-level 2D DWT. (b) Intensity profile of image in Figure 1.8a along row number 164.

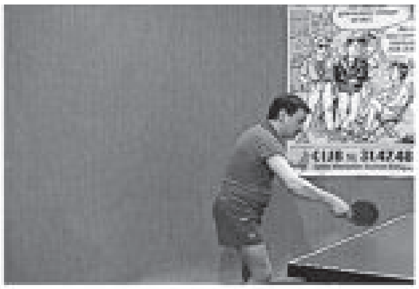
```
% Figure1_6.m
% 2D Discrete Wavelet Transform (DWT)
% Computes multi-level 2D DWT of an intensity image
close all
clear
I = imread('cameraman.tif');
figure,imshow(I), title('Original Image')
L = 2; % number of levels in DWT
[W,B] = wavedec2(I,L,'db2'); % do a 2-level DWT using db2 wavelet
% declare level-1 subimages
w11 = zeros(B(3,1),B(3,1));
w12 = zeros(B(3,1),B(3,1));
w13 = zeros(B(3,1),B(3,1));
% declare level-2 subimages
w21 = zeros(B(1,1),B(1,1));
w22 = zeros(B(1,1),B(1,1));
w23 = zeros(B(1,1),B(1,1));
w24 = zeros(B(1,1),B(1,1));
% extract level-1 2D DWT coefficients
offSet11 = 4*B(1,1)*B(1,2);
offSet12 = 4*B(1,1)*B(1,2)+B(3,1)*B(3,2);
offSet13 = 4*B(1,1)*B(1,2)+2*B(3,1)*B(3,2);
for c = 1:B(2,2)
    for r = 1:B(2,1)
        w11(r,c) = W(offSet11+(c-1)*B(3,1)+r);
        w12(r,c) = W(offSet12+(c-1)*B(3,1)+r);
        w13(r,c) = W(offSet13+(c-1)*B(3,1)+r);
    end
end
% extract level-2 2D DWT coefficients
offSet22 = B(1,1)*B(1,2);
offSet23 = 2*B(1,1)*B(1,2);
offSet24 = 3*B(1,1)*B(1,2);
for c = 1:B(1,2)
    for r = 1:B(1,1)
        w21(r,c) = W((c-1)*B(1,1)+r);
        w22(r,c) = W(offSet22+(c-1)*B(1,1)+r);
        w23(r,c) = W(offSet23+(c-1)*B(1,1)+r);
        w24(r,c) = W(offSet24+(c-1)*B(1,1)+r);
    end
end
% declare output array y to store all the DWT coefficients
%y = zeros(261,261);
y = zeros(2*B(1,1)+B(3,1),2*B(1,2)+B(3,2));
y(1:B(1,1),1:B(1,1))=w21;
y(1:B(1,1),B(1,1)+1:2*B(1,1))=w22;
y(B(1,1)+1:2*B(1,1),1:B(1,1))=w23;
y(B(1,1)+1:2*B(1,1),B(1,1)+1:2*B(1,1))=w24;
%
y(1:B(3,1),2*B(1,1)+1:261)=w11;
y(2*B(1,1)+1:261,1:129)=w12;
y(2*B(1,1)+1:261,2*B(1,1)+1:261)=w13;
figure,imshow(y,[]),title([num2str(L) '-level 2D DWT'])
```

```
% Figure1_7.m
% An example to show the effect of quantizing the 2D DWT
% coefficients of an intensity image along with intensity
% profile along a specified row
```

```
%
close all
clear
I = imread('cameraman.tif');
figure,imshow(I), title('Original Image')
% do a 1-level 2D DWT
[W,B] = wavedec2(I,1,'db2');
w11 = zeros(B(1,1),B(1,2));
w12 = zeros(B(1,1),B(1,2));
w13 = zeros(B(1,1),B(1,2));
w14 = zeros(B(1,1),B(1,2));
%
offSet12 = B(1,1)*B(1,2);
offSet13 = 2*B(1,1)*B(1,2);
offSet14 = 3*B(1,1)*B(1,2);
% quantize only the approximation coefficients
Qstep = 16;
for c = 1:B(1,2)
    for r = 1:B(1,1)
        W((c-1)*B(1,1)+r) = floor(W((c-1)*B(1,1)+r)/Qstep)*Qstep;
        %{
        W(offSet12+(c-1)*B(1,1)+r) = floor(W(offSet12+(c-1)*B(1,1)+r)/8)*8;
        W(offSet13+(c-1)*B(1,1)+r) = floor(W(offSet13+(c-1)*B(1,1)+r)/8)*8;
        W(offSet14+(c-1)*B(1,1)+r) = floor(W(offSet14+(c-1)*B(1,1)+r)/8)*8;
        %}
    end
end
% do inverse 2D DWT
y = waverec2(W,B,'db2');
figure,imshow(y,[])
% plot profile
ProfRow = 164;
figure,plot(1:size(I,2),I(ProfRow,:),'k','LineWidth',2)
hold on
plot(1:size(I,2),y(ProfRow,:),'-.k','LineWidth',1)
title(['Profile of row ' num2str(ProfRow)])
xlabel('Pixel number'), ylabel('Amplitude')
%legend(['Row' ' ' num2str(ProfRow)],0)
legend('Original','Compressed',0)
```

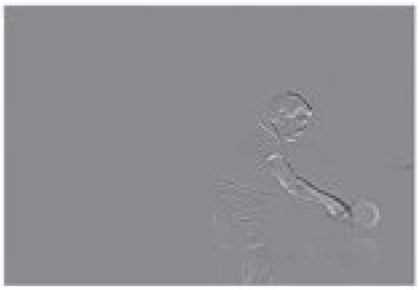
1.3.2 Video Compression

So far our discussion on compression has been on still images. These techniques try to exploit the *spatial correlation* that exists in a still image. When we want to compress video or sequence images we have an added dimension to exploit, namely, the temporal dimension. Generally, there is little or very little change in the spatial arrangement of objects between two or more consecutive *frames* in a video. Therefore, it is advantageous to send or store the differences between consecutive frames rather than sending or storing each frame. The difference frame is called the *residual* or *differential* frame and may contain far less details than the actual frame itself. Due to this reduction in the details in the differential frames, compression is achieved. To illustrate the idea, let us consider compressing two consecutive frames (frame 120 and frame 121) of a video sequence as shown in Figures 1.8a,b, respectively (see M-file listing shown below). The difference between frames 121 and 120 is shown

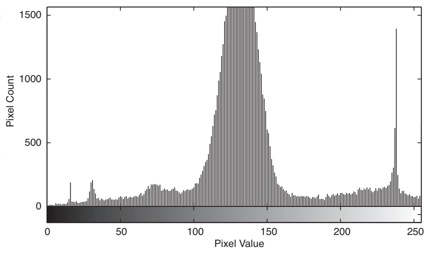


(a)

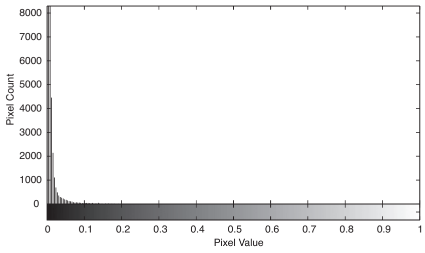
(b)



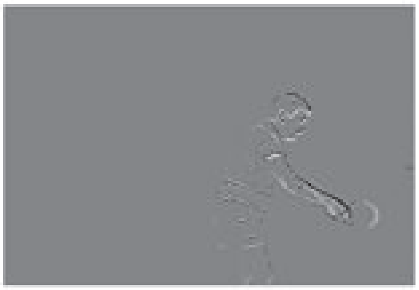
(c)



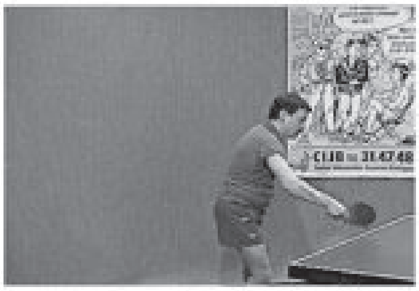
(d)



(e)



(f)



(g)

Figure 1.8 Compressing a video sequence: (a) frame 120 of a table tennis video sequence; (b) frame 121 of the video sequence; (c) difference between frames 121 and 120; (d) histogram of frame 121; (e) histogram of the difference of frames; (g) quantized difference frame; and (h) Reconstructed frame 121 by adding the quantized difference frame to frame 120.

in Figure 1.8c. The differential frame has a small amount of details corresponding to the movements of the hand and the racket. Note that stationary objects do not appear in the difference frame. This is evident from the histogram of the differential frame shown in Figure 1.8e, where the intensity range occupied by the differential pixels is much smaller. Compare this with the histogram of frame 121 in Figure 1.8d which is much wider. The quantized differential frame and the reconstructed frame 121 are shown in Figures 1.8f,g, respectively. We see some distortions in the edges due to quantization.

When objects move between successive frames, simple differencing will introduce large residual values especially when the motion is large. Due to relative motion of objects, simple differencing is not efficient from the point of view of achievable compression. It is more advantageous to determine or estimate the relative motions of objects between successive frames and compensate for the motion and then do the differencing to achieve a much higher compression. This type of prediction is known as *motion compensated prediction*. Because we perform motion *estimation* and *compensation* at the encoder, we need to inform the decoder about this motion compensation. This is done by sending motion vectors as side information, which conveys the object motion in the horizontal and vertical directions. The decoder then uses the motion vectors to align the blocks and reconstruct the image.

```
% Figure1_8.m
% generates a differential frame by subtracting two
% temporally adjacent intensity image frames
% quantizes the differential frame and reconstructs
% original frame by adding quantized differential frame
% to the other frame.
close all
clear
Frm1 = 'tt120.ras';
Frm2 = 'tt121.ras';
I = imread(Frm1); % read frame # 120
I1 = im2single(I); % convert from uint8 to float single
I = imread(Frm2); % read frame # 121
figure,imhist(I,256),title(['Histogram of frame ' num2str(121)])
xlabel('Pixel Value'), ylabel('Pixel Count')
I2 = im2single(I); % convert from uint8 to float single
clear I
figure,imshow(I1,[]), title([num2str(120) 'th frame'])
figure,imshow(I2,[]), title([num2str(121) 'st frame'])
%
Idiff = imsubtract(I2,I1); % subtract frame 120 from 121
figure,imhist(Idiff,256),title('Histogram of difference image')
xlabel('Pixel Value'), ylabel('Pixel Count')
figure,imshow(Idiff,[]),title('Difference image')
% quantize and dequantize the differential image
IdiffQ = round(4*Idiff)/4;
figure,imshow(IdiffQ,[]),title('Quantized Difference image')
y = I1 + IdiffQ; % reconstruct frame 121
figure,imshow(y,[]),title('Reconstructed image')
```

A video sequence is generally divided into scenes with scene changes marking the boundaries between consecutive scenes. Frames within a scene are similar and there is a high temporal correlation between successive frames within a scene. We may, therefore, send differential frames within a scene to achieve high compression. However, when the scene changes, differencing may result in much more details than the actual frame due to the absence of correlation, and therefore, compression may not be possible. The first frame in a scene is referred to as the *key frame*, and it is compressed by any of the above-mentioned schemes such as the DCT or DWT. Other frames in the scene are compressed using temporal differencing. A detailed discussion on video compression follows in a later chapter.

1.3.3 Lossless Compression

The above-mentioned compression schemes are lossy because there is always a loss of some information when reconstructing the image from the compressed image. There is another category of compression methods wherein the image decoding or reconstruction is exact, that is, there is no loss of any information about the original image. Although this will be very exciting to a communications engineer, the achievable *compression ratio* is usually around 2:1, which may not always be sufficient from a storage or transmission point of view. However, there are situations where lossless image and video compression may be necessary. Digital mastering of movies is done by lossless compression. After editing a movie, it is compressed with loss for distribution. In telemedicine, medical images need to be compressed losslessly so as to enable a physician at a remote site to diagnose unambiguously. In general, a lossless compression scheme considers an image to consist of a finite *alphabet* of discrete *symbols* and relies on the probability of occurrence of these symbols to achieve lossless compression. For instance, if the image pixels have values between 0 and 255, then the alphabet consists of 256 symbols one for each integer value with a characteristic probability distribution that depends on the source. We can then generate binary *codeword* for each symbol in the alphabet, wherein the code length of a symbol increases with its decreasing probability in a logarithmic fashion. This is called a *variable-length coding*. Huffman coding [2] is a familiar example of variable-length coding. It is also called *entropy coding* because the average code length of a large sequence approaches the entropy of the source. As an example, consider a discrete source with an alphabet $A = \{a_1, a_2, a_3, a_4\}$ with respective probabilities of $\frac{1}{8}$, $\frac{1}{2}$, $\frac{1}{8}$, and $\frac{1}{4}$. Then one possible set of codes for the symbols is shown in the table below. These are variable-length codes, and we see that no code is a prefix to other codes. Hence, these codes are also known as prefix codes. Observe that the most likely symbol a_2 has the least code length and the least probable symbols a_1 and a_3 have the largest code length. We also find that the average number of bits per symbol is 1.75, which happens to be the entropy of this source. We will discuss source entropy in detail in Chapter 5. One drawback of Huffman coding is that the entire codebook must be available at the decoder. Depending on the number of codewords, the amount of side information about the codebook to be transmitted may be very large and the coding efficiency may be reduced. If the number

Table 1.1 Variable-length codes

Symbol	Probability	Code
a_1	1/8	110
a_2	1/2	0
a_3	1/8	111
a_4	1/4	10

of symbols is small, then we can use a more efficient lossless coding scheme called *arithmetic coding*. Arithmetic coding does not require the transmission of codebook and so achieves a higher compression than Huffman coding would. For compressing textual information, there is an efficient scheme known as *Lempel–Ziv* (LZ) *coding* [3] method. As we are concerned only with image and video compression here, we will not discuss LZ method further.

With this short description of the various compression methods for still image and video, we can now look at the plethora of compression schemes in a tree diagram as illustrated in Figure 1.9. It should be pointed out that lossless compression is always included as part of a lossy compression even though it is not explicitly shown in the figure. It is used to losslessly encode the various quantized pixels or transform coefficients that take place in the compression chain.

1.4 VIDEO COMPRESSION STANDARDS

Interoperability is crucial when different platforms and devices are involved in the delivery of images and video data. If for instance images and video are compressed using a proprietary algorithm, then decompression at the user end is not feasible unless the same proprietary algorithm is used, thereby encouraging monopolization.

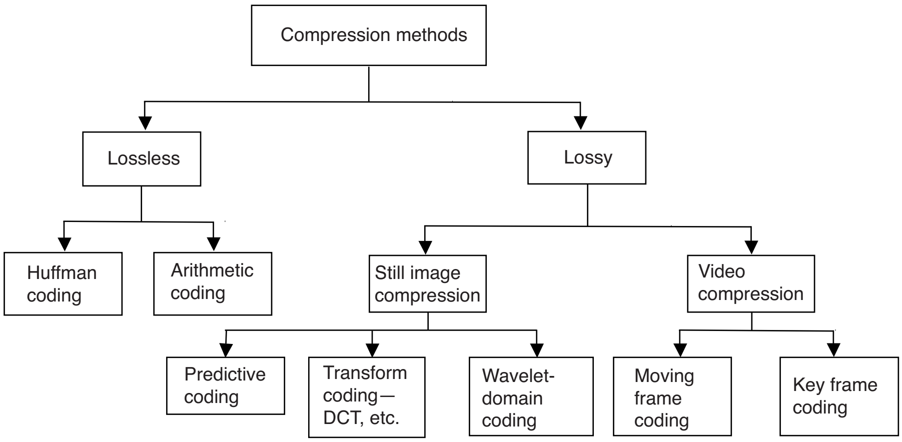


Figure 1.9 A taxonomy of image and video compression methods.

This, therefore, calls for a standardization of the compression algorithms as well as data transportation mechanism and protocols so as to guarantee not only interoperability but also competitiveness. This will eventually open up growth potential for the technology and will benefit the consumers as the prices will go down. This has motivated people to form organizations across nations to develop solutions to interoperability.

The first successful standard for still image compression known as JPEG was developed jointly by the International Organization for Standardization (ISO) and International Telegraph and Telephone Consultative Committee (CCITT) in a collaborative effort. CCITT is now known as International Telecommunication Union—Telecommunication (ITU-T). JPEG standard uses DCT as the compression tool for grayscale and true color still image compression. In 2000, JPEG [4] adopted 2D DWT as the compression vehicle.

For video coding and distribution, MPEG was developed under the auspicious of ISO and International Electrotechnical Commission (IEC) groups. MPEG [5] denotes a family of standards used to compress audio-visual information. Since its inception MPEG standard has been extended to several versions. MPEG-1 was meant for video compression at about 1.5 Mb/s rate suitable for CD ROM. MPEG-2 aims for higher data rates of 10 Mb/s or more and is intended for SD and HD TV applications. MPEG-4 is intended for very low data rates of 64 kb/s or less. MPEG-7 is more on standardization of description of multimedia information rather than compression. It is intended for enabling efficient search of multimedia contents and is aptly called *multimedia content description interface*. MPEG-21 aims at enabling the use of multimedia sources across many different networks and devices used by different communities in a transparent manner. This is to be accomplished by defining the entire multimedia framework as *digital items*. Details about various MPEG standards will be given in Chapter 10.

1.5 ORGANIZATION OF THE BOOK

We begin with image acquisition techniques in Chapter 2. It describes image sampling and quantization schemes followed by various color coordinates used in the representation of color images and various video formats. Unitary transforms, especially the DCT, are important compression vehicles, and so in Chapter 3, we will define the unitary transforms and discuss their properties. We will then describe image transforms such as KLT and DCT and illustrate their merits and demerits by way of examples. In Chapter 4, 2D DWT will be defined along with methods of its computation as it finds extensive use in image and video compression. Chapter 5 starts with a brief description of information theory and source entropy and then describes lossless coding methods such as Huffman coding and arithmetic coding with some examples. It also shows examples of constructing Huffman codes for a specific image source. We will then develop the idea of predictive coding and give detailed descriptions of DPCM in Chapter 6 followed by transform domain coding procedures for still image compression in Chapter 7. Chapter 7 also describes JPEG standard for still image compression. Chapter 8 deals with image compression in the wavelet domain as well as JPEG2000 standard. Video coding principles will be discussed in Chapter 9. Various motion estimation techniques will be described in Chapter 9 with several examples. Compression standards such as MPEG will be discussed in Chapter 10 with examples.

1.6 SUMMARY

Still image and video sources require wide bandwidths for real-time transmission or large storage memory space. Therefore, some form of data compression must be applied to the visual data before transmission or storage. In this chapter, we have introduced terminologies of lossy and lossless methods of compressing still images and video. The existing lossy compression schemes are DPCM, transform coding, and wavelet-based coding. Although DPCM is very simple to implement, it does not yield high compression that is required for most image sources. It also suffers from distortions that are objectionable in applications such as HDTV. DCT is the most popular form of transform coding as it achieves high compression at good visual quality and is, therefore, used as the compression vehicle in JPEG and MPEG standards. More recently 2D DWT has gained importance in video compression because of its ability to achieve high compression with good quality and because of the availability of a wide variety of wavelets. The examples given in this chapter show how each one of these techniques introduces artifacts at high compression ratios.

In order to reconstruct images from compressed data without incurring any loss whatsoever, we mentioned two techniques, namely, Huffman and arithmetic coding. Even though lossless coding achieves only about 2:1 compression, it is necessary where no loss is tolerable, as in medical image compression. It is also used in all lossy compression systems to represent quantized pixel values or coefficient values for storage or transmission and also to gain additional compression.

REFERENCES

1. J. B. O'Neal, Jr., "Differential pulse code modulation (DPCM) with entropy coding," *IEEE Trans. Inf. Theory,* IT-21 (2), 169–174, 1976.
2. D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, 40, 1098–1101, 1951.
3. J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, IT-24 (5), 530–536, 1978.
4. JPEG, *Part I: Final Draft International Standard* (ISO/IEC FDIS15444–1), ISO/IEC JTC1/SC29/WG1N1855, 2000.
5. MPEG, URL for MPEG organization is http://www.mpeg.org/.

2

IMAGE ACQUISITION

2.1 INTRODUCTION

Digital images are acquired through cameras using photo sensor arrays. The sensors are made from semiconductors and may be of *charge-coupled devices* or *complementary metal oxide semiconductor* devices. The photo detector elements in the arrays are built with a certain size that determines the image resolution achievable with that particular camera. To capture color images, a digital camera must have either a prism assembly or a color filter array (CFA). The prism assembly splits the incoming light three-ways, and optical filters are used to separate split light into red, green, and blue spectral components. Each color component excites a photo sensor array to capture the corresponding image. All three-component images are of the same size. The three photo sensor arrays have to be aligned perfectly so that the three images are registered. Cameras with prism assembly are a bit bulky and are used typically in scientific and or high-end applications. Consumer cameras use a single chip and a CFA to capture color images without using a prism assembly. The most commonly used CFA is the Bayer filter array. The CFA is overlaid on the sensor array during chip fabrication and uses alternating color filters, one filter per pixel. This arrangement produces three-component images with a full spatial resolution for the green component and half resolution for each of the red and blue components. The advantage, of course, is the small and compact size of the camera. The disadvantage is that the resulting image has reduced spatial and color resolutions.

Whether a camera is three chip or single chip, one should be able to determine analytically how the spatial resolution is related to the image pixel size. One should

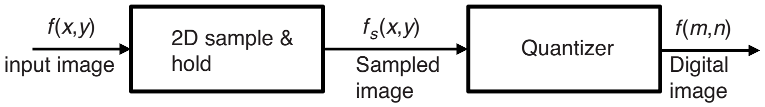


Figure 2.1 Sampling and quantizing a continuous image.

also be able to determine the distortions in the reproduced or displayed continuous image as a result of the spatial sampling and quantization used. In the following section, we will develop the necessary mathematical equations to describe the processes of sampling a continuous image and reconstructing it from its samples.

2.2 SAMPLING A CONTINUOUS IMAGE

An image $f(x,y)$, $-\infty \le x, y \le \infty$ to be sensed by a photo sensor array is a continuous distribution of light intensity in the two continuous spatial coordinates x and y. Even though practical images are of finite size, we assume here that the extent of the image is infinite to be more general. In order to acquire a digital image from $f(x, y)$, (a) it must be discretized or sampled in the spatial coordinates and (b) the sample values must be quantized and represented in digital form, see Figure 2.1. The process of discretizing $f(x, y)$ in the two spatial coordinates is called *sampling* and the process of representing the sample values in digital form is known as *analog-to-digital conversion*. Let us first study the sampling process.

The sampled image will be denoted by $f_S(x, y)$, $-\infty \le x, y \le \infty$ and can be expressed as

$$f_S(x, y) = f(x, y)_{|x=m\Delta x, y=n\Delta y} \tag{2.1}$$

where $-\infty \le m,\ n \le \infty$ are integers and Δx and Δy are the spacing between samples in the two spatial dimensions, respectively. Since the spacing is constant the resulting sampling process is known as *uniform sampling*. Further, equation (2.1) represents sampling the image with an impulse array and is called ideal or impulse sampling. In an ideal image sampling, the sample width approaches zero. The sampling indicated by equation (2.1) can be interpreted as multiplying $f(x, y)$ by a sampling function $s(x, y)$, which, for ideal sampling, is an array of impulses spaced uniformly at integer multiples of Δx and Δy in the two spatial coordinates, respectively. Figure 2.2 shows an array of impulses. An impulse or Dirac delta function in the two-dimensional (2D) spatial domain is defined by

$$\delta(x, y) = 0, \quad \text{for}\, x, y \ne 0 \tag{2.2a}$$

together with

$$\underset{\lim \varepsilon \to 0}{\int_{-\varepsilon}^{\varepsilon}} \int_{-\varepsilon}^{\varepsilon} \delta(x, y)\, dx\, dy = 1 \tag{2.2b}$$

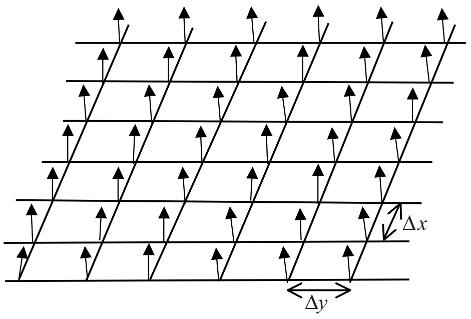


Figure 2.2 A 2D array of Dirac delta functions.

Equation (2.2) implies that the Dirac delta function has unit area, while the width approaches zero. Note that the ideal sampling function can be expressed as

$$s(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \delta(x - m\Delta x, y - n\Delta y) \tag{2.3}$$

Thus, the sampled image $f_S(x, y)$ can be written as

$$f_s(x, y) = f(x, y) \times s(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m\Delta x, n\Delta y)\, \delta(x - m\Delta x, y - n\Delta y) \tag{2.4}$$

In order for us to recover the original continuous image $f(x, y)$ from the sampled image $f_S(x, y)$, we need to determine the maximum spacing Δx and Δy. This is done easily if we use the Fourier transform. So, let $F\left(\omega_x, \omega_y\right)$ be the 2D continuous Fourier transform of $f(x, y)$ as defined by

$$F\left(\omega_x, \omega_y\right) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)\, e^{-j\left(\omega_x x + \omega_y y\right)} dx dy \tag{2.5}$$

The Fourier transform of the 2D sampling function in equation (2.3) can be shown to be another 2D impulse array and is given by

$$\text{FT}\{s(x, y)\} = \frac{1}{\Delta x \Delta y} \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \delta\left(\omega_x - m\omega_{xS}, \omega_y - n\omega_{yS}\right) \tag{2.6}$$

where $\omega_{xS} = 2\pi/\Delta x$ and $\omega_{yS} = 2\pi/\Delta y$ are the sampling frequencies in radians per unit distance in the respective spatial dimensions. Because the sampled image is the product of the continuous image and the sampling function as in equation (2.4), the

2D Fourier transform of the sampled image is the convolution of the corresponding Fourier transforms. Therefore, we have the Fourier transform $F_S\left(\omega_x,\omega_y\right)$ of $f_S\left(x,y\right)$ expressed as

$$F_S\left(\omega_x,\omega_y\right) = F\left(\omega_x,\omega_y\right) \otimes \left\{ \frac{1}{\Delta x \Delta y} \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \delta\left(\omega_x - m\omega_{xS}, \omega_y - n\omega_{yS}\right) \right\} \tag{2.7}$$

In equation (2.7), $\otimes$ represents the 2D convolution. Equation (2.7) can be simplified and written as

$$F_S\left(\omega_x,\omega_y\right) = \frac{1}{\Delta x \Delta y} \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} F\left(\omega_x - m\omega_{xS}, \omega_y - n\omega_{yS}\right) \tag{2.8}$$

From equation (2.8), we see that the Fourier transform of the ideally sampled image is obtained by (a) replicating the Fourier transform of the continuous image at multiples of the sampling frequencies in the two dimensions, (b) scaling the replicas by $1/(\Delta x \Delta y)$, and (c) adding the resulting functions. Since the Fourier transform of the sampled image is continuous, it is possible to recover the original continuous image from the sampled image by filtering the sampled image by a suitable linear filter. If the continuous image is lowpass, then exact recovery is feasible by an ideal lowpass filter, provided the continuous image is band limited to say $\omega_{xS}/2$ and $\omega_{yS}/2$ in the two spatial frequencies, respectively. That is to say, if

$$F\left(\omega_x,\omega_y\right) = \begin{cases} \text{nonzero}, & \text{for } |\omega_x| \le \dfrac{\omega_{xS}}{2}, \left|\omega_y\right| \le \dfrac{\omega_{yS}}{2} \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

then $f\left(x, y\right)$ can be recovered from the sampled image by filtering it by an ideal lowpass filter with cutoff frequencies $\omega_{xC} = \omega_{xS}/2$ and $\omega_{yC} = \omega_{yS}/2$ in the two frequency axes, respectively. This is possible because the Fourier transform of the sampled image will be nonoverlapping and will be identical to that of the continuous image in the region specified in equation (2.9). To see this, let the ideal lowpass filter $H\left(\omega_x,\omega_y\right)$ be specified by

$$H\left(\omega_{\mathrm{x}},\omega_y\right) = \begin{cases} \Delta x \Delta y, & |\omega_x| \le \omega_{xC}, \left|\omega_y\right| \le \omega_{yC} \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

Then the response $\tilde{F}\left(\omega_x,\omega_y\right)$ of the ideal lowpass filter to the sampled image has the Fourier transform, which is the product of $H\left(\omega_x,\omega_y\right)$ and $F_S\left(\omega_x,\omega_y\right)$ and is given by

$$\tilde{F}\left(\omega_x,\omega_y\right) = F_S\left(\omega_x,\omega_y\right) H\left(\omega_x,\omega_y\right) = F\left(\omega_x,\omega_y\right) \tag{2.11}$$

Equation (2.11) indicates that the reconstructed image is a replica of the original continuous image. We can state formally the sampling process with constraints on the sample spacing in the following theorem.

Sampling Theorem for Lowpass Image

A lowpass continuous image $f(x, y)$ having maximum frequencies as given in equation (2.9) can be recovered exactly from its samples $f(m\Delta x, n\Delta y)$ spaced uniformly in a rectangular grid with spacing Δx and Δy provided the sampling rates satisfy

$$\frac{1}{\Delta x} = F_{xS} \geq 2F_{xC}, \quad \text{equivalently } \frac{2\pi}{\Delta x} = \omega_{xS} \geq 2\omega_{xC} \tag{2.12a}$$

$$\frac{1}{\Delta y} = F_{yS} \geq 2F_{yC}, \quad \text{equivalently } \frac{2\pi}{\Delta y} = \omega_{yS} \geq 2\omega_{yC} \tag{2.12b}$$

The sampling frequencies F_{xS} and F_{yS} in cycles per unit distance equal to twice the respective maximum image frequencies F_{xC} and F_{yC} are called the Nyquist frequencies. A more intuitive interpretation of the Nyquist frequencies is that the spacing between samples Δx and Δy must be at most half the size of the finest details to be resolved. From equation (2.11), the reconstructed image is obtained by taking the inverse Fourier transform of $\tilde{F}(\omega_x, \omega_y)$. Equivalently, the reconstructed image is obtained by convolving the sampled image by the reconstruction filter impulse response and can be written formally as

$$\tilde{f}(x, y) = f_S(x, y) \otimes h(x, y) \tag{2.13}$$

The impulse response of the lowpass filter having the transfer function $H(\omega_x, \omega_y)$ can be evaluated from

$$h(x, y) = \frac{1}{4\pi^2} \int_{-\omega_{xS}/2}^{\omega_{xS}/2} \int_{\omega_{yS}/2}^{\omega_{yS}/2} H(\omega_x, \omega_y) e^{-j(\omega_x x + \omega_y y)} d\omega_x d\omega_y \tag{2.14}$$

Substituting for $H(\omega_x, \omega_y)$ from equation (2.10) in (2.14) and carrying out the integration, we find that

$$h(x, y) = \left\{ \frac{\sin(\pi x F_{xS})}{\pi x F_{xS}} \right\} \left\{ \frac{\sin(\pi y F_{yS})}{\pi y F_{yS}} \right\} \tag{2.15}$$

Using equations (2.4) and (2.15) in (2.13), we have

$$\tilde{f}(x, y) = \left\{ \sum_{-\infty}^{\infty} \sum_{-\infty}^{\infty} f(m\Delta x, n\Delta y) \delta(x - m\Delta x, y - n\Delta y) \right\} \otimes h(x, y) \tag{2.16}$$

Replacing the convolution symbol, equation (2.16) is rewritten as

$$\tilde{f}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(u,v) \times \left\{ \sum_{m=-\infty}^{\infty} \sum_{n=-\infty} f(m\Delta x, n\Delta y)\,\delta(x-u-m\Delta x, y-v-n\Delta y) \right\} du dv \tag{2.17}$$

Interchanging the order of integration and summation, equation (2.17) can be written as

$$\tilde{f}(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m\Delta x, n\Delta y) \times \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(u,v)\,\delta(x-u-m\Delta x, y-v-n\Delta y)\, du dv \tag{2.18}$$

which reduces to

$$\tilde{f}(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m\Delta x, n\Delta y)\, h(x-m\Delta x, y-n\Delta y) \tag{2.19}$$

Substituting for $h(x,y)$ from equation (2.15), the reconstructed image in terms of the sampled image is found to be

$$\tilde{f}(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m\Delta x, n\Delta y) \left\{ \frac{\sin(\pi(xF_{xS}-m))}{\pi(xF_{xS}-m)} \right\} \left\{ \frac{\sin\left(\pi\left(yF_{yS}-n\right)\right)}{\pi\left(yF_{yS}-n\right)} \right\} \tag{2.20}$$

Since the *sinc* functions in equation (2.20) attain unit values at multiples of Δx and Δy, the reconstructed image exactly equals the input image at the sampling locations and at other locations the ideal lowpass filter interpolates to obtain the original image. Thus, the continuous image is recovered from the sampled image exactly by linear filtering when the sampling is ideal and the sampling rates satisfy the Nyquist criterion as given in equation (2.12). What happens to the reconstructed image if the sampling rates are less than the Nyquist frequencies?

2.2.1 Aliasing Distortion

Sampling theorem guarantees exact recovery of a continuous image, which is low-pass and band limited to frequencies $F_{xS}/2$ and $F_{yS}/2$, from its samples taken at

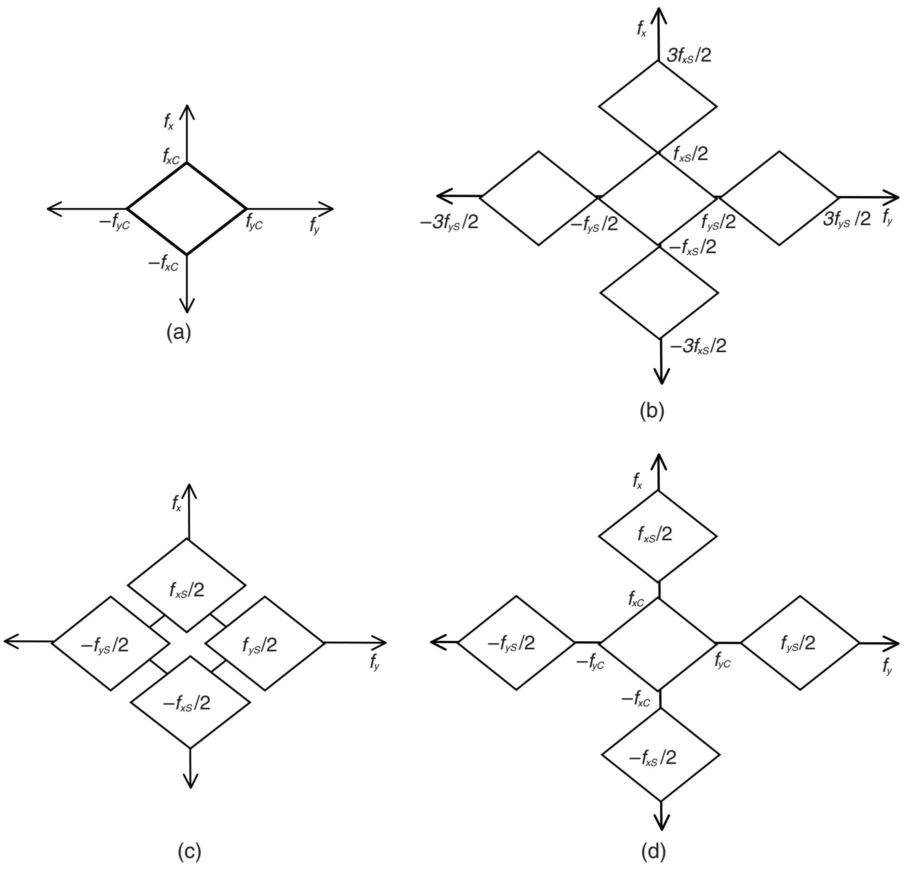


Figure 2.3 Fourier domain illustration of aliasing distortion due to sampling a continuous image: (a) Fourier transform of a continuous lowpass image, (b) Fourier transform of the sampled image with sampling frequencies exactly equal to the respective Nyquist frequencies, (c) Fourier transform of the sampled image with undersampling, and (d) Fourier transform of the sampled image for oversampling.

F_{xS} and F_{yS} samples per unit distance. If the sampling rates are below the Nyquist rates, namely, F_{xS} and F_{yS}, then the continuous image cannot be recovered exactly from its samples by filtering and so the reconstructed image suffers from a distortion known as *aliasing distortion*. This can be inferred from Figure 2.3. Figure 2.3a shows the Fourier transform of a continuous lowpass image with cutoff frequencies of f_{xC} and f_{yC} in the two spatial directions. Figure 2.3b shows the Fourier transform of the sampled image, where the sampling frequencies equal twice the respective cutoff frequencies. It is seen from Figure 2.3b that there is no overlap of the replicas and that the Fourier transform of the sampled image is identical to that of the continuous image in the region $\left[(-f_{xC}, f_{xC}) \times \left(-f_{yC}, f_{yC}\right)\right]$, and therefore, it can be recovered by filtering the sampled image by an ideal lowpass filter with cutoff frequencies equal to half the sampling frequencies. When the sampling rates are less than the Nyquist rates, the replicas overlap and the portion of the Fourier transform in the region specified by $\left[(-f_{xC}, f_{xC}) \times \left(-f_{yC}, f_{yC}\right)\right]$ no longer corresponds to that of the continuous image, and therefore, exact recovery is not possible, see Figure 2.3c. In this case, the frequencies above $f_{xS}/2$ and $f_{yS}/2$ *alias* themselves as low frequencies by folding over and hence the name aliasing distortion. The frequencies $f_{xS}/2$ and $f_{yS}/2$ are called the *fold over* frequencies. When the sampling rates are greater than the corresponding Nyquist rates, the replicas do not overlap and there is no aliasing distortion as shown in Figure 2.3d.

Example 2.1 Read a grayscale image, downsample it by a factor of M, and reconstruct the full size image from the downsampled image. Now, prefilter the original image with a lowpass filter and repeat downsampling and reconstruction as before. Discuss the effects. Choose a value of 4 for M.

Solution Let us read Barbara image, which is of size 661×628 pixels. Figure 2.4a is the original image downsampled by 4 without any prefiltering. The reconstructed image is shown in Figure 2.4b. The image has been cropped so as to have a manageable size. We can clearly see the aliasing distortions in both figures—the patterns in the cloth. Now, we filter the original image with a Gaussian lowpass filter of size 7×7 and then downsample it as before. Figures 2.4c,d correspond to the downsampled and reconstructed images with prefiltering. We see no aliasing distortions in the downsampled and reconstructed images when a lowpass prefiltering is applied before downsampling. However, the reconstructed image is blurry, which is due to lowpass filtering.

```
% Example2_1.m
% Resamples an original image to study the
% effects of undersampling
%
A = imread('barbara.tif');
[Height,Width,Depth] = size(A);
if Depth == 1
    f = A;
else
    f = A(:,:,1);
end
% downsample by a factor of M, no prefiltering
M = 4;
f1 = f(1:M:Height,1:M:Width);
figure,imshow(f1), title(['Downsampled by ' num2str(M) ':No
prefiltering'])
f1 = imresize(f1,[Height,Width],'bicubic');%reconstruct to full size
figure,imshow(f1(60:275,302:585))
title('Reconstructed from downsampled image: No prefiltering')
%
```

```
% downsample by a factor of M, after prefiltering
% use a gaussian lowpass filter
f = imfilter(f,fspecial('gaussian',7,2.5),'symmetric','same');
f1 = f(1:M:Height,1:M:Width);
figure,imshow(f1), title(['Downsampled by ' num2str(M) ' after
prefiltering'])
f1 = imresize(f1,[Height,Width],'bicubic');
figure,imshow(f1(60:275,302:585))
title('Reconstructed from downsampled image: with prefiltering')
```

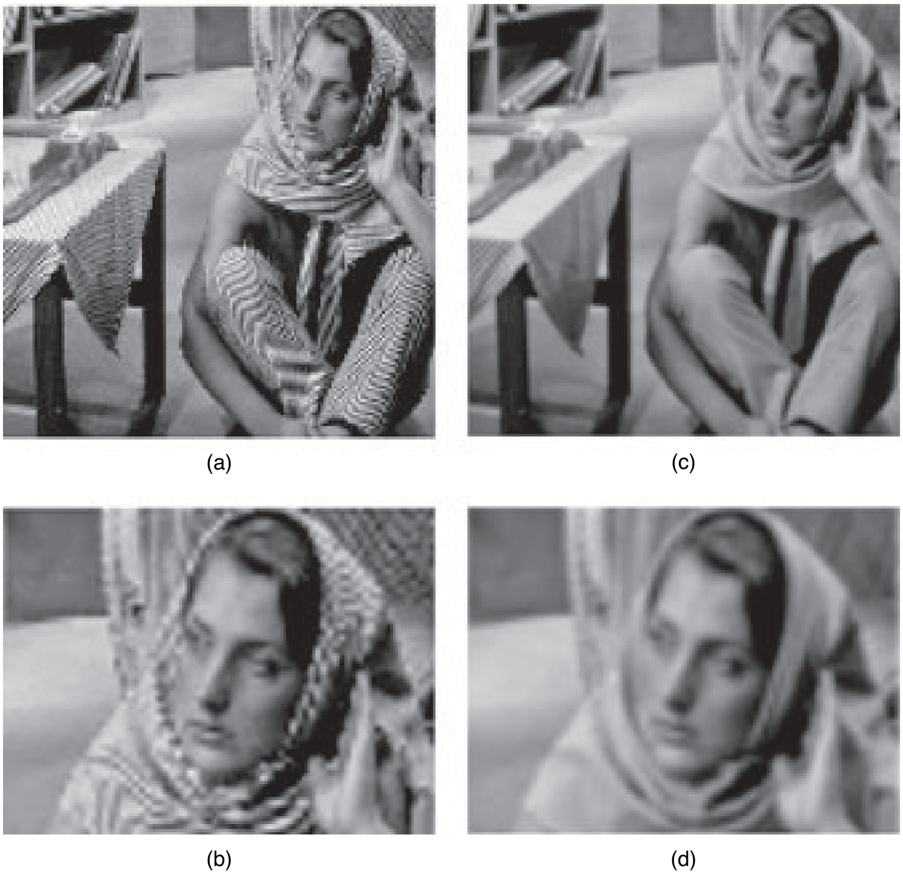


Figure 2.4 Illustration of aliasing distortion in a real image: (a) original image downsampled by a factor of 4 in both dimensions with no prefiltering, (b) reconstructed image and cropped to size 215 $\times$ 275, (c) original image prefiltered by a 7 $\times$ 7 Gaussian lowpass filter and then downsampled by 4 in both directions, (d) image reconstructed from that in (c). In (b) and (d), the reconstruction is carried out using bicubic spline interpolation.

2.2.2 Nonideal Sampling

Practical sampling devices use rectangular pulses of finite width. That is, the sampling function is an array of rectangular pulses rather than impulses. Therefore, the sampling function can be written as

$$s(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} p(x - m\Delta x, y - n\Delta y) \tag{2.21}$$

where $p(x, y)$ is a rectangular pulse of finite extent and of unit area. Then the sampling process is equivalent to convolving the continuous image with the pulse $p(x, y)$ and then sampling with a Dirac delta function [1]. The net effect of sampling with a finite-width pulse array is equivalent to prefiltering the image with a lowpass filter corresponding to the pulse $p(x, y)$ followed by ideal sampling. The lowpass filter will blur the image. This is the additional distortion to the aliasing distortion that we discussed above. To illustrate the effect the nonideal sampling has on the sampled image, let us look at the following example.

Example 2.2 Consider an image shown in Figure 2.5a, where the image is a rectangular array of size 64×64 pixels. It is sampled by a rectangular pulse of size 32×32 pixels as shown in Figure 2.5b. The sampled image is shown in Figure 2.5c. We can clearly see the smearing effect of sampling with a rectangular pulse of finite width. The 2D Fourier transforms of the input image, sampling pulse, and the sampled image are shown in Figures 2.5d–f, respectively. Since the smearing effect is equivalent to lowpass filtering, the Fourier transform of the sampled image is a bit narrower than that of the input image. The Fourier transform of an ideal impulse and the Fourier transform of the corresponding sampled image are shown in Figures 2.5g,h, respectively. In contrast to the sampling with a pulse of finite width, we see that the Fourier transform of the ideally sampled image is the same as that of the input image. Finally, the reconstructed image using an ideal lowpass filter is shown in Figure 2.5i. It is identical to the sampled image because we employed sampling with a pulse of finite width.

```
% Example2_2.m
% Example showing the effect of image sampling with a
% finite width pulse
% Two cases are possible: 'delta' and 'pulse'
% delta corresponds to impulse sampling and pulse
% to sampling with finite width pulse.
% image array size is NxN pixels
% actual image area to be sampled is MxM
% TxT is the sampling pulse area

samplingType = 'pulse'; % 'delta' or 'pulse'
N = 256; % array size is NxN
f = zeros(N,N); % image to be sampled
```

```
M = 32; % image pulse width
M1 = N/2 - M + 1; % image beginning point
M2 = N/2+M; % image end point
f(M1:M2,M1:M2) = 1; % test image has a value 1
%
p = zeros(N,N); % sampling rectangular pulse
if strcmpi(samplingType,'delta')
    p(128,128) = 1;
end
if strcmpi(samplingType,'pulse')
    T = M/2; % sampling pulse width
    T1 = N/2 - T + 1; % sampling pulse beginning point
    T2 = N/2+T; % sampling pulse end point
    p(T1:T2,T1:T2) = 1; % sampling pulse; width = 1/2 of image
end
fs = conv2(f,p,'same'); % convolution of image & pulse
figure,imshow(f,[]),title('Input Image')
figure,imshow(p,[]), title('Sampling Pulse')
figure,imshow(fs,[]),title('Sampled Image')
%
figure,imshow(log10(1+5.5*abs(fftshift(fft2(f)))),[])
title('Fourier Transform of input image')
if strcmpi(samplingType,'delta')
    figure,imshow(log10(1+1.05*abs(fftshift(fft2(p)))),[])
    title('Fourier Transform of Delta Function')
end
if strcmpi(samplingType,'pulse')
    figure,imshow(log10(1+5.5*abs(fftshift(fft2(p)))),[])
    title('Fourier Transform of rectangular pulse')
end
%
Fs = fftshift(fft2(fs));
figure,imshow(log10(1+5.5*abs(Fs)),[])
title('Fourier Transform of sampled image')
% h is the 2D reconstruction filter
%h = (sinc(0.1*pi*(-128:127)))' * sinc(0.1*pi*(-128:127));
h = (sinc(0.2*pi*(-128:127)))' * sinc(0.2*pi*(-128:127));
H = fft2(h);
figure,imshow(ifftshift(ifft2(fft2(fs).*H,'symmetric')),[])
title('Reconstructed image')
```

2.2.3 Nonrectangular Sampling Grids

So far our discussion has been on image sampling on a rectangular grid. This is the most commonly used sampling grid structure because many display systems such as TV use raster scanning, that is, scanning from left to right and top to bottom. Also, most digital cameras have their sensors built in rectangular grid arrays.

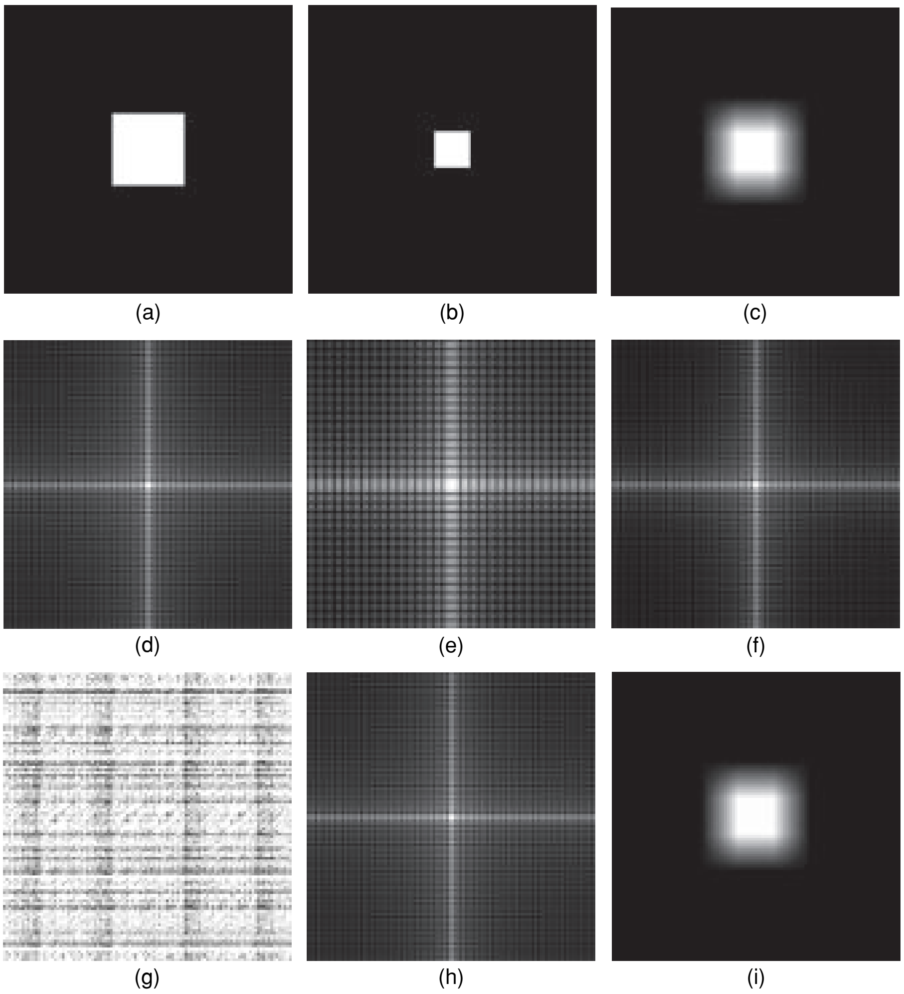


Figure 2.5 Effect of nonideal sampling: (a) a 64 × 64 BW image, (b) rectangular sampling image of size 32 × 32 pixels, (c) sampled image, (d) 2D Fourier transform of (a), (e) 2D Fourier transform of (b), (f) 2D Fourier transform of the sampled image in (b), (g) 2D Fourier transform of an ideal impulse, (h) 2D Fourier transform of (a) sampled by impulse, and (i) image in (c) obtained by filtering by an ideal lowpass filter.

However, it is possible to use nonrectangular grids, such as hexagonal sampling grid, to acquire a digital image. An advantage of using hexagonal sampling grid is that the acquired image has 13.4% less data than that acquired using the rectangular sampling grid [2]. It has also been found that edge detection is more efficient with hexagonally sampled images. Hexagonal sampling is used widely in machine vision and biomedical imaging. Although the compression standards such as Moving Picture Experts Group (MPEG) use rectangular grid structure for coding still and moving images especially for motion estimation and compensation, it may be more efficient to employ hexagonal grids for such purposes for better accuracy in motion estimation and higher compression ratio.

Let us briefly describe here the process of converting pixels from rectangular to hexagonal sampling grids and vice versa and illustrate the processes by a couple of examples. The set of points in the $(n_1,\ n_2)$-plane are transformed linearly into the set of points $(t_1,\ t_2)$ using the transformation

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} \tag{2.22}$$

In equation (2.22), the vectors $\mathbf{v_1} = [v_{11}\ \ v_{21}]'$and $\mathbf{v}_2 = [v_{12}\ \ v_{22}]'$ are linearly independent. For the set of points (t_1, t_2) to be on a hexagonal sampling grid, the transformation takes the form

$$\mathbf{v} = \begin{bmatrix} \Delta\mathbf{y} & \Delta\mathbf{y} \\ \Delta\mathbf{x} & -\Delta\mathbf{x} \end{bmatrix} \tag{2.23}$$

Example 2.3 Convert a 256 × 256 array of pixels of alternating black and white (BW) dots to a hexagonal sampling grid and display both arrays as images. Assume the sample spacing to be equal to unity.

Solution We will use the linear transformation

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

to transform pixels in the rectangular array into the hexagonal array. Note that the range for the hexagonal grid is $[(2, 512) \times (-255,255)]$. Figures 2.6a,b show the rectangular and hexagonal sampling grids, respectively, while Figures 2.6c,d correspond to the zoomed versions with a zoom factor of 4. The MATLAB code for Example 2.3 is listed below.

Example 2.4 In this example, we will convert the Barbara image from the original rectangular sampling to hexagonal sampling grid using equation (2.22) and display them. Figure 2.7a is the cropped image of Barbara using rectangular sampling grid. The corresponding hexagonally sampled image is shown in Figure 2.7b. If M and N are the number of rows and columns, respectively, of an image, then the size of the hexagonally sampled image will be $M + N - 1 \times M + N$. In the MATLAB code listed below, conversion from hexagonal grid back to rectangular grid is also included. The code reads the specified image, converts to hexagonal grid, displays the respective images, converts hexagonal to rectangular grid, and displays the valid region.

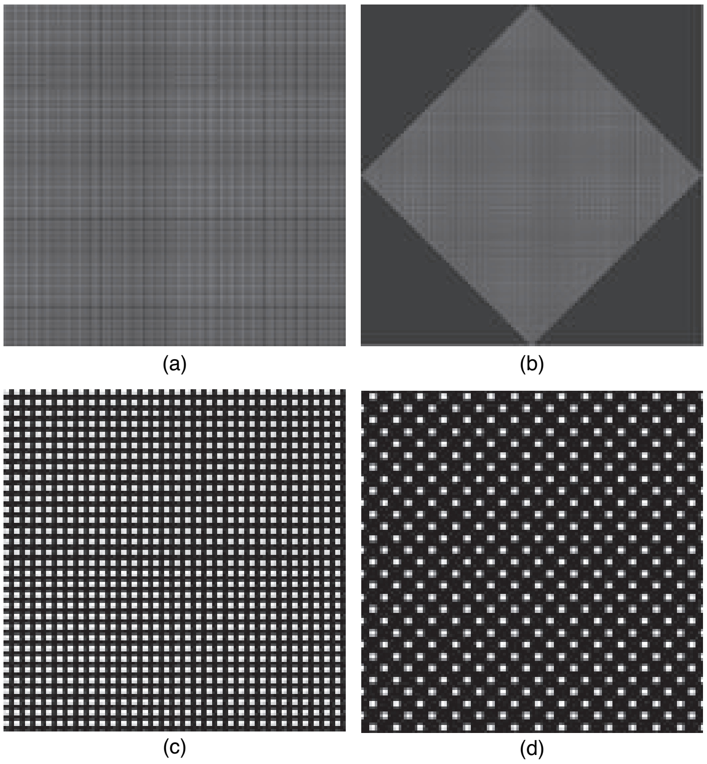


Figure 2.6 Converting from rectangular to hexagonal sampling grids: (a) pixels on a rectangular grid with unit spacing in both horizontal and vertical directions, (b) pixels on hexagonal grids after applying the transformation in equation (2.22), (c) zoomed version of (a), and (d) zoomed version of (b). In both (c) and (d), the zoom factor is 4.

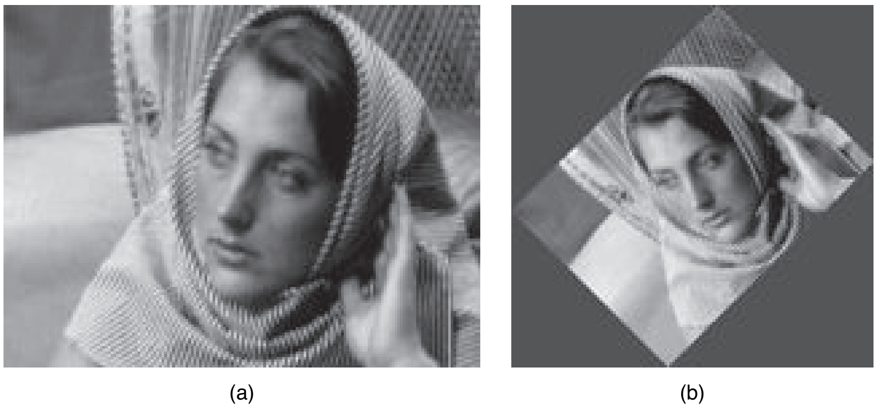


Figure 2.7 Converting a real image from rectangular to hexagonal grids: (a) cropped original image in rectangular grids and (b) corresponding hexagonally sampled image.

```
% Example2_3.m
% convert rectangular to hexagonal grid

clear
close all
M = 256; N = 256;
f = uint8(zeros(N,N));
g = uint8(zeros(M+N,M+N));
% create an image with alternating black & white
dots
% for visibility sake, these are R/2xR/2 squares
%
R = 2; % pixel replication factor
for k = 1:R:M
    for l = 1:R:N
        for k1 = 1:R/2
            for l1 = 1:R/2
                f(k+k1-1,l+l1-1) = 255;
            end
        end
    end
end
figure,imshow(f), title('Rectangular grid')
Fig1 = figure;
imshow(f), title('Rectangular grid')
zoom(Fig1,4)
%
% rectangular to hexagonal transformation
V = [1 1; 1 -1];
for n1 = 1:M
    for n2 = 1:N
        t1 = V(1,1)*n1 + V(1,2)*n2;
        t2 = V(2,1)*n1 + V(2,2)*n2;
        g(513-t1,256+t2) = f(n1,n2);
    end
end
figure,imshow(g), title('Hexagonal grid')
g1 = imcrop(g,[130 130 230 230]);
Fig2 = figure;
imshow(g1), title('Hexagonal grid')
zoom(Fig2,4)
```

```
% Example2_4.m
% convert rectangular to hexagonal grid
% using linear transformation
% Sample spacing is DX in the row dimension
% and DY in the column dimension
% transformation is [t1 t2]' = V*[n1 n2]';
% where V = [DX DY; DX -DY];
% In our example we assume DX = DY = 1;
clear
close all
```

```
%A = imread('cameraman.tif');
A = imread('barbara.tif');
%{
[M,N,D] = size(A);
N2 = 2*N;
M2 = 2*M;
%}
D = 3;
M = 216; N = 284; M2 = 2*M; N2 = 2*N;
%M = 164; N = 205; M2 = 2*M; N2 = 2*N;
if D > 1
    %f = A(:,:,1);
    f = A(60:275,302:585,1);
    %f = A(182:345,76:280,1);
else
    f = A;
end
figure,imshow(f), title('Rectangular grid')
g = uint8(zeros(M+N,M+N));% transformed image twice as big as
the input
%
% Convert rectangular to hexagonal sampling
V = [1 1; 1 -1]; % rectangular to hexagonal transformation
for n1 = 1:M
    for n2 = 1:N
        t1 = V(1,1)*n1 + V(1,2)*n2;
        t2 = V(2,1)*n1 + V(2,2)*n2;
        g(t1,N+t2) = f(n1,n2);
    end
end
figure,imshow(g'),title('Hexagonal grid')
% Now revert to rectangular grid
VI = inv(V);
f1 = uint8(zeros(M+N,M+N));
for t1 = 2:M+N
    %for t2 = -N+1:N-1
    for t2 = -N+1:M-1
        n1 = floor(VI(1,1)*t1 + VI(1,2)*t2);
        n2 = floor(VI(2,1)*t1 + VI(2,2)*t2);
        %f1(n1+M/2,n2+N/2) = g(N2+1-t1,M+t2);
        %f1(n1+round((N-1)/2),n2+round((M-1)/2)) = g(N2+1- t1,M+t2);
        f1(n1+round((N-1)/2),n2+round((M-1)/2)) = g(M+N+1- t1,N+t2);
    end
end
% image is flipped right and down
f1 = f1';
%f1(:,1:N2) = f1(:,N2:-1:1);
f1(:,1:M+N) = f1(:,M+N:-1:1);
f1(1:M+N,:) = f1(M+N:-1:1,:);
figure,imshow(f1(round(N/2)+2:round(N/2)+M+1,round(M/2)+2:rou
nd(M/2)+N+1))
title('Rectangular grid')
```

2.3 IMAGE QUANTIZATION

As indicated in Figure 2.1, the image samples have a continuum of values. That is, the samples are analog and must be represented in digital format for storage or transmission. Because the number of bits (binary digits) for representing each image sample is limited—typically 8 or 10 bits, the analog samples must first be *quantized* to a finite number of levels and then the particular level must be coded in binary number. Because the quantization process compresses the continuum of analog values to a finite number of discrete values, some distortion is introduced in the displayed analog image. This distortion is known as *quantization noise* and might manifest as patches, especially in flat areas. This is also known as *contouring*.

In simple terms, a scalar quantizer observes an input analog sample and outputs a numerical value. The output numerical value is a close approximation to the input value. The output values are predetermined corresponding to the range of the input and the number of bits allowed in the quantizer. Formally, we define a scalar quantizer $Q(\cdot)$ to be a mapping of input *decision intervals* $\{d_k, k = 1, 2, \ldots, L + 1\}$ to output or *reconstruction levels* $\{r_k, k = 1, \ldots, L\}$. Thus,

$$Q(x) = \hat{x} = r_k \tag{2.24}$$

We assume that the quantizer output levels are chosen such that

$$r_1 \prec r_2 \prec \cdots \prec r_L \tag{2.25}$$

The number of bits required to address any one of the output levels is

$$B = \lceil \log_2 L \rceil \text{ (bits)} \tag{2.26}$$

where $\lceil x \rceil$ is the nearest integer equal to or larger than x.

A design of a quantizer amounts to specifying the decision intervals and corresponding output levels and a mapping rule. Since there are many possibilities to partition the input range, we are interested in the optimal quantizer that minimizes a certain criterion or cost function. In the following section, we will develop the algorithm for such an optimal quantizer.

2.3.1 Lloyd–Max Quantizer

A Lloyd–Max quantizer is an optimal quantizer in the mean square error (MSE) sense [3,4]. It depends on the input probability density function (pdf) and the number of levels allowed in the quantizer. We point out here that the input analog sample is a random variable. Hence, it is described by a pdf. The quantizer design is as follows: Let x be a real-valued scalar random variable with a continuous pdf $p_x(x)$. The optimal MSE or Lloyd–Max quantizer design amounts to determining the input decision intervals $\{d_k\}$ and corresponding reconstruction levels $\{r_k\}$ for given L levels such that the MSE is minimized:

$$\text{MSE} = E\left\{(x-\hat{x})^2\right\} = \int_{d_1}^{d_{L+1}} (x-\hat{x})^2\, p_x(x)\, dx \tag{2.27}$$

In equation (2.27), E denotes the expectation operator. Note that the output value is constant and equal to r_j over the input range $\left[d_j, d_{j+1}\right)$. Therefore, equation (2.27) can be written as

$$\text{MSE} = \sum_{j=1}^{L} \int_{d_j}^{d_{j+1}} \left(x-r_j\right)^2 p_x(x)\, dx \tag{2.28}$$

Minimization of equation (2.28) is obtained by differentiating the MSE with respect to d_j and r_j and setting the partial derivatives to zero. Thus,

$$\frac{\partial \text{MSE}}{\partial d_j} = \left(d_j - r_{j-1}\right)^2 p_x\left(d_j\right) - \left(d_j - r_j\right)^2 p_x\left(d_j\right) = 0 \tag{2.29a}$$

$$\frac{\partial \text{MSE}}{\partial r_j} = 2\int_{d_j}^{d_{j+1}} \left(x - r_j\right) p_x(x)\, dx = 0,\ 1 \le j \le L \tag{2.29b}$$

From equation (2.29a), we obtain

$$\left(d_j - r_{j-1}\right)^2 = \left(d_j - r_j\right)^2 \tag{2.30}$$

Because $r_j \succ r_{j-1}$, equation (2.30) after simplification yields

$$d_j = \frac{r_j + r_{j-1}}{2} \tag{2.31a}$$

From equation (2.29b), we get

$$r_j = \frac{\int_{d_j}^{d_{j+1}} x p_x(x)\, dx}{\int_{d_j}^{d_{j+1}} p_x(x)\, dx} \tag{2.31b}$$

Equation (2.31a) states that the optimal decision intervals are the midpoints of the corresponding optimal output levels and equation (2.31b) states that the optimal output levels are the centroids of the decision intervals. We see that the Lloyd–Max quantizer results in transcendental equations, and they can be solved by iteration. Fortunately, many books tabulate the optimal decision intervals and corresponding output levels for given input pdf and L. These tables assume that the input analog signal to have unit variance. To obtain the optimal quantizer decision intervals and output levels for an analog signal having a specific variance, one must multiply the tabulated values by the standard deviation of the actual signal.

2.3.2 Uniform Quantizer

When the pdf of the analog sample is uniform, the decision intervals and output levels of the Lloyd–Max quantizer can be computed analytically as shown below. In this case, the decision intervals are all equal as well as the intervals between the output levels and the quantizer is called a *uniform* quantizer. The uniform pdf of the input image is given by

$$p_x(x) = \frac{1}{x_{\max} - x_{\min}} = \frac{1}{d_{L+1} - d_1} \tag{2.32}$$

Using equation (2.32) in (2.31b), we get

$$r_j = \frac{\dfrac{\left(d_{j+1} - d_j\right)^2}{2\left(x_{\max} - x_{\min}\right)}}{\dfrac{\left(d_{j+1} - d_j\right)}{\left(x_{\max} - x_{\min}\right)}} = \frac{\left(d_{j+1} + d_j\right)}{2} \tag{2.33}$$

From equations (2.33) and (2.31a), we can write

$$d_{j+1} - d_j = d_j - d_{j-1} = \Delta, 2 \le j \le L \tag{2.34}$$

From the above equation, we see that the interval between any two consecutive decision boundaries is the same and equal to the quantization step size Δ. Using equation (2.34), we can also express d_j in terms of d_{j+1} and d_{j-1} as

$$d_j = \frac{d_{j+1} + d_{j-1}}{2} \tag{2.35}$$

The quantization step size of the uniform quantizer is related to the number of levels L or the number of bits B, as given in equation (2.36).

$$\Delta = \frac{x_{\max} - x_{\min}}{L} = \frac{d_{L+1} - d_1}{2^B} \tag{2.36}$$

Finally, combining equations (2.33) and (2.34) we can express the reconstruction levels by

$$r_j = \frac{d_j + \Delta + d_j}{2} = d_j + \frac{\Delta}{2} \tag{2.37}$$

From equation (2.37), we find that the intervals between the output levels are also equal to Δ. Thus, the design of the optimal or Lloyd–Max quantizer for a signal with uniform distribution is carried out as follows:

1. Divide the input range $(x_{\min}, x_{\max})$ into L equal steps Δ as given in equation (2.36) with $d_1 = x_{\min}$ and $d_{L+1} = x_{\max}$.
2. Determine the L output levels from equation (2.37).
3. Quantize a sample x according to $Q(x) = r_j$, if $d_j \leq x \prec d_{j+1}$.

2.3.3 Quantizer Performance

For the uniform quantizer, the quantization error $e = x - \hat{x}$ has the range $(-\frac{\Delta}{2}, \frac{\Delta}{2})$, and it is uniformly distributed as well. Therefore, the MSE σ_q^2 for the uniform quantizer can be found from

$$\sigma_q^2 = \frac{1}{\Delta} \int_{-\Delta/2}^{\Delta/2} e^2 de = \frac{\Delta^2}{12} \tag{2.38}$$

We can express the performance of a uniform quantizer in terms of the achievable signal-to-noise ratio (SNR). The noise power due to quantization is given by equation (2.38). The variance of a uniformly distributed random variable in the range $(x_{\min}, x_{\max})$ is evaluated from

$$\sigma_x^2 = \int_{x_{\min}}^{x_{\max}} (x - \mu)^2 \, p_x(x) dx = \frac{1}{(x_{\max} - x_{\min})} \int_{x_{\min}}^{x_{\max}} (x - \mu)^2 \, dx \tag{2.39}$$

Where μ is the average value of the input signal, as given by

$$\mu = \frac{1}{(x_{\max} - x_{\min})} \int_{x_{\min}}^{x_{\max}} x dx = \frac{x_{\max} + x_{\min}}{2} \tag{2.40}$$

Using equation (2.40) in (2.39) and with some algebraic manipulation, we find that the signal variance equals

$$\sigma_x^2 = \frac{(x_{\max} - x_{\min})^2}{12} \tag{2.41}$$

Therefore, the SNR of a uniform quantizer is found to be

$$\text{SNR} = \frac{\sigma_x^2}{\sigma_q^2} = \frac{(x_{\max} - x_{\min})^2}{\Delta^2} = \frac{(x_{\max} - x_{\min})^2}{\dfrac{(x_{\max} - x_{\min})^2}{2^{2B}}} = 2^{2B} \tag{2.42}$$

In dB, a B-bit uniform quantizer yields an SNR of

$$\text{SNR} = 10 \log 2^{2B} \cong 6B \quad \text{(dB)} \tag{2.43}$$

From equation (2.43), we find that each additional bit in the quantizer improves its SNR by 6 dB. Stated in another way, the Lloyd–Max quantizer for signals with uniform pdf gives an SNR of 6 dB/bit of quantization.

Example 2.5 Read a BW image with 8 bits/pixel and requantize it to B bits/pixel using a uniform quantizer. Display both the original and requantized images and observe the differences, if any. Plot the SNR in dB due to quantization noise. For display purpose, assume B to be 3 and 5 bits. For plotting, vary B from 1 to 7 bits.

Solution For the uniform quantizer with B bits of quantization, the quantization step size is found to be

$$\Delta = \frac{255}{2^B}$$

Then, the decision boundaries and output levels can be calculated from

$$d_k = d_{k-1} + \Delta, 2 \le k \le L + 1; \quad d_1 = 0, d_{L+1} = 255$$
$$r_k = d_k + \frac{\Delta}{2}, 1 \le k \le L$$

After calculating the decision boundaries and reconstruction levels, the B-bit quantizer assigns to each input pixel an out value using the rule:

$$\text{output } r_k \text{ if } d_k \le \text{input} \prec d_{k+1}, 1 \le k \le L$$

We will use the Barbara image for this example. Since this image is rather large in size, we will crop it to a smaller size. Figure 2.8a is the cropped original 8-bpp image, and Figures 2.8b,c correspond to the requantized images at 3 and 5 bpp, respectively. We see that flat areas appear very patchy especially in the 3-bpp image

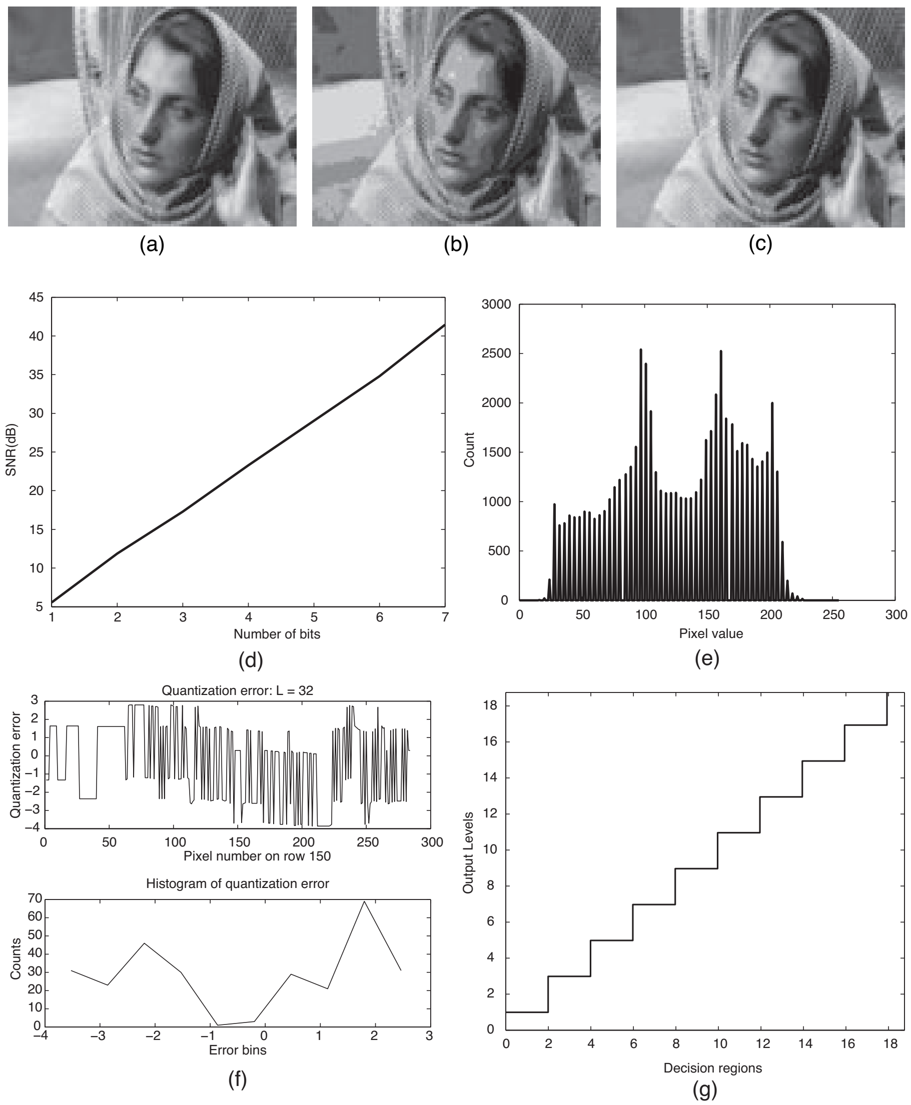


Figure 2.8 An example of a uniform quantizer: (a) cropped original 8-bpp image, (b) image in (a) requantized to 3 bpp using a uniform quantizer, (c) image in (a) requantized to 5 bpp using a uniform quantizer, (d) SNR due to quantization versus bpp, (e) histogram of the input 8-bpp image, (f) error due to quantization: the top figure is the pixel error and the bottom figure is the corresponding histogram, and (g) plot of input decision boundaries versus reconstruction levels.

as compared with the 5-bpp image. Because of the large quantization step size, a large neighborhood of pixels gets quantized to the same level and this makes the image look patchy in the flat areas. The SNR in dB due to quantization noise is shown in Figure 2.8d for 1–7 bits of quantization. The SNR versus bits is linear over a large range of quantization bits. From the plot we find the slope to be around, 6.3 dB/bit, which is slightly larger than that obtained from analysis. At 6 bpp, the SNR is around 35 dB and the distortion is hardly noticeable. It should be pointed out that the uniform quantizer is optimal only when the pdf of the input image is uniform. However, the histogram of the image in this example is not exactly uniform, as shown in Figure 2.8e. The error due to quantization for $L = 32$ is shown in Figure 2.8f (top figure), where we notice that the quantization error ranges between -4 and $+4$, as expected. The corresponding error histogram is shown in the bottom figure, where we find the error distribution to be broad. It is not exactly uniform due to the particular nature of the image we picked and due to the input image already being digitized. Finally, a plot of the input decision intervals versus the reconstruction levels for the uniform quantizer is shown in Figure 2.8g, which is of the type staircase, as expected. A listing of the MATLAB codes for this example is shown below. Both assignment rules are included in the following code.

```
% Example2_5.m
% Uniform quantizer design
% Computes decision intervals and output levels
% for quantization bits from 1 to B
% Quantizes the input image to B bits and
% Computes Signal-to-Quantization noise in dB
% for 1 to B bits and plots
%
clear
close all
%
A = imread('barbara.tif');
%A = imread('cameraman.tif');
[Height,Width,Depth] = size(A);
if Depth == 1
    f = double(A);
else
    f = double(A(:,:,1));
end
if Height>512 || Width > 512
    figure,imshow(uint8(f(60:275,302:585)))
    [Counts,Bins] = imhist(uint8(f(60:275,302:585)),256);
else
    figure,imshow(uint8(f))
    [Counts,Bins] = imhist(uint8(f),256);
end
title('Image quantized to 8 bits')
% Calculate the histogram of the input image
figure,plot(Bins,Counts,'k', 'LineWidth', 2)
title('Histogram of input image')
xlabel('Pixel Value')
```

```
ylabel('Count')
%
f1 = zeros(size(f));
E = zeros(size(f));
sigmaf = std2(f);
B = 7;
snr = zeros(B,1);
% Quantization method 1 is the process of using decision intervals
% and corresponding reconstruction levels
% method 2 is similar to JPEG quantization rule
QuantizerMethod = 1;% options 1 or 2
switch QuantizerMethod
    case 1
        % Design a uniform quantizer for bits 1 to 7
        for b = 1:B
            L = 2^b;
            q = 255/L;
            q2 = q/2;
            d = linspace(0,255,L+1);
            r = zeros(L,1);
            for k = 1:L
                r(k) = d(k)+q2;
            end
            for k = 1:L
                [x,y] = find(f>d(k) & f<=d(k+1));
                for j = 1:length(x)
                    f1(x(j),y(j)) = round(r(k));
                    E(x(j),y(j)) = double(f(x(j),y(j)))-r(k);
                    % include surf plot
                end
                if b == 5 && k == 32
                    figure,subplot(2,1,1),plot(E(150,:),'k','LineWidth',2)
                    title(['Quantization error: L = ' num2str(k)])
                    xlabel('pixel # on row 150')
                    ylabel('Quantization error')
                    [Counts,Bins]= hist(E(150,:));
                    subplot(2,1,2),plot(Bins,Counts,'k','LineWidth',2)
                    title('Histogram of quantization error')
                    xlabel('Error bins'), ylabel('Counts')
                end
                clear x y
            end
            clear E
            if b == 3 || b == 5
                if Height>512 || Width > 512
                    figure,imshow(uint8(f1(60:275,302:585)))
                else
                    figure,imshow(uint8(f1))
                end
                title(['Image quantized to ' num2str(b) ' bits'])
            end
            snr(b) = 20*log10(sigmaf/std2(f-f1));
        end
        figure,plot(1:B,snr,'k','LineWidth',2)
        title('SNR Vs Bits/pixel of a uniform quantizer')
        xlabel('# bits')
        ylabel('SNR(dB)')
        figure,stairs(d(1:L),r,'k','LineWidth',2)
        title('Uniform Quantizer: Decision intervals & output levels')
        xlabel('Decision Regions')
        ylabel('Output Levels')
    case 2
        for b = 1:B
            L = 2^b;
            q = 255/L;
            f1 = round(floor(f/q + 0.5)*q);
            if b == 5
                E = f - f1;
                figure,subplot(2,1,1),plot(E(150,:),'k','LineWidth',2)
                title(['Quantization error: L = ' num2str(L)])
                xlabel('pixel # on row 150')
                ylabel('Quantization error')
                [Counts,Bins]= hist(E(150,:));
                subplot(2,1,2),plot(Bins,Counts,'k','LineWidth',2)
                title('Histogram of quantization error')
                xlabel('Error bins'), ylabel('Counts')
            end
            if b == 3 || b == 5
                if Height>512 || Width > 512
                    figure,imshow(uint8(f1(60:275,302:585)))
                else
                    figure,imshow(uint8(f1))
                end
                title(['Image quantized to ' num2str(b) ' bits'])
            end
            snr(b) = 20*log10(sigmaf/std2(f-f1));
        end
        figure,plot(1:B,snr,'k','LineWidth',2)
        title('SNR Vs Bits/pixel of a uniform quantizer')
        xlabel('# bits')
        ylabel('SNR(dB)')
end
```

2.3.4 PDF Optimized Quantizer

In the case of uniform quantizers, the pdf of the analog sample was assumed to be uniform, and therefore, we obtained the closed form solutions for optimal decision regions and output levels. Moreover, the intervals between any two consecutive decision regions as well as the intervals between any two consecutive output levels were constant. When the pdf of the input analog samples is not uniform, then the quantization steps are not constant and the optimal solutions are obtained by solving the transcendental equations (2.31). This results in a *nonuniform* quantizer and is referred to as pdf optimized quantizer. As we mentioned before, one can solve equation (2.31) iteratively to obtain the optimal solutions. One such procedure is called the Lloyd algorithm [5] and is described below.

Lloyd Algorithm

Given an input image and number of levels L of quantization:

1. Choose an initial set R_1 of L output levels, initial average distortion T, $\varepsilon \succ 0$, and set $k = 1$.
2. Partition the input pixels into regions D_k using the output set R_k according to the nearest neighbor condition, via,

$$D_k = \{f : d(f, r_k) \leq d(f, r_l); \quad \text{all } 1 \neq k\}$$

3. From the partition regions found in step 2, find the new output levels R_{k+1} as the centroids of the input partitions.
4. Calculate the new average distortion due to the new output levels. If the absolute difference between the previous and current average distortions relative to the previous average distortion is less than ε, stop. The input decision boundaries are computed as the midpoints of the output levels. Otherwise, set k to $k + 1$ and go to step 2.

In the following example, we will take up the design of a pdf-optimized quantizer using a single image and compare the results with that of the uniform quantizer of Example 2.5.

Example 2.6 Design an optimal quantizer for an input BW image with 8 bpp using Lloyd algorithm. Display the requantized images at 3 and 5 bpp. Plot the SNR due to quantization for bits 1–7.

Solution We will use the same image here as was used in Example 2.5. Using Lloyd's algorithm, we design the quantizers for 3 and 5 bpp. The requantized images at 3 and 5 bpp are shown in Figures 2.9a,b, respectively. There are some improvements in the flat areas compared with that of the corresponding uniform quantizer. The SNR values are 20.1 and 33.68 dB for 3 and 5 bpp, respectively, for the nonuniform quantizer, whereas they are 17.3 and 29.04 dB, respectively, for the uniform quantizer. Figure 2.9c shows a plot of the decision regions versus output levels of the nonuniform quantizer for $L = 32$ levels. The number of pixels belonging to the different output levels is plotted against output levels and is shown in Figure 2.9d (top figure), while the bottom Figure 2.9d shows the histogram of the input image for the same number of bins—32 in the example. The two are nearly identical and the slight difference is due to small number of bits of quantization. Thus, we have designed the pdf-optimized quantizer for the input image. Overall, we find the nonuniform quantizer to perform much better than the uniform quantizer for the same number of bits of quantization. Another important point of observation is that the iterations converge much faster if we use a difference distortion relative to the previous distortion rather than absolute difference in distortions. In fact, the average number of iterations is 12 when the relative distortion measure is used to check convergence,

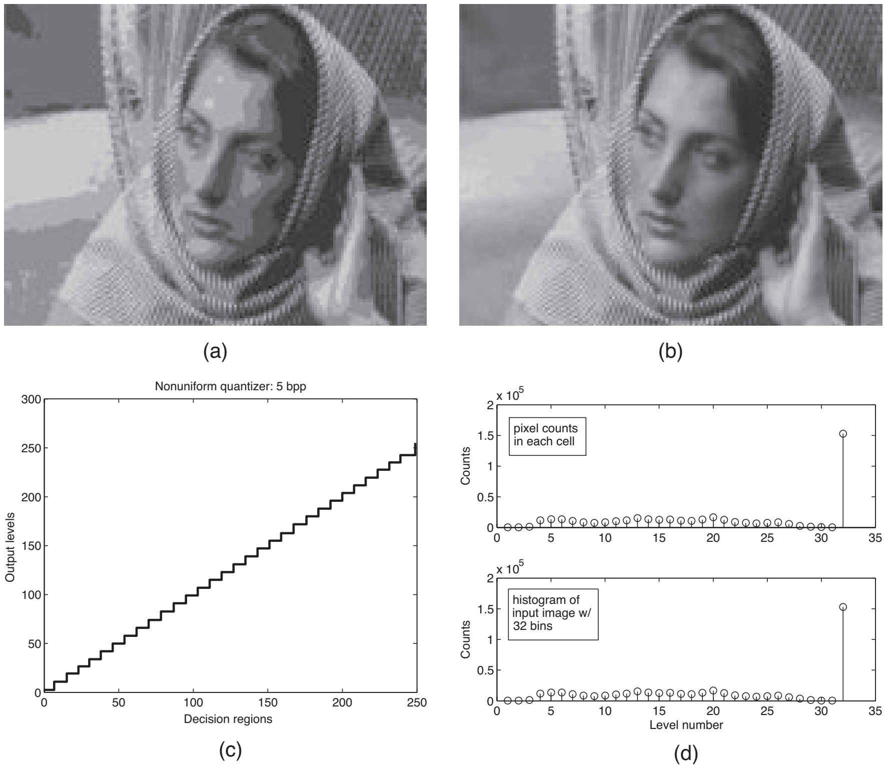


Figure 2.9 An example of a nonuniform quantizer: (a) image requantized to 3 bpp, (b) image requantized to 5 bpp, (c) plot of input decision regions versus output levels of the nonuniform quantizer, (d) the top figure is the count of quantized pixels belonging to the different output levels versus output levels, and the bottom figure is histogram of the input image for the same number of levels as in the top figure.

while it is 82 when the absolute difference measure is used for the same ε value of 0.1. The MATLAB code for the Lloyd algorithm is listed below. Finally, Table 2.1 lists the input decision boundaries and corresponding output levels for 3 and 4 bits of quantization generated from the Barbara image.

```
% Example2_6.m
% Design of a Nonuniform Quantizer for a given input image
% Starts with an initial set of output levels and
% iteratively builds a final set that results in the
% least mean square error between the input and
% reconstructed image
clear
close all

A = imread('cameraman.tif');
%A = imread('barbara.tif');
[Height,Width,Depth] = size(A);
```

```
if Depth == 1
    f = double(A);
else
    f = double(A(:,:,1));
end
%
%figure,imshow(A(60:275,302:585,1));% cropped for Barbara image
figure,imshow(A(:,:,1));
title('Original 8 bpp image')
%
sigmaf = std2(f); % std dev of input image
B = 7; % bits of quantization
snr = zeros(B,1); % signal-to-quantization noise in dB
err = 0.1e-01; % accuracy to terminate iteration
%
for b = 5:5
    L = 2^b; % number of output levels
    %C = linspace(0.5,255,L); % initial output levels
    C = linspace(2,253,L);
    PartitionRegion = zeros(L,Height,Width);
    % partitions consisting of input pixels
    RegionCount = zeros(L,1); % number of elements in each partition
    RegionSum = zeros(L,1); % sum of pixels in a given partition region
    Distortion = zeros(L,1); % minimum distortion in a partition region
    TotalDistortion = 1.0e+5; % total minimum distortion
    %TD = abs(TotalDistortion-sum(Distortion)); % initial total distortion
    % alternate convergence measure
    TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
    % start iteration
    It = 1; % iteration number
    while TD >= err
        for r = 1:Height
            for c = 1:Width
                d1 = abs(f(r,c)-C(1));
                J = 1;
                for k = 2:L
                    d2 = abs(f(r,c)-C(k));
                    if d2<d1
                        d1 = d2;
                        J = k;
                    end
                end
                RegionCount(J) = RegionCount(J) + 1;
                RegionSum(J) = RegionSum(J) + f(r,c);
                PartitionRegion(J,r,c) = f(r,c);
                Distortion(J) = Distortion(J)+(f(r,c)-C(J))*(f(r,c)-C(J));
            end
        end
        Index = logical(RegionCount == 0);% check for empty regions
        RegionCount(Index) = 1;% set empty regions to count 1
        Distortion = Distortion ./RegionCount;% average minimum distortions
        %TD = abs(TotalDistortion-sum(Distortion));
        % alternate convergence measure
        TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
        C = RegionSum ./RegionCount; % refined output levels
        TotalDistortion = sum(Distortion);% total minimum distortion
        It = It + 1;
    end
    clear Index
    sprintf('Number of iterations = %d; \t bpp = %d',It-1,b)
    %sprintf('SNR = %4.2f dB',10*log10(sigmaf*sigmaf/TotalDistortion))
    DR = zeros(L+1,1);
    DR(L+1) = 255;
    f1 = zeros(size(f));
    for k = 2:L
        DR(k) = (C(k-1)+C(k))/2;
    end
    % plot output levels against decision regions
    if b == 5
        figure,stairs(DR(1:L),C,'k','LineWidth',2)
        title(['Nonuniform quantizer: ' num2str(b) ' bpp'])
        xlabel('Decision Regions')
        ylabel('Output Levels')
        figure,subplot(2,1,1),stem(RegionCount/(It-1),'k')
        title(['Comparison of output & input histograms at ' num2str(b) ' bpp'])
        ylabel('Counts')
        [Counts,Bins] = imhist(uint8(f),32);
        subplot(2,1,2),stem(Counts,'k')
        xlabel('Level number'), ylabel('Counts')
    end
    % decode using the codebook generated
    for r = 1:Height
        for c = 1:Width
            d1 = abs(f(r,c)-C(1));
            for k = 2:L
                d2 = abs(f(r,c)-C(k));
                if d2<d1
                    d1 = d2;
                    J = k;
                end
            end
            f1(r,c) = C(J);
        end
    end
    % decode using decision regions & codebook
    %{
    for k = 1:L
        [x,y] = find(f>DR(k) & f<=DR(k+1));
        if ~isempty(x)
            for j = 1:length(x)
                f1(x(j),y(j)) = C(k);
            end
        end
        clear x y
    end
    %}
    if b == 3 || b == 5
        %figure,imshow(uint8(f1(60:275,302:585)));% display requantized image
        figure,imshow(uint8(f1))
        title(['Requantized image at ' num2str(b) ' bpp'])
    end
    mse = sum(sum((f-f1).*(f-f1)))/(Height*Width);
    snr(b) = 10*log10(sigmaf*sigmaf/mse);
end
figure,plot(1:B,snr,'k','LineWidth',2)
title('SNR Vs Bits/pixel of a nonuniform quantizer')
xlabel('# bits')
ylabel('SNR(dB)')
```

Table 2.1 Decision regions and output levels for 3- and 4-bit Lloyd–Max quantizers using Barbara image

3 bit		4 bit	
Decision Region	Output Levels	Decision Region	Output Levels
0	31	0	14
40	49	14	27
63	76	32	36
92	107	43	50
127	147	59	68
161	176	78	87
187	198	95	102
206	214	111	119
255		127	135
		144	154
		162	171
		179	187
		194	201
		207	212
		218	224
		230	235
		255	

In the above example, we considered a specific image to be quantized using the nonuniform quantizer. As we will see in a later chapter, residual errors that might be encountered, for instance, in DPCM, all have a characteristic pdf—Laplacian to be precise. Therefore, it is useful and more efficient to design nonuniform quantizers with different number of bits of quantization before using them in the encoder. The following MATLAB code iteratively solves for the optimum input decision and output reconstruction levels for a given number of bits of quantization when the input is a random variable with zero mean and unit variance. Three different pdfs are assumed: Uniform, Gaussian, and Laplacian. Input decision boundaries and output levels are computed for positive values of the input. For negative values of the input, the decision boundaries and output levels are simply the negative of those for positive input values. The MATLAB code plots the output levels against the input decision regions as a staircase. It also plots the count of input sample values belonging to each decision region against the levels as a stem plot. Finally, it also prints the input decision boundaries and corresponding output levels for positive input values. It is understood that $d_{L+1} = \infty$.

```
% QuantizationTable.m
% Design a Nonuniform Quantizer for an input signal
% with known pdf and generate optimum decision
% boundaries and output levels for a specified
% number of bits of quantization
%
```

```
% parameters to be specified:
% Width x Height = size of random image field to be generated
% B = number of bits of quantization
% PDFtype = type of pdf of the signal to be quantized
%   admissible types: 'uniform', 'gaussian' & 'laplacian'
% err = accuracy to terminate iteration
% The input signal is assumed to be zero-mean random variables(rv's)
% decision levels d(-k) = -d(k)
% similarly, output levels c(-k) = -c(k)

clear
close all
Height = 256;
Width = 256;
B = 4; % bits of quantization
PDFtype = 'laplacian'; % options: 'uniform', 'gaussian' & 'laplacian'
%
switch PDFtype
    case 'uniform'
        f = rand(Height,Width)-0.5;% zero-mean uniform pdf
        f = sqrt(12)*f(:);% var = 1
        f = f(logical(f>=0));
        fMax = max(f);
        err = 0.1e-02;
    case 'gaussian'
        f = randn(Height,Width);
        f = f(logical(f>=0));% only positive values of the r.v. are used
        f = f(:);
        fMax = max(f);
        err = 0.1e-03;
    case 'laplacian'
        f = rand(Height,Width);% uniform pdf
        f = (-1/sqrt(2))*log(1-f(:));% transformation for laplacian
        % r.v. with var = 1
        fMax = max(f);
        err = 0.1e-03;
end
%
Lf = length(f);
L = 2^B/2; % number of output levels
% choose an initial set of output levels
if strcmpi(PDFtype,'uniform')
    C = linspace(0,fMax,L);
else
    C = linspace(0,0.7*fMax,L);
end
PartitionRegion = zeros(L,Lf); % partitions consisting of input rv's
RegionCount = zeros(L,1); % number of elements in each partition
RegionSum = zeros(L,1); % sum of rv's in a given partition region
Distortion = zeros(L,1); % minimum distortion in a partition region
TotalDistortion = 1.0e+5; % total minimum distortion
% convergence factor is TD
TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
% start iteration
It = 1; % iteration number
```

```
while TD >= err
    for r = 1:Lf
        d1 = abs(f(r)-C(1));
        J = 1;
        for k = 2:L % choose the nearest neighbor to C(k)
            d2 = abs(f(r)-C(k));
            if d2<d1
                d1 = d2;
                J = k;
            end
        end
        RegionCount(J) = RegionCount(J) + 1;
        RegionSum(J) = RegionSum(J) + f(r);
        PartitionRegion(J,r) = f(r);
        Distortion(J) = Distortion(J)+(f(r)-C(J))*(f(r)-C(J));
    end
    Index = logical(RegionCount == 0);% check for empty regions
    RegionCount(Index) = 1;% set empty regions to count 1
    Distortion = Distortion ./RegionCount;% average minimum distortions
    TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
    C = RegionSum ./RegionCount; % refined output levels
    TotalDistortion = sum(Distortion);% total minimum distortion
    It = It + 1;
end
clear Index
sprintf('Number of iterations = %d; \t bpp = %d',It,B)
% compute optimum decision boundaries from the optimum output levels
DR = zeros(L+1,1);
DR(L+1) = fMax;
for k = 2:L
    DR(k) = (C(k-1)+C(k))/2;
end
% plot output levels against decision regions
figure,stairs(DR(1:L),C,'k','LineWidth',2)
title(['Nonuniform quantizer: ' num2str(B) ' bpp' ' ' PDFtype ' pdf'])
xlabel('Decision Regions')
ylabel('Output Levels')
% decode using the codebook generated
f1 = zeros(size(f)); % quantized output
for r = 1:Lf
    d1 = abs(f(r)-C(1));
    for k = 2:L
        d2 = abs(f(r)-C(k));
        if d2<d1
            d1 = d2;
            J = k;
        end
    end
    f1(r) = C(J);
end
% calculate mse between input and quantized output
mse = sum(Distortion);
sprintf('Number of levels = %d;\t MSE = %5.3f',L,mse)
% plot number of elements in each decision region as stem plot
figure,stem(RegionCount/(It-1),'k')
```

```
title(['Region Count at ' num2str(B) ' bpp' ' ' PDFtype ' pdf'])
xlabel('Level number'), ylabel('Count')
% print decision and output levels
sprintf('Decision boundaries \t Output levels \n')
sprintf('%5.3f\t\t %5.3f\n',DR(1:L),C)
```

2.3.5 Dithering

When an image is quantized coarsely, we notice contouring effects. This happens whether we use a uniform or a nonuniform quantizer, as seen in Figures 2.8b,c and Figures 2.9a,b. One way to mitigate contouring is to add a small amount of *dither* or pseudo random noise to the input image before quantization and then quantize the image [6, 7]. We can then subtract the dither from the quantized image. This is shown in Figure 2.10. The dither perturbs the input by a small amount such that pixels having more or less the same values in a neighborhood fall into different decision regions and are, therefore, assigned slightly different output levels. This eliminates or minimizes the contouring. As an example, the Barbara image is re-quantized to 4 bpp using a uniform dithered quantizer and is shown in Figure 2.11a. The same image quantized to 4 bpp without dithering is shown in Figure 2.11b. We notice marked improvements in using dithering where no contouring is visible. Figure 2.11c corresponds to the image using the dithered quantizer but without subtracting the dither after quantization. We notice some grainy noise in the flat areas, which is due to the addition of dither. However, there is no contouring in the quantized image. Thus, dithering has eliminated contouring without increasing quantization bits. The MATLAB code for quantizing an image with dithering is listed below.

```
% Dithering.m
% Quantize an input image using a uniform quantizer
% with dithering to eliminate contouring effect
% Dither is a uniformly distributed pseudo random noise
% between [-8,8]
clear
close all
%
B = 4; % # bits of quantization
A = imread('barbara.tif');
%A = imread('cameraman.tif');
[Height,Width,Depth] = size(A);
if Depth == 1
    f = double(A);
else
    f = double(A(:,:,1));
end
if Height>512 || Width > 512
    figure,imshow(uint8(f(60:275,302:585)))
    [Counts,Bins] = imhist(uint8(f(60:275,302:585)),256);
else
    figure,imshow(uint8(f))
    [Counts,Bins] = imhist(uint8(f),256);
```

```
end
title('Image quantized to 8 bits')
%
u = 16*(rand(Height,Width)-0.5); % dither or pseudo random noise
f1 = f + u; % noise added to input image
f2 = zeros(size(f));% quantized image without dithering
g = zeros(size(f)); % quantized image with dithering
g1 = zeros(size(f)); % quantized image with dithering with no subtraction
% after quantization
% Design a uniform quantizer with B bpp
L = 2^B;
q = 255/L;
q2 = q/2;
d = linspace(0,255,L+1); % decision regions
r = zeros(L,1); % output levels
for k = 1:L
    r(k) = d(k)+q2;
end
% Quantize input image without dithering
for k = 1:L
    [x,y] = find(f>d(k) & f<=d(k+1));
    for j = 1:length(x)
        f2(x(j),y(j)) = round(r(k));
    end
end
% Quantize input image with dithering
for k = 1:L
    [x,y] = find(f1>d(k) & f1<=d(k+1));
    for j = 1:length(x)
        g(x(j),y(j)) = round(r(k))-u(x(j),y(j));
        g1(x(j),y(j)) = round(r(k));
    end
end
% display quantized image without dithering
if Height>512 || Width > 512
    figure,imshow(uint8(f2(60:275,302:585)))
else
    figure,imshow(uint8(f2))
end
title(['No dithering: Image quantized to ' num2str(B) ' bits'])
% display quantized image with dithering
if Height>512 || Width > 512
    figure,imshow(uint8(g(60:275,302:585)))
else
    figure,imshow(uint8(g))
end
title(['With dithering: Image quantized to ' num2str(B) ' bits'])
if Height>512 || Width > 512
    figure,imshow(uint8(g1(60:275,302:585)))
else
    figure,imshow(uint8(g1))
end
title(['No dither subtraction: Image quantized to ' num2str(B) ' bits'])
```

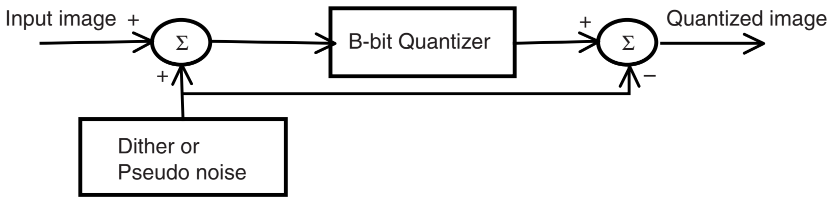


Figure 2.10 Block diagram showing the use of dithering signal in image quantization.

2.4 COLOR IMAGE REPRESENTATION

The discussions in the previous sections pertain to sampling and quantizing monochrome images. A BW image is termed *luminance* or *luma*. On the other hand, a color image has three components corresponding to the red, green, and blue primaries. Typically, all the three components of a color image have the same spatial and grayscale resolutions. They are also highly correlated. Therefore, performing the

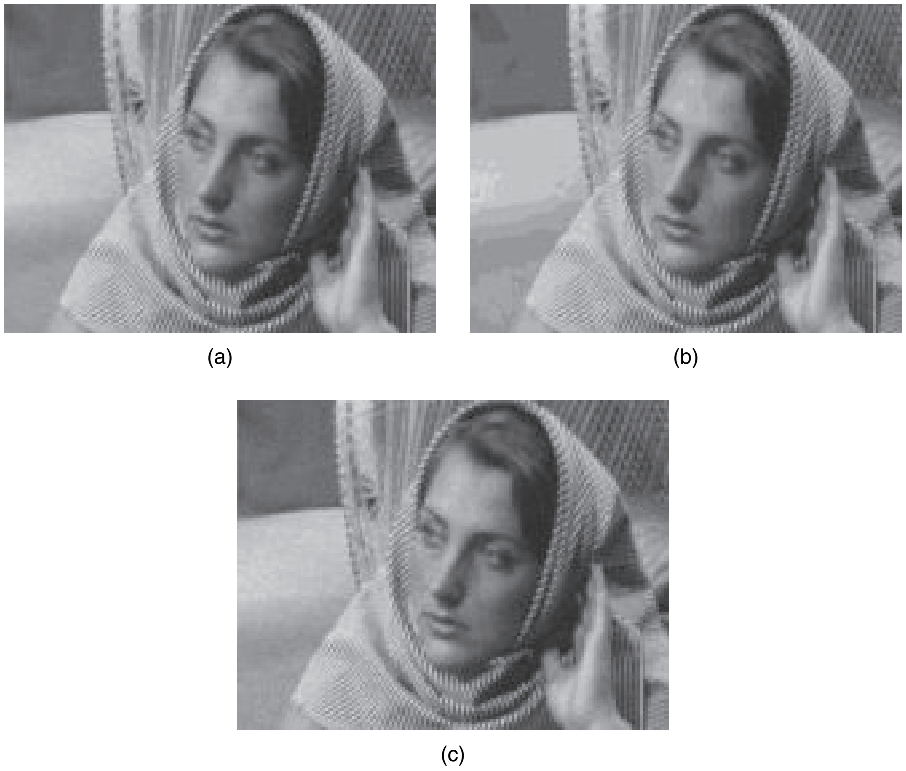


Figure 2.11 An example of a dithered quantization: (a) Barbara image quantized to 4 bpp using a uniform dithered quantizer, (b) same image at 4 bpp using a uniform quantizer without dithering, and (c) same as in (a) but dither is not subtracted after quantization.

same type of processing on each color component will result in either false coloring or poor visual appearance. For instance, if we want to enhance a color image by equalizing the histograms of the components individually, then the so-called color-enhanced image will appear totally different or it may even have very poor visual quality compared with the original image. Again, when we want to compress a color image, we can achieve a higher compression ratio and good visual quality if we use color components that are more or less uncorrelated and perform different compression operations on the components. In order to achieve higher compression or to obtain better visual enhancement, we must transform a color image from its native RGB coordinates into another suitable set of coordinates. The purpose of color coordinate transformation is to render the color image in three orthogonal coordinates so that they can be manipulated individually to obtain optimum results without sacrificing the visual quality. We will briefly describe such invertible transformations in the following.

2.4.1 Gamma Correction

Video display systems such as CRTs (cathode ray tubes) have a nonlinear luminance response Y to input voltage V as given by

$$Y \propto V^{\gamma} \tag{2.44}$$

Typical values for γ are between 2 and 3. The human visual system responds to the incident light roughly as $\frac{1}{3}$ power. Therefore, the net effect of viewing a video displayed on a CRT is a linear relationship to the input signal. However, an image compression system works with images captured directly from real-world scenes and for the decompressed images to be viewed on a CRT by a human to be nearly the same as the captured images, then the captured images must be corrected using the inverse of γ. This is known as *gamma correction*. A video camera, for instance, applies gamma correction before digitizing the video.

2.4.2 YUV and YCbCr Components

The perceived brightness is termed *luma*, while *luminance* refers to the brightness of light intensity. Human visual system has less ability to discern details in color than in luminance information. Therefore, color details can be reduced provided the luminance details are preserved. To achieve higher compression efficiency and visual quality, compression systems convert the input gamma-corrected red, green, and blue components into luma and *chroma* components. As these RGB equivalent components are orthogonal, each component may be manipulated independently to achieve high compression without introducing false coloring in the reconstructed image. This is the reason for the color coordinate transformation.

The luma component, which is the BW equivalent is a weighted sum of the R, G, and B components, and the chroma corresponds to the color differences with luma being absent. Specifically, luma is obtained from the gamma-corrected R', G', and

B' by the linear transformation

$$Y' = 0.299\ R' + 0.587G' + 0.114\ B' \tag{2.45}$$

Equation (2.45) corresponds to the luma used in standard-definition television (SDTV). However, high-definition television (HDTV) uses a slightly different equation for luma as given by

$$Y' = 0.2126R' + 0.7152G' + 0.0722B' \tag{2.46}$$

In both equations (2.45) and (2.46), the values for the gamma-corrected red, green, and blue components lie between 0 and 1. As can be seen from equation (2.46), the green component is given a larger weight in HDTV than in SDTV. In both cases, when all three components have unity values, the luma attains a value of 1, which corresponds to white. Similarly, when all three components are equal to zero, the luma value is also zero and corresponds to black.

The chroma components are obtained from the gamma-corrected red, green, and blue components by subtracting out the luma from blue and red components. The chroma components are denoted U' and V', and for SDTV, are written as

$$U' = B' - Y' = -0.299R' - 0.587G' + 0.886B' \tag{2.47}$$

$$V' = R' - Y' = 0.701R' - 0.587G' - 0.114B' \tag{2.48}$$

In matrix form, equations (2.45), (2.47), and (2.48) can be written as

$$\begin{bmatrix} Y \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.49}$$

Similarly, for HDTV the Y U′ V′ is written in matrix equation as

$$\begin{bmatrix} Y \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.2126 & -0.7152 & 0.9278 \\ 0.7874 & -0.7152 & -0.0722 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.50}$$

To recover the gamma-corrected red, green, and blue components from the luma and chroma components, we use the inverse matrix relation corresponding to equations (2.49) and (2.50), which are given in equations (2.51) and (2.52), respectively, for SDTV and HDTV:

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -0.1942 & -0.5094 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} \tag{2.51}$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -0.1010 & -0.2973 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} \tag{2.52}$$

In MPEG jargon, equations (2.49) and (2.50) and the corresponding inverse transformations in (2.51) and (2.52) are called irreversible because they introduce rounding errors. The standard also specifies reversible color transformation, which is described in equation (2.53) and the corresponding inverse in (2.54).

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} \left\lfloor \dfrac{R' + 2G' + B'}{4} \right\rfloor \\ R' - G' \\ B' - G' \end{bmatrix} \tag{2.53}$$

$$\begin{bmatrix} G' \\ R' \\ B' \end{bmatrix} = \begin{bmatrix} Y' - \left\lfloor \dfrac{U + V}{4} \right\rfloor \\ V + G' \\ U + G' \end{bmatrix} \tag{2.54}$$

When dealing with digital images, the pixel values are between 0 and 255 for 8-bit images. In order to accommodate underflow and overflow that may occur during digital processing such as filtering some foot room and headroom are allowed in the video compression standards. In such cases, the color difference components in digital representation are denoted *Cb* and *Cr*. More precisely, for 8-bit color images an offset of 16 is added to the luma component, which is scaled by 229 so that black level corresponds to 16 and white level to 235. Luma values above 235 are clipped to 235 and values below 16 are clipped to 16. The chroma components are represented in offset binary form by adding 128 to $\frac{112}{0.866}U'$ and $\frac{112}{0.701}V'$ so that *Cb* and *Cr* are in the range 16–240. Thus, Y′CbCr in matrix form for SDTV is expressed as

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.55}$$

Similarly, Y′CbCr for HDTV for 8-bit images is given by

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 46.559 & 156.629 & 15.812 \\ -25.664 & -86.336 & 112 \\ 112 & -101.730 & -10.270 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.56}$$

Corresponding to equations (2.54) and (2.55), the gamma-corrected digital values for the red, green, and blue components can be obtained from Y′CbCr using equation (2.57) for SDTV and (2.58) for HDTV:

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 0.0046 & 0 & 0.0063 \\ 0.0046 & -0.0015 & -0.0032 \\ 0.0046 & 0.0079 & 0 \end{bmatrix} \begin{bmatrix} Y' - 16 \\ Cb - 128 \\ Cr - 128 \end{bmatrix} \tag{2.57}$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 0.0046 & 0 & 0.007 \\ 0.0046 & -0.0008 & -0.0021 \\ 0.0046 & 0.0083 & 0 \end{bmatrix} \begin{bmatrix} Y' - 16 \\ Cb - 128 \\ Cr - 128 \end{bmatrix} \tag{2.58}$$

2.4.3 Video Format

A CRT-based display device, such as a TV, utilizes a left-to-right, top-to-bottom scanning mechanism called *raster scanning*. In North America, a BW image is scanned using a standard adopted by the Radio Electronics Television Manufacturers Association (RETMA). A complete scan of the image is called a *frame*. In SDTV, each frame consists of 525 lines and the video is captured at the rate of 30 fps (actually, at 29.97 fps). The video is *interlaced* to mean that each frame consists of two *fields* called the first field and second field. The image is scanned twice successively to capture the two fields giving a field rate of 60 per second (59.94 per second or 59.94 Hz, to be precise). Each field is made up of 262.5 lines and is offset by half the frame period. In Europe, Asia, and Australia, the field rate is 50 Hz and each field is made up of 312.5 lines to give a total of 625 lines per frame. In practice, a digital video is obtained as follows: A spatio-temporal image is converted into a one-dimensional time-domain signal by scanning the analog video in a raster scan order. In a TV camera, an electron beam scans an electrically charged plate on to which an image is focused, thereby generating a time-domain signal. Corresponding to an interlaced video, each field consists of alternating lines. In order to accommodate the return of the electron beam to the left of the screen as well as to the top of the screen, horizontal and vertical synchronizing pulses are introduced into the time-domain video signal. The time-domain signal is sampled at the appropriate rate satisfying the Nyquist rate, each sample is then converted to an 8-bit (or higher) digital value and recorded as an array of numbers, thus yielding digital video. For more information on temporal sampling and digital conversion, readers are referred to [8].

As opposed to interlaced scanning, *progressive scanning* or noninterlaced scanning is used in digital camcorders, desktop computers, digital TV, and HDTV. In progressive scanning, a frame of image is scanned sequentially from left-to-right and top-to-bottom.

The two systems, namely, the interlaced and progressive scanning, are denoted by the number of lines followed by the letter *i* or *p* followed by the frame rate. For example, the interlaced scanning for SDTV in the North American standard will be denoted *480i29.97*. Note that there are 480 active lines in a frame. Similarly, *625i50* refers to the PAL system.

A color video consists of luma and chroma components. Because the human visual system has a higher sensitivity to luminance than to chrominance, the chroma components are lowpass filtered and subsampled. In digital encoding of color video, the components used are Y', Cb, and Cr. The chroma components Cb and Cr are lowpass filtered and subsampled. This gives rise to three different subsampling formats as described below.

4:4:4 Subsampling

In this subsampling scheme, all three components have the same spatial resolution and represent the maximum spatial and color accuracy. The term 4:4:4 is used to indicate that for every four Y pixels, there are four Cb pixels and four Cr pixels. This scheme is typically used in video editing or digital mastering.

4:2:2 Subsampling

In this scheme, both Cb and Cr components are lowpass filtered in the horizontal direction and subsampled by a factor of 2. Thus, for every four Y pixels, there are two Cb pixels and two Cr pixels. This reduces the raw data of a color image to 66.67%. This scheme is used in MPEG2 for HDTV.

4:2:0 Subsampling

In this scheme, Cb and Cr components are lowpass filtered in horizontal and vertical directions and subsampled by a factor of 2 in both dimensions. Thus, for every four Y pixels, there is one Cb pixel and one Cr pixel. This scheme is used in MPEG2 for SDTV.

2.5 SUMMARY

Digital images are acquired by first sampling analog images in the spatial dimensions at proper sampling rates and then quantizing the samples with sufficient number of bits for better accuracy. These sampling rates are dictated by the Nyquist sampling rates. When the sampling rates are below the Nyquist rates, aliasing distortions occur that are objectionable. We have demonstrated the effect of undersampling in Example 2.1 involving real images where the contouring effect due to aliasing was clearly visible. Practical sampling techniques use pulse of finite spatial extent as opposed to the ideal impulse sampling. The result is the smearing of the image as shown in Example 2.2.

Even though most image acquisition devices are based on sampling using rectangular grids, image sampling using nonrectangular grids are of importance, especially in machine vision and biomedical imaging. We have described, in particular, the linear transformation for hexagonal sampling grid and illustrated it by a couple of examples in this chapter. Example 2.3 shows how to convert pixels from rectangular to hexagonal grids and vice versa, while in Example 2.4 is shown the sampling grid transformation for a real image.

Image quantization follows the sampling process. It approximates the value of an analog sample and represents it in a binary number. An optimal quantizer is one that minimizes the MSE between the input analog image and the output digital image and is called the Lloyd–Max quantizer. Such an optimal quantizer is obtained by solving the transcendental equations (2.31a) and (2.31b). An iterative procedure for solving the transcendental equations called the Lloyd algorithm has been described, and the corresponding MATLAB code has been listed. Both uniform and nonuniform quantizer designs have been explained using Examples 2.5 and 2.6, respectively. We have also developed MATLAB code for the design of Lloyd–Max quantizer for input signals having uniform, Gaussian, and Laplacian pdfs.

One effect of quantizing an image coarsely is the contouring, which occurs when pixels in a neighborhood all have nearly the same value. We showed that dithering is one way of mitigating the contouring effect and demonstrated how the dithered quantizer improves the image quality at 4 bpp (Figure 2.8).

In this chapter, we have also described briefly the color coordinates used in image compression as well as the interlaced and progressive scanning used in NTSC and PAL systems. All the examples are accompanied by the respective MATLAB code that generates all the images and various other numerical outputs.

REFERENCES

1. W. K. Pratt, *Digital Image Processing*, 2nd edition, John Wiley, New York, 1991.
2. R. M. Mersereau, "The processing of hexagonally sampled two dimensional signals," *Proc. IEEE*, 930–949, 1979.
3. S. P. Lloyd, "Least squares quantization in PCM," unpublished Bell Laboratories Technical note. Portions were presented at the Institute of Mathematical Statistics Meeting, Atlantic City, New Jersey, September 1957. Later published in *IEEE Trans. Inf. Theory*, IT-28, 127–135, 1982.
4. J. Max, "Quantizing for minimum distortion," *IEEE Trans. Inf. Theory*, 7–12, 1960.
5. A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer, 1992.
6. N. S. Jayant, *Digital Coding of Waveforms*, Prentice Hall, Englewood Cliffs, NJ, 1984.

7. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
8. A. Netravali and B. G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, New York, 1988.

PROBLEMS

2.1. For an analog image with amplitude in the range 0 - 1 V, design a uniform quantizer with 256 output levels by specifying the decision boundaries and output levels. Calculate the mean square error for the quantizer.

2.2. Consider a random variable X with zero mean and unit variance and the corresponding decision and reconstruction levels $\{d_k\}$ and $\{r_k\}$, respectively. If $Y = \mu + \sigma X$, then show that the decision and reconstruction levels for Y are given, respectively, by $\mu + \sigma d_k$ and $\mu + \sigma r_k$.

2.3. Show that a Laplacian random variable with unit variance can be generated from a uniformly distributed random variable U by using the transformation $-\frac{1}{\sqrt{2}} \ln(U)$.

3 IMAGE TRANSFORMS

3.1 INTRODUCTION

As we have seen in the previous chapter, image acquisition devices capture images in the spatial domain and video as a sequence of images, also in the spatial domain. We have also seen that captured images can be manipulated in the spatial or pixel domain for the purpose of visual enhancement. However, certain image processing operations are more intuitive and efficient if we process the image in a different domain. For instance, digital filters are easy to design and interpret if we adopt the Fourier transform domain. Another interesting aspect of using a frequency domain representation of an image stems from the fact that it is much more efficient to decorrelate an image in the frequency domain than in the spatial domain. This will prove to be very useful in image and video compression. Again, it is more efficient to capture certain features of an image in the frequency domain for pattern classification and identification purposes than in the pixel domain. This is so because of the reduction of dimensions in the frequency domain and the resulting reduction in the computational load. It must be pointed out that Fourier or frequency domain is not the only alternative domain to represent images. In fact, any orthogonal coordinates will be suitable to represent an image. The type of domain to choose depends on the intended application. Moreover, the transformation that we choose must be linear and invertible so that we can recover the image from the transform domain. It should also be orthogonal so that one can manipulate the transformed image independently along the different dimensions.

Our purpose here in using an orthogonal transform to represent an image in the new domain is to be able to achieve a high degree of data compression without sacrificing visual quality. With that in mind, we will study orthogonal and unitary transforms and explore their theoretical limits on achievable compression as well as their implementation issues. More specifically, we will describe Karhunen–Loève transform (KLT), discrete cosine transform (DCT), and so on in detail in this chapter and discrete wavelet transform in the next chapter along with some examples.

3.2 UNITARY TRANSFORMS

For an image of size $N \times N$ pixels, a unitary transform implies that the image can be represented as a linear combination of N^2 *basis* images. These basis images may be independent of the image being transformed as in DCT, or may be computed from the image itself as in KLT. Unitary transforms have very useful properties, especially from the standpoint of image compression and we will concentrate on such transforms. In the following, we will describe both one-dimensional (1D) and two-dimensional (2D) unitary transforms.

3.2.1 1D Unitary Transform

Consider a 1D sequence as a vector $\mathbf{x}^{\mathrm{T}} = [x[n], 0 \le n \le N-1]$. The linear transformation of $\mathbf{x}$ is written as

$$\mathbf{y} = \mathbf{A}\mathbf{x} \quad \text{or} \quad y(k) = \sum_{n=0}^{N-1} a(k,n)x[n], 0 \le k \le N-1 \tag{3.1}$$

The transformation in equation (3.1) is said to be *unitary* if the inverse of the matrix $\mathbf{A}$ is its own conjugate transpose, that is, $\mathbf{A}^{-1} = \mathbf{A}^{*\mathrm{T}}$. We can then recover the original sequence from $\mathbf{y}$ as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} = \mathbf{A}^{*\mathrm{T}}\mathbf{y} \quad \text{or} \quad x[n] = \sum_{k=0}^{N-1} y(k)\, a^{*}(k,n), 0 \le n \le N-1 \tag{3.2}$$

$\mathbf{A}$ is called the *kernel* matrix and its elements may be real or complex. The elements of $\mathbf{y}$ are called the transform coefficients, and the vector $\mathbf{y}$ is known as the transformed vector.

Orthogonal Transform If the elements of a unitary transform matrix $\mathbf{A}$ are real, then $\mathbf{A}$ is called an *orthogonal transform* and its inverse is its own transpose, that is, $\mathbf{A}^{-1} = \mathbf{A}^{\mathrm{T}}$. However, a unitary transform need not be orthogonal. The following example illustrates this statement.

Example 3.1 Let $\mathbf{A}$ be given by

$$\mathbf{A} = \frac{1}{\sqrt{a^2+1}} \begin{bmatrix} a & -j \\ j & -a \end{bmatrix}, 0 \prec a \prec 1$$

Then,

$$\mathbf{A}^{*\mathrm{T}} = \frac{1}{\sqrt{a^2+1}}\begin{bmatrix} a & -j \\ j & -a \end{bmatrix} \text{ and } \mathbf{AA}^{*\mathrm{T}} = \frac{1}{\left(a^2+1\right)}\begin{bmatrix} a^2+1 & 0 \\ 0 & a^2+1 \end{bmatrix} = \mathbf{I}$$

But it is easily seen that

$$\mathbf{AA}^{\mathrm{T}} = \frac{1}{\left(a^2+1\right)}\begin{bmatrix} a^2-1 & 2ja \\ 2ja & a^2-1 \end{bmatrix} \neq \mathbf{I}$$

Therefore, **A** is unitary but not orthogonal.

Basis Vector Equation (3.2) can also be written as

$$\mathbf{x} = \sum_{k=0}^{n-1} y(k)\,\mathbf{a}_k^* \tag{3.3}$$

The vectors $\mathbf{a}_k^* = \left[a^*(k,n), 0 \le n \le N-1\right]^{\mathrm{T}}$, which are the columns of $\mathbf{A}^{*\mathrm{T}}$ are called the *basis* vectors. We can think of the basis vectors as the discrete-time sinusoids at different frequencies and the set of transform coefficients as the unique *spectral signature* of the sequence when the transform is sinusoidal.

1D Discrete Fourier Transform The discrete Fourier transform (DFT) of a sequence $\{x[n], 0 \le n \le N-1\}$ is denoted by $X(k)$ and is define as

$$X(k) = \sum_{n=0}^{N-1} x[n]\,e^{-j(2\pi/N)/nk}, \quad 0 \le k \le N-1 \tag{3.4}$$

In the DSP world, $e^{-j(2\pi)/N}$ is abbreviated by W_N. In terms of W_N, equation (3.4) appears as

$$X(k) = \sum_{n=0}^{N-1} x[n]\,W_N^{nk}, \quad 0 \le k \le N-1 \tag{3.5}$$

The inverse discrete Fourier transform (IDFT) of $X(k)$ gives us the original sequence back and is obtained by multiplying both sides of equation (3.5) by W_N^{-mk} and summing over k:

$$\sum_{k=0}^{N-1} X(k)\,W_N^{-mk} = \sum_{k=0}^{N-1} W_N^{-mk} \sum_{n=0}^{N-1} x[n]\,W_N^{nk} = \sum_{n=0}^{N-1} x[n] \sum_{k=0}^{N-1} W_N^{(n-m)k} \tag{3.6}$$

Since

$$\sum_{k=0}^{N-1} W_N^{(n-m)k} = \begin{cases} N, & \text{for} \quad m = n \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

equation (3.6) reduces to

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X(k)\, W_N^{-nk}, \quad 0 \le n \le N-1 \tag{3.8}$$

The DFT pair as defined in equations (3.5) and (3.8) is not unitary because of the mismatch in the scale factor between the forward and inverse transforms. By incorporating the same scale factor $1/\sqrt{N}$ in the forward and inverse transforms, we may write the DFT pair as

$$X(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n]\, W_N^{nk}, \quad 0 \le k \le N-1 \tag{3.9}$$

$$x[n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k)\, W_N^{-kn}, \quad 0 \le n \le N-1 \tag{3.10}$$

The DFT pair given by equations (3.9) and (3.10) is unitary because it is easily verified that $\{(1/\sqrt{N})W_N^{nk}\} \times \{(1/\sqrt{N})W_N^{-nk}\} = \mathbf{I}$.

Efficient Implementation of 1D DFT The 1D DFT of an N-point sequence can be obtained in $\frac{N}{2}\log_2(N)$ multiply–add operations. One such fast algorithm is the most familiar *fast Fourier transform* (FFT) [1,2]. There are two implementations of the FFT algorithm called *decimation in time* and *decimation in frequency*, both of which use the divide and conquer rule. Let us look at the decimation in time algorithm.

Consider the DFT of a sequence as defined in equation (3.5). Let the length of the sequence be a power of two, that is, $N = 2^m$, where m is a positive integer. By dividing the input sequence into even and odd indexed sequences, we can rewrite equation (3.5) as

$$X(k) = \sum_{n=0}^{(N/2)-1} x[2n]\, W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x[2n+1]\, W_N^{(2n+1)k} \tag{3.11}$$

Now, $W_N^{2nk} = e^{-j(2\pi/N)2nk} = e^{-j[2\pi/(N/2)]nk} = W_{N/2}^{nk}$ and $W_N^{(2n+1)k} = W_{N/2}^{nk} W_N^k$. Therefore, equation (3.11) becomes

$$X(k) = \sum_{n=0}^{(N/2)-1} x[2n]\, W_{N/2}^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} x[2n+1]\, W_{N/2}^{nk} \tag{3.12}$$

However, $W_N^{k+(N/2)} = W_N^k e^{-j(2\pi/N)N/2} = -W_N^k$. Therefore, the N-point DFT is given in terms of two $N/2$-point DFTs as

$$X(k) = X_1(k) + W_N^k X_2(k) \tag{3.13a}$$

$$X\left(k + \frac{N}{2}\right) = X_1(k) - W_N^k X_2(k) \tag{3.13b}$$

where we have used the fact that

$$X_1(k) = \sum_{n=0}^{(N/2)-1} x[2n]\, W_{N/2}^{nk} \tag{3.14a}$$

$$X_2(k) = \sum_{n=0}^{(N/2)-1} x[2n+1]\, W_{N/2}^{nk} \tag{3.14b}$$

Next, the two $\frac{N}{2}$-point sequences each can be divided into even and odd indexed sequences and the corresponding DFTs can be computed in terms of two $\frac{N}{4}$-point sequences. One can continue to divide the input sequences into even and odd indexed sequences and compute the corresponding DFTs until left with 2 points. The DFT of a 2-point sequence is simply the sum and difference of the inputs. An example of implementing an 8-point DFT using the decimation in time FFT algorithm is illustrated in a signal flow diagram and is shown in Figure 3.1. Figure 3.1a shows the 8-point DFT in terms of two 4-point DFTs corresponding to even and odd indexed sequences, Figure 3.1b shows the 4-point DFTs in terms of 2-point DFTs, and finally, Figure 3.1c shows the complete 8-point DFT. It is interesting to note that the ordering of the input sequence can be obtained by reversing the binary digits of the original sequence order. This is known as *bit reversal*. There are $\log_2\left(2^3\right) = 3$ stages of computation and each stage has $\frac{N}{2} = 4$ complex multiplies. Thus, the total number of multiplies to compute an 8-point DFT using the decimation in time FFT equals $\frac{N}{2}(\log_2 N) = 12$. Compare this with the brute force method, which will require 64 multiplies. Thus, FFT is an efficient algorithm to compute the DFT of a sequence. However, the DFT is not a popular transform for image compression because it is not optimal in decorrelating an image and because it is not purely real.

Similar to the decimation in time algorithm, one can decimate the DFT coefficients into even and odd numbered points and continue the process until left with

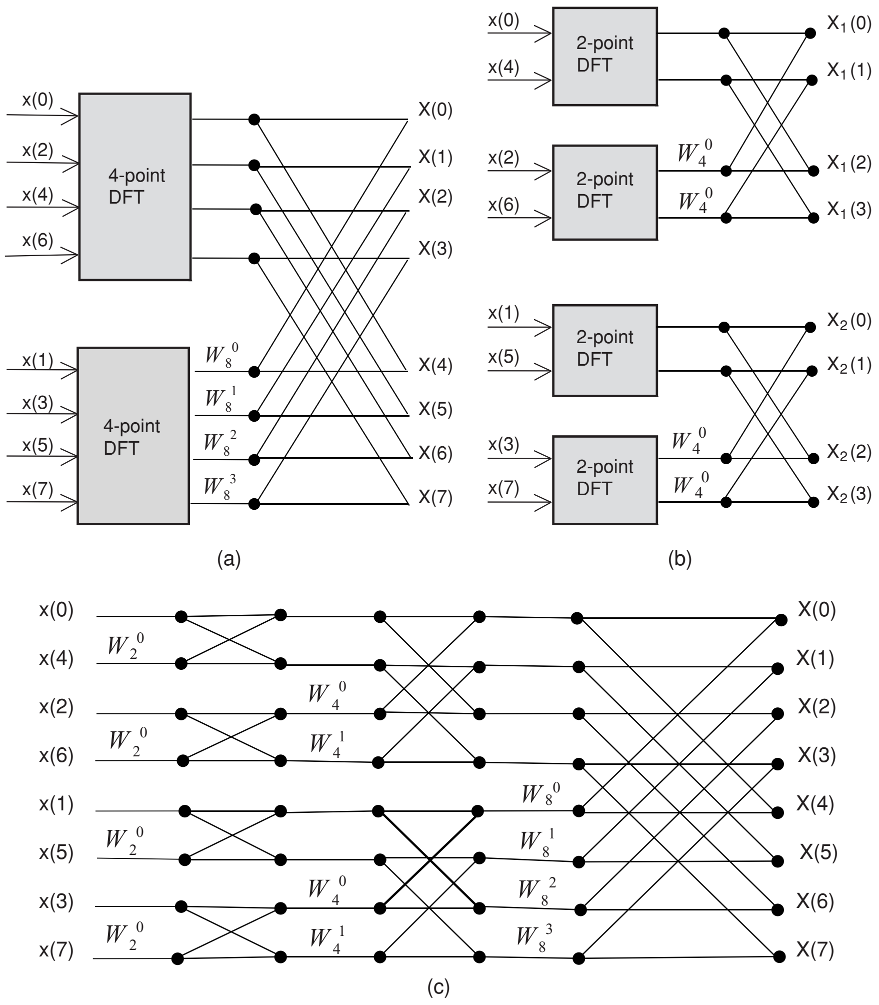


Figure 3.1 Signal flowgraph of an 8-point decimation in time FFT algorithm: (a) 8-point DFT in terms of two 4-point DFTs, (b) 4-point DFTs in terms of 2-point DFTs, and (c) Complete 8-point DFT.

2 points. This algorithm is equivalent to the decimation in time counterpart in terms of the computational load.

1D Discrete Cosine Transform As we will see later, the DCT is much more efficient than the DFT in decorrelating the pixels and so is very useful as a compression vehicle [3]. The DCT of a sequence $\{x[n], 0 \le n \le N-1\}$ is defined as

$$X(k) = \alpha(k) \sum_{n=0}^{N-1} x[n] \cos\left[\frac{(2n+1)\pi k}{2N}\right], \quad 0 \le k \le N-1 \tag{3.15}$$

where

$$\alpha(k) = \begin{cases} \dfrac{1}{\sqrt{N}}, & \text{for} \quad k = 0 \\ \sqrt{\dfrac{2}{N}}, & \text{for} \quad 1 \le k \le N-1 \end{cases} \tag{3.16}$$

The elements of the 1D DCT kernel matrix are given by

$$A_{\mathrm{DCT}}(k,n) = \begin{cases} \dfrac{1}{\sqrt{N}}, k = 0; & 0 \le n \le N-1 \\ \sqrt{\dfrac{2}{N}} \cos\left[\dfrac{(2n+1)\pi k}{2N}\right], & 1 \le k \le N-1; \quad 0 \le n \le N-1 \end{cases} \tag{3.17}$$

Because the 1D DCT is orthogonal, the sequence $\{x[n]\}$ can be recovered from

$$x[n] = \sum_{k=0}^{N-1} \alpha(k) X(k) \cos\left[\frac{(2n+1)\pi k}{2N}\right], \quad 0 \le n \le N-1 \tag{3.18}$$

Fast DCT. Like 1D DFT, the 1D DCT is also a fast algorithm implying that an N-point DCT can be computed with $\frac{N}{2}(\log_2(N))$ multiplications [4–6]. Along the same lines as the DFT, we can use the divide and conquer rule to compute the DCT in stages with each stage involving $\frac{N}{2}$ multiplications. The number of stages to compute an N-point DCT will be $\log_2(N)$. A signal flowgraph of an 8-point DCT is shown in Figure 3.2. The corresponding flowgraph to compute the inverse discrete cosine transform (IDCT) is shown in Figure 3.3.

DCT can also be computed using the FFT algorithm. It can be shown that the DCT of an N-point sequence $x[n]$ is given by

$$X(k) = \mathrm{Re}\left\{\alpha(k)\, W_{2N}^{K/2} \mathrm{DFT}(\tilde{x}[n])\right\}$$

where

$$\begin{aligned} \tilde{x}[n] &= x[2n], & 0 \le n \le \frac{N}{2} - 1 \\ \tilde{x}[N-1-n] &= x[2n+1], & 0 \le n \le \frac{N}{2} - 1 \end{aligned}$$

Similarly, the IDCT of $\{X(k)\}$ can be obtained from its IDFT as

$$\begin{aligned} \tilde{x}[2n] = x[2n] &= \mathrm{Re}\left\{\mathrm{IDFT}\left(\alpha(k) X(k) W_{2N}^{-k/2}\right)\right\}, & 0 \le n \le \frac{N}{2} \\ x[2n+1] &= \tilde{x}[N-1-n], & 0 \le n \le \frac{N}{2} - 1 \end{aligned}$$

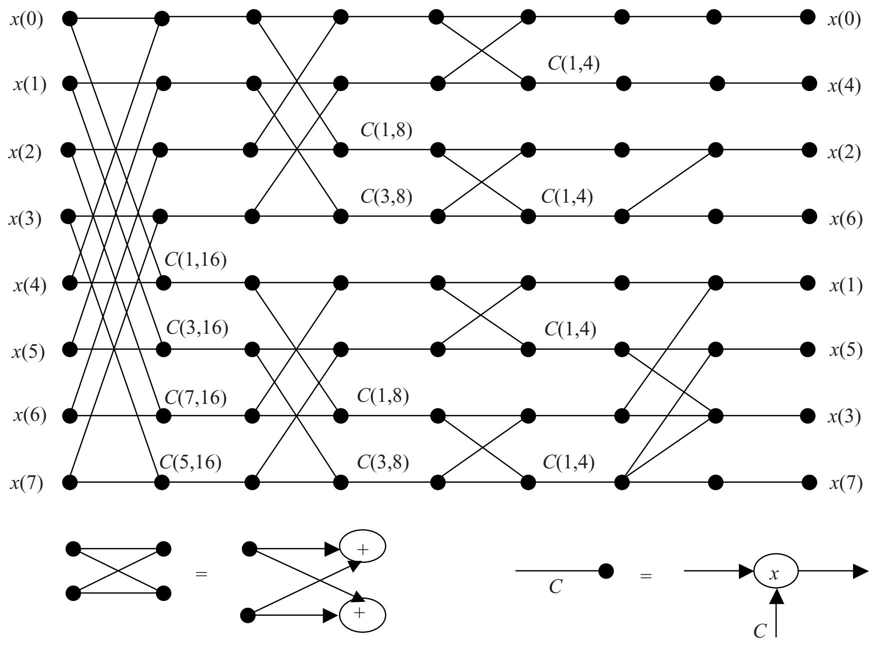
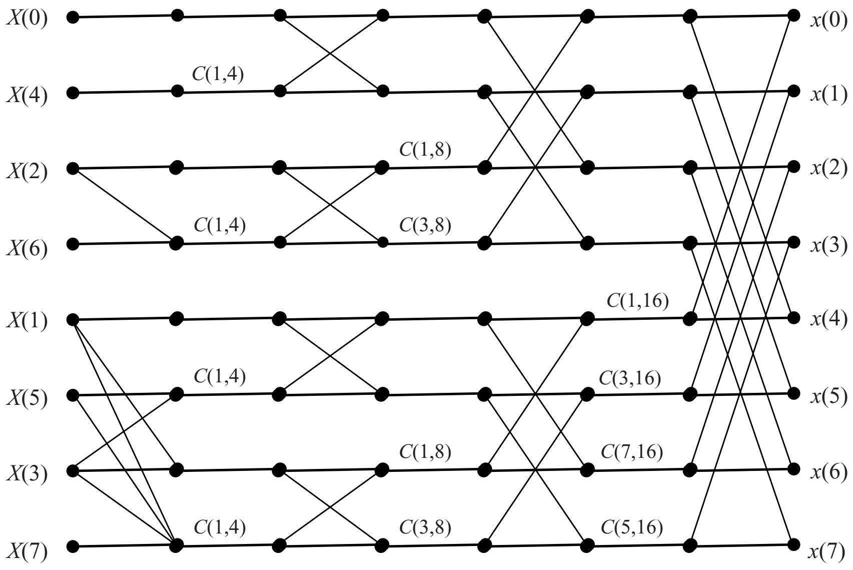


Figure 3.2 Signal flowgraph of an 8-point fast DCT using decimation in frequency algorithm.



$$C(i,\ j) = \frac{1}{2\cos(\pi\ i/j)}$$

Figure 3.3 Signal flowgraph of 8-point IDCT corresponding to Figure 3.2.

A listing of MATLAB program called FastDCTImage to compute the 2D DCT of an image using FFT is shown below. It calls two functions, namely, FastDCT and FastIDCT, to compute the 2D DCT and IDCT of an $N \times N$ block of pixels.

```
% FastDCTImage.m
% computes the N x N DCT of an input image
% using FFT algorithm

A = imread('cameraman.tif');
x = double(A(:,:,1));% strip one of three components
%{
% for cropped image
Height = 256; Width = 256;
A = imread('barbara.tif');
x = double(A(59:58+Height,331:330+Width,1));
%}
figure,imshow(x,[]), title('original image')
% compute the 2D DCT of the image in blocks of 8 x 8 pixels
% by calling MATLAB function blkproc
% FastDCT computes the 2D DCT of an 8 x 8 block
% Y will be of the same size as the image x
Y = blkproc(x,[8 8],@FastDCT);
%
% now compute the inverse 2D DCT by calling again the
% MATLAB function blkproc with FastIDCT, which
% calculates the 2D inverse DCT
z = blkproc(Y,[8 8],@FastIDCT);
figure,imshow(uint8(z)),title('inverse DCT of the image')
%
function y = FastDCT(x)
% y = FastDCT(x)
% computes the dct of an N x N block x
% using decimation in time FFT algorithm
% x is an N x N matrix and y is the dct of x
% N is assumed to be of the type 2^m, m is a
% positive integer

N = size(x,1);
x1 = zeros(N,N);
y = zeros(N,N);
% obtain x1 from x: x1(n) = x(2n), 0 <= n <= N/2;
% x1(N-1-n)= x(2n+1), 0<= n <= N/2
x1(:,1:N/2) = x(:,1:2:N);
x1(:,N+1-(1:N/2))= x(:,2:2:N);
% compute the DFT of each row of x1: X1(k) = sum(x1(n)*WN^nk);
for m = 1:N
    x1(m,:) = fft(x1(m,:));
end
```

```
%multiply DFT by alpha(k) and W2N^(k/2) and take the real part
% alpha(0) = 1/sqrt(N) and alpha(2:N) = sqrt(2/N);
% W2N = exp(-i*2*pi/(2*N))
% MATLAB executes complex function faster if you use 1i
% instead of i
% where i = sqrt(-1).
a(1) = 1/sqrt(N);
a(2:N) = sqrt(2/N);
W2N = exp(-1i*2*pi/(2*N));
for m = 1:N
    y(m,1:N) = real(a.*W2N .^((0:N-1)/2).*x1(m,1:N));
end
% Now compute the DCT along columns which is the same
% as computing the DCT of y' along rows
y = y';
x1(:,1:N/2) = y(:,1:2:N);
x1(:,N+1-(1:N/2))= y(:,2:2:N);
for m = 1:N
    x1(m,:) = fft(x1(m,:));
end
% Again, multiply the dft by alpha(k) and W2N^(k/2)
for m = 1:N
    y(m,1:N) = real(a.*W2N .^((0:N-1)/2).*x1(m,1:N));
end
% transpose y to get the dct in the same order as input
% pixel block x
y = y';

 %

 function y = FastIDCT(x)
% y = FastIDCT(x)
% computes the idct of an N x N block
% using decimation in time IFFT
% N must be a power of 2, i.e., N = 2^m, where
% m is a positive integer

N = size(x,1);
x3 = zeros(N,N);
% multiply x by a(k)*W2N^(-k/2) & compute the idft of each row
W = exp(1i*pi/(2*N));
a(1) = 1/sqrt(N);
a(2:N) = sqrt(2/N);
for m = 1:N
    x(m,1:N) = a.*x(m,1:N).*W .^(0:N-1);
    x3(m,:) = ifft(x(m,:));
end
% multiply ifft(x) by N because ifft includes 1/N factor
x3 = x3 * N;
```

```
% reorder the output sequence to get the correct output
% sequence
y(:,1:2:N) = x3(:,1:N/2);
y(:,2:2:N)= x3(:,N:-1:N/2+1);
y = real(y);
%
% multiply y by a(k)*W2N^(-k/2) & compute the idft of each
% column which is the same thing as computing idft
% of y' along rows
y = y';
for m = 1:N
    x3(m,1:N) = a.*y(m,1:N).*W .^(0:N-1);
    x3(m,:) = ifft(x3(m,:));
end
x3 = x3 * N;
% reorder the output sequence to get the correct output
% sequence
y(:,1:2:N) = x3(:,1:N/2);
y(:,2:2:N)= x3(:,N:-1:N/2+1);
% even though y is supposed to be real, the imaginary parts
% of y will be zero and y will appear as complex.
% so, take the real part of y' to get the true IDCT
% remember that we transposed y in calculating the IDCT
% along columns. So, we have to transpose it to get the
% image in the normal order.
y = real(y');
```

1D Discrete Sine Transform For an N-point sequence$\{x[n], 0 \le n \le N-1\}$, the forward and inverse *discrete sine transform* (DST) pair is given by

$$X(k) = \sqrt{\frac{2}{N+1}} \sum_{n=0}^{N-1} x[n] \sin\left[\frac{(n+1)(k+1)\pi}{N+1}\right], \quad 0 \le k \le N-1 \tag{3.19}$$

$$x[n] = \sqrt{\frac{2}{N+1}} \sum_{k=0}^{N-1} X(k) \sin\left[\frac{(n+1)(k+1)\pi}{N+1}\right], \quad 0 \le n \le N-1 \tag{3.20}$$

Similar to the 1D DCT, the elements of the kernel matrix of DST are expressed as

$$A_{\mathrm{DST}}(k,n) = \sqrt{\frac{2}{N+1}} \sin\left(\frac{(n+1)(k+1)\pi}{N+1}\right), \quad 0 \le n,k \le N-1 \tag{3.21}$$

1D Discrete Hartley Transform The discrete Hartley transform [7, 8] uses both cosine and sine functions in its kernel matrix and is defined for a sequence

$\{x[n], 0 \le n \le N-1\}$ as

$$X(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] \operatorname{cas}\left(\frac{2\pi nk}{N}\right), \quad 0 \le k \le N-1 \tag{3.22}$$

The inverse discrete Hartley transform is given by

$$x[n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k) \operatorname{cas}\left(\frac{2\pi nk}{N}\right), \quad 0 \le n \le N-1 \tag{3.23}$$

The elements of the discrete Hartley transform matrix are written as

$$A_{\text{Hartley}}(k,n) = \frac{1}{\sqrt{N}} \operatorname{cas}\left(\frac{2\pi nk}{N}\right), \quad 0 \le n,k \le N-1 \tag{3.24}$$

In equations (3.22) through (3.24), the function cas stands for

$$\operatorname{cas}(\theta) = \cos(\theta) + \sin(\theta) \tag{3.25}$$

1D Hadamard, Haar, and Slant Transforms The discrete unitary transforms defined above all have sinusoidal basis functions. But Hadamard, Haar, and Slant transforms all use rectangular or *sequency* basis functions. We will define them in the following.

1D Hadamard Transform. For a 2-point sequence, the 2×2 kernel matrix of Hadamard transform [9] is given by

$$\mathbf{A}_{\text{H}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.26}$$

A useful property of Hadamard transform is that the higher order matrices can be generated recursively from the lower order matrices. For example, given an $N \times N$ Hadamard matrix, the $2N \times 2N$ Hadamard matrix is obtained from

$$\mathbf{A}_{\text{H}2N} = \left[\begin{array}{c|c} \mathbf{A}_{\text{H}N} & \mathbf{A}_{\text{H}N} \\ \hline \mathbf{A}_{\text{H}N} & -\mathbf{A}_{\text{H}N} \end{array}\right] \tag{3.27}$$

For example, the 4 × 4 Hadamard matrix can be obtained from equations (3.26) and (3.27) as

$$\mathbf{A}_{\mathrm{H4}} = \left[\begin{array}{c|c} \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ \hline \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & -\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{array}\right] = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{3.28}$$

1D Haar Transform. The kernel matrices corresponding to the 4 × 4 and 8 × 8 Haar transform [10, 11] are given by

$$\mathbf{A}_{\mathrm{Haar}4\times4} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix} \tag{3.29}$$

$$\mathbf{A}_{\mathrm{Haar}8\times8} = \frac{1}{\sqrt{8}}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix} \tag{3.30}$$

1D Slant Transform. For $N = 2$, the kernel matrix of the Slant transform is identical to that of the Hadamard transform. For $N = 4$, the Slant transform matrix is described by the following equation [12]:

$$\mathbf{A}_{\mathrm{Slant}4\times4} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 & 1 & 1 & 1 \\ \frac{3}{\sqrt{5}} & \frac{1}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & -\frac{3}{\sqrt{5}} \\ 1 & -1 & -1 & 1 \\ \frac{1}{\sqrt{5}} & -\frac{3}{\sqrt{5}} & \frac{3}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \end{bmatrix} \tag{3.31}$$

There exists a recursive relation to generate higher order Slant matrices from the lower order ones. The recursive relation to generate order N kernel matrix of the

Slant transform from order $N/2$ is described by the following equation:

$$\mathbf{A}_{\text{Slant}N} = \frac{1}{\sqrt{2}} \begin{bmatrix} \begin{matrix} 1 & 0 \\ a_N & b_N \end{matrix} & \mathbf{0} & \begin{matrix} 1 & 0 \\ -a_N & b_N \end{matrix} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{(N-4)/2} & \mathbf{0} & \mathbf{I}_{(N-4)/2} \\ \begin{matrix} 0 & 1 \\ -b_N & a_N \end{matrix} & \mathbf{0} & \begin{matrix} 0 & -1 \\ b_N & a_N \end{matrix} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{(N-4)/2} & \mathbf{0} & -\mathbf{I}_{(N-4)/2} \end{bmatrix} \left[\begin{array}{c|c} \mathbf{A}_{\text{Slant}N/2} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{A}_{\text{Slant}N/2} \end{array}\right] \tag{3.32}$$

In equation (3.32), I_M is an $M \times M$ identity matrix and the constants a_N and b_N are defined by

$$a_{2N} = \sqrt{\frac{3N^2}{4N^2 - 1}} \tag{3.33}$$

$$b_{2N} = \sqrt{\frac{N^2 - 1}{4N^2 - 1}} \tag{3.34}$$

3.2.2 2D Unitary Transform

If we write an $N \times N$ image as a vector $\mathbf{x}$ of N^2 elements, then its 2D linear transform can be written in matrix form as

$$\mathbf{y} = \mathbf{Ax} \tag{3.34}$$

where $\mathbf{y}$ is a vector of N^2 elements and the transform matrix $\mathbf{A}$ is of size $N^2 \times N^2$. In expanded form, equation (3.34) is expressed by

$$Y(k,l) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x[m,n]\, A(k,m;l,n),\ 0 \le k,l \le N-1 \tag{3.35}$$

If the 2D transform $\mathbf{A}$ is orthogonal, then the inverse transform is defined by

$$x[m,n] = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} Y(k,l)\, A(k,m;l,n),\ 0 \le m,n \le N-1 \tag{3.36}$$

2D Separable Transform. If the 2D transform matrix $\mathbf{A}$ can be expressed as the product of two matrices

$$A(k,m; \quad l,n) = A_1(k,m)\, A_2(l,n) \tag{3.37}$$

then the 2D transform $\mathbf{A}$ is said to be separable. When the 2D transform is separable, the transformed image can be computed using two 1D transforms. To prove this statement, substitute equation (3.37) in (3.35) and write the transformed image **y** as

$$\begin{aligned} Y(k,l) &= \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x[m,n]\,A_1(k,m)\,A_2(l,n) \\ &= \sum_{m=0}^{N-1}\left\{\sum_{n=0}^{N-1} x[m,n]\,A_2(l,n)\right\} A_1(k,m) \end{aligned} \tag{3.38}$$

In equation (3.38), the quantity within the bracket is the transform of the image along the columns, that is, that the transform of each row is computed. Denoting it $x_c(k,m)$, we can rewrite equation (3.38) as

$$Y(k,l) = \sum_{m=0}^{N-1} x_c(k,m)\,A_1(k,m), \quad 0 \le k \le N-1 \tag{3.39}$$

where $x_c(k,m)$ is given by

$$x_c(k,m) = \sum_{n=0}^{N-1} x[m,n]\,A_2(l,n), \quad 0 \le l \le N-1 \tag{3.40}$$

Thus, we first calculate the 1D transform of the image along the columns as given by equation (3.40) and then calculate the 1D transform along the rows according to equation (3.39) to obtain the true 2D transformed image **y**. This is called *row–column* operation. In typical image processing, the 2D transform is separable and the two 1D transforms are identical as well. Moreover, we are interested in unitary transforms. In such cases, the 2D transformed image can be expressed as

$$\mathbf{y} = \mathbf{AxA} \tag{3.41}$$

The inverse transform takes a similar form as

$$\mathbf{x} = \mathbf{A}^{*\mathrm{T}}\mathbf{y}\mathbf{A}^{*\mathrm{T}} \tag{3.42}$$

All of the unitary transforms discussed so far are separable and can, therefore, be implemented in a row–column operation.

Basis Image. Consider the image expressed in terms of its 2D transform coefficients as given in equation (3.36). Corresponding to the unitary transform **A**, let the vector $\mathbf{a}_k^*$ denotes the *k*th column of $\mathbf{A}^{*\mathrm{T}}$. Define the $N \times N$ matrices

$$\mathbf{A}_{k,l}^* = \mathbf{a}_k^*\mathbf{a}_l^{*\mathrm{T}}, \quad 0 \le k,l \le N-1 \tag{3.43}$$

We can then write equation (3.36) as

$$x[m,n] = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} Y(k,l)\,\mathbf{A}_{k,l}^{*} \tag{3.44}$$

which is an expansion of the $N \times N$ image as a weighted sum of the *basis* images $\mathbf{A}_{k,l}^{*}$. In equation (3.44), the weights are the 2D transform coefficients as computed from equation (3.35). Similar to the 1D case, the weights in equation (3.44) represent the unique signature of the image in the transform domain. The basis images are unique to the given unitary transform. Let us consider an example to elucidate the idea.

Example 3.2 Expand the 2×2 image $\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ in terms of the basis images corresponding to the DCT.

Solution From equation (3.17), the DCT kernel matrix is found to be $\mathbf{A} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. The transform coefficients are found from $\mathbf{y} = \mathbf{AxA} = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$. Next, we compute the four basis images as follows:

$$T_{00} = \begin{bmatrix} a_{00} \\ a_{10} \end{bmatrix} \times \begin{bmatrix} a_{00} & a_{10} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad T_{01} = \begin{bmatrix} a_{00} \\ a_{10} \end{bmatrix} \times \begin{bmatrix} a_{01} & a_{11} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix},$$

$$T_{10} = \begin{bmatrix} a_{01} \\ a_{11} \end{bmatrix} \times \begin{bmatrix} a_{00} & a_{10} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}, \quad T_{11} = \begin{bmatrix} a_{01} \\ a_{11} \end{bmatrix} \times \begin{bmatrix} a_{01} & a_{11} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

Next, we calculate the weighted sum of the basis images to obtain the input image. Thus,

$$y(0,0)\,T_{00} + y(0,1)\,T_{01} + y(1,0)\,T_{10} + y(1,1)\,T_{11} = 3T_{00} - T_{11} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} = \mathbf{x}$$

Example 3.3 Compute and display the 8×8 basis images for the 2D unitary transforms DCT, DST, Hartley, Haar, Hadamard, and Slant.

Solution Referring to the transforms described in the preceding discussion, the MATLAB code to compute and display the basis images is listed below. The basis images for DCT, DST, Hartley, Haar, Hadamard, and Slant transforms are shown in Figures 3.4a–f, respectively, for a transform size of 8×8. For good visibility sake, each 8×8 basis image is magnified by a factor of 2 in both dimensions and its values

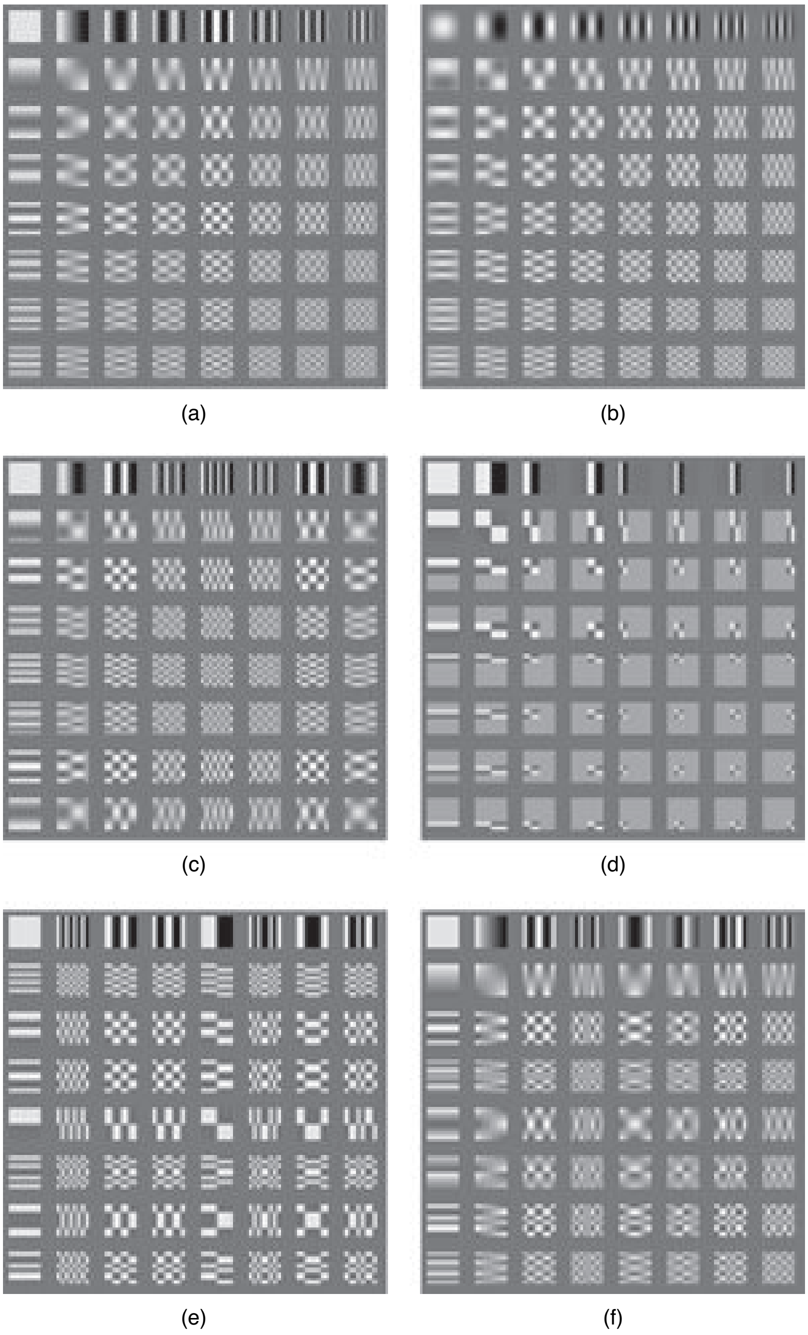


Figure 3.4 Basis images for some important unitary transforms: (a) DCT, (b) DST, (c) discrete Hartley transform, (d) discrete Haar transform, (e) discrete Hadamard transform, and (f) discrete Slant transform.

are scaled to between 0 and 255. The background value is set to 16. The MATLAB code for Example 3.3 is listed below.

```
% Example3_3.m
% Computes the basis images corresponding to a given
% 2D unitary transform and size
% and displays as images
%
% Allowed transforms = 'DCT', 'DST', 'Hartley', 'Haar',
% 'Hadamard', and 'Slant'
%
N = 8; % N x N is the size of the unitary transform
% Only N = 8 is allowed for Haar transform
TransformType = 'DCT';
% compute the basis images by calling the
% function BasisImages
[y,T] = BasisImages(N,TransformType); % T is the N^2 x N^2
                                      % basis images
%
X = ones(192,192) * 16; % inscribe each N x N submatrix into X
for k = 0:N-1
    rIndx = k*24 + 1;
    for l = 0:N-1
        cIndx = l*24 +1;
        MaxT = max(max(T(k*N+1:k*N+N,l*N+1:l*N+N)));
        MinT = min(min(T(k*N+1:k*N+N,l*N+1:l*N+N)));
        % normalize to between 0 & 255 for display purpose
        if k == 0
            t1 = T(k*N+1:k*N+N,l*N+1:l*N+N)/MaxT*255;
        else
            t1 = (T(k*N+1:k*N+N,l*N+1:l*N+N)-MinT)/
                 (MaxT-MinT)*255;
        end
        T1 = imresize(t1,[16 16],'bicubic'); % exapnd matrix
                                             % to 2N x 2N
        X(rIndx+3:rIndx+18,cIndx+3:cIndx+18) = T1;
    end
end
figure,imshow(X,[]),title([num2str(N) 'x' num2str(N) ' basis
images for ' TransformType]);
```

```
function [y,T] = BasisImages(N,Type)
% Returns the N x N transform coefficients and
% N^2 x N^2 basis images for a given
% 2D unitary transform of size N x N
%
```

```
% Input:
% N x N = size of the transform
% For Haar transform, only N = 8 is allowed
% Type = unitary transform type
% Permissible transforms are:
% DCT, DST, Hartley, Haar, Hadamard, and Slant
%
y = zeros(N,N);
T = zeros(N*N,N*N);
switch Type
    case 'DCT'
        for k = 0:N-1
            y(k+1,:) = cos((2*(0:N-1)+1)*pi*k/(2*N));
        end
        y(1,:) = y(1,:)/sqrt(N);
        y(2:N,:) = y(2:N,:) * sqrt(2/N);
    case 'DST'
        for k = 0:N-1
            y(k+1,:) = sin(((0:N-1)+1)*pi*(k+1)/(N+1));
        end
        y = y*sqrt(2/(N+1));
    case 'Hartley'
        for k = 0:N-1
            y(k+1,:) = cos((0:N-1)*2*pi*k/N) + sin((0:N-1)*2*pi*k/N);
        end
        y = y*sqrt(1/N);
    case 'Haar'
        sq2 = sqrt(2);
        y = [1 1 1 1 1 1 1 1; 1 1 1 1 -1 -1 -1 -1;...
            sq2 sq2 -sq2 -sq2 0 0 0 0; 0 0 0 0 sq2
            sq2 -sq2 -sq2;...
            2 -2 0 0 0 0 0 0; 0 0 2 -2 0 0 0 0;...
            0 0 0 0 2 -2 0 0; 0 0 0 0 0 0 2 -2];
        y = y/sqrt(8);
    case 'Hadamard'
        y = [1 1; 1 -1];
        y = y/sqrt(2);
        n = 2;
        while n < N
            n = 2*n;
            y = [y y; y -y];
            y = y/sqrt(2);
        end
    case 'Slant'
        a8 = sqrt(16/21);
        b8 = sqrt(5/21);
        s4 = [1 1 1 1; 3/sqrt(5) 1/sqrt(5) -1/sqrt(5) -3/sqrt(5);...
            1 -1 -1 1; 1/sqrt(5) -3/sqrt(5) 3/sqrt(5) -1/sqrt(5)];
        s4 = s4/2;
        h1 = [1 0 0 0 1 0 0 0];
        h2 = [a8 b8 0 0 -a8 b8 0 0];
        h3 = [0 0 1 0 0 0 1 0];
        h4 = [0 0 0 1 0 0 0 1];
        h5 = [0 1 0 0 0 -1 0 0];
        h6 = [-b8 a8 0 0 b8 a8 0 0];
        h7 = [0 0 1 0 0 0 -1 0];
        h8 = [0 0 0 1 0 0 0 -1];
        y = [h1;h2;h3;h4;h5;h6;h7;h8]*[s4 zeros(4,4);...
            zeros(4,4) s4]/sqrt(2);
end
%
y1 = zeros(N,N);
if ~strcmpi(Type,'hadamard')
    y1 = y'; % Transpose the transform matrix except for Hadamard
else
    y1 = y;
end
for k = 0:N-1
    rInd = k*N + 1;
    for l = 0:N-1
        cInd = l*N + 1;
        T(rInd:rInd+N-1,cInd:cInd+N-1) = y1(:,k+1) * y1(:,l+1)';
    end
end
```

Example 3.4 Read an intensity image and expand it in terms of the DCT basis images and show how the image quality improves as more and more basis images are included in the expansion. Also, plot the mean square error (MSE) in reconstructing the image using from one to $N \times N$ basis images. Take N to be 8.

Solution As we did in the previous example, we first calculate the 64, 8×8 basis images corresponding to the 8×8 DCT. Next, we read in the Barbara image, compute the 2D DCT of each 8×8 block using the raster scanning pattern, and expand each such block in terms of the computed basis images. Once we reconstruct the entire image using the basis images, we then calculate the MSE between the original image and the reconstructed image. To visualize the difference in the quality of the reconstructed image, we use 1 and 13 basis images in the reconstruction. Figure 3.5a is the image reconstructed with just one basis image (the first one). Since the first basis image is proportional to the average value of a block of pixels (see Figure 3.4a), it is no surprise that the image looks all blocky. When we use the first 13 basis images (all 8 in the first row and the first 5 in the second row) in the reconstruction, the image quality is much better, as shown in Figure 3.5b. When all the 64 basis images are used, the reconstruction is perfect and the resulting MSE is zero. Figure 3.5c is the reconstructed image with all 64 basis images. The MSE due to partial reconstruction is shown in Figure 3.5d, where we see that the MSE decreases rapidly as the

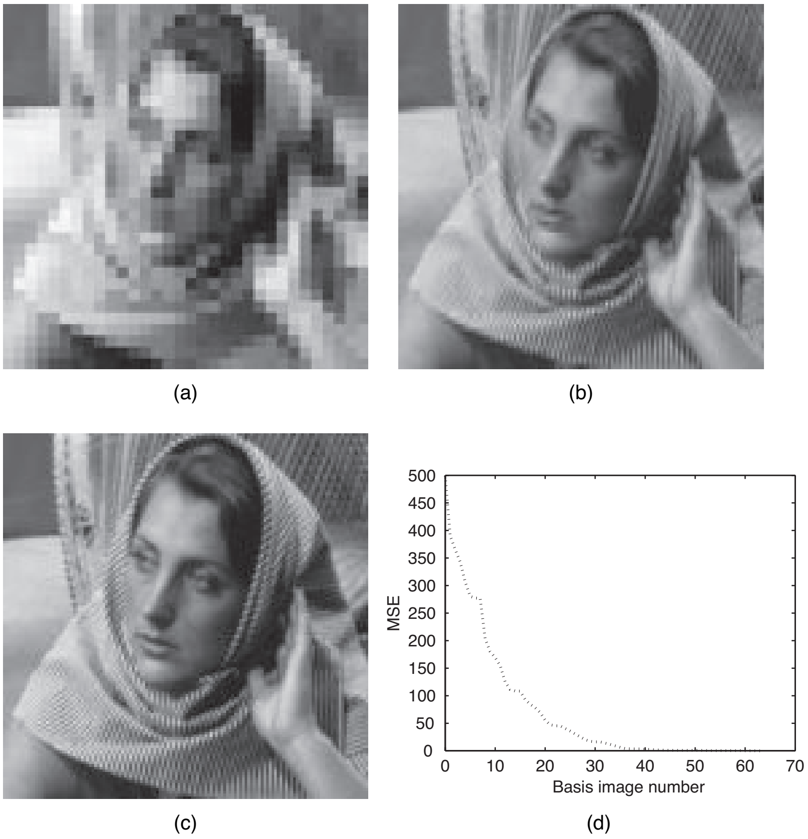


Figure 3.5 Image reconstruction using 8 × 8 DCT basis images: (a) image reconstructed using the first basis image, (b) image reconstructed with 13 basis images belonging to the first and second row, (c) image reconstructed using all 64 basis images, and (d) MSE due to image reconstruction using from 1 to 64 basis images.

number of basis images used in the reconstruction increases. See the MATLAB code listed below.

```
% Example3_4.m
% Expand a given image in terms of the
% basis images corresponding to a user
% defined 2D unitary transform and size
%
N = 8;
TransformType = 'DCT';% options: 'DCT', 'DST', 'Hartley', 'Haar',
% 'Hadamard' and 'Slant'
%
% calculate the 1D transform kernel and the basis images
% by calling the function BasisImages
[y,T] = BasisImages(N,TransformType);
```

```
% read an image
A = imread('barbara.tif');
Height = 256;
Width = 256;
x = A(59:58+Height,331:330+Width,1);% crpped input image
%{
% alternative image to read
A = imread('cameraman.tif');
%[Height,Width,Depth] = size(A);
x = A(:,:,1);
%}
clear A
%
x1 = zeros(N,N);% N x N array of transform coefficients
x2 = zeros(size(x)); % reconstructed image
mse = zeros(N,N); % mse corresponding to using 1 to N*N basis
images
% expand input image in terms of basis images and
% calculate mse due to using 1 to N*N basis images in
% reconstruction
for m = 0:N-1   % take each basis image along rows
    mInd = m*N + 1;
    for n = 0:N-1   % take each basis image along columns
        nInd = n*N + 1;
        for r = 1:N:Height % take N rows at a time
            for c = 1:N:Width   % take N columns at a time
                x1 = y*double( x(r:r+N-1,c:c+N-1))*y';
                sum1 = x2(r:r+N-1,c:c+N-1);% accumulate image
                x2(r:r+N-1,c:c+N-1) = sum1 +...
                    x1(m+1,n+1) * T(mInd:mInd+N-1,nInd:nInd+N-1);
            end
        end
        % show reconstructed image with 1 and 13 basis images
        if (m == 0 && n == 0) || (m == 1 && n == 4)
            figure,imshow(x2,[])
            title(['No. of basis images used = ' num2str(m*N+n+1)])
        end
        mse(m+1,n+1) = (std2(double(x) - x2)) ^2;
    end
end
%
figure,imshow(x2,[])
title('Reconstruction using all basis images')
%
figure,plot(0:N^2-1,[mse(1,:) mse(2,:) mse(3,:) mse(4,:) mse(5,:)
    mse(6,:)... mse(7,:) mse(8,:)],'k:','LineWidth',2)
xlabel([' # of ' TransformType ' basis images used'])
ylabel('MSE'), title(['2D ' TransformType ' Transform']);
```

3.3 KARHUNEN–LOÈVE TRANSFORM

Our objective in compressing an image is to be able to achieve as high a compression as possible with the lowest distortion. That is why we turned to transforming an image from the pixel domain to other domains. Remember that the basic idea behind achieving high compression is to decorrelate the pixels, which is more efficient in the transformed domain. The optimal transform is one that completely decorrelates the pixels in the transform domain. Such an optimal transform is known as the KLT in the literature [13–15]. We will now derive the KLT.

Let $\mathbf{x}$ be an N vector of real random variables and $\mathbf{y} = \mathbf{Ax}$ be the transformed vector. We are looking for the matrix $\mathbf{A}$ that will pairwise completely decorrelate $\mathbf{y}$. This implies that the autocorrelation matrix of $\mathbf{y}$ is diagonal. Denoting the autocorrelation matrix of $\mathbf{y}$ by $\mathbf{R}_{\mathrm{YY}}$, we can write

$$\mathbf{R}_{\mathrm{YY}} = E\left(\mathbf{yy}^{\mathrm{T}}\right) = E\left(\mathbf{Axx}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\right) = \mathbf{A}E\left(\mathbf{xx}^{\mathrm{T}}\right)\mathbf{A}^{\mathrm{T}} = \mathbf{AR}_{\mathrm{XX}}\mathbf{A}^{\mathrm{T}} \tag{3.45}$$

In equation (3.45), the expectation operation is taken with respect to the random vector $\mathbf{x}$ and $\mathbf{R}_{\mathrm{XX}}$ is the autocorrelation matrix of $\mathbf{x}$. We want $\mathbf{R}_{\mathrm{YY}}$ to be a diagonal matrix. The matrix that diagonalizes $\mathbf{R}_{\mathrm{YY}}$ is shown to be the matrix whose column vectors are the eigenvectors of $\mathbf{R}_{\mathrm{XX}}$. Since $\mathbf{R}_{\mathrm{XX}}$ is symmetric and positive semi definite, the N eigenvalues and eigenvectors of $\mathbf{R}_{\mathrm{XX}}$ are real. Let $\mathbf{L}$ be the matrix whose columns are the N eigenvectors of $\mathbf{R}_{\mathrm{XX}}$, that is,

$$\mathbf{L} = [\mathbf{l}_1 \quad \mathbf{l}_2 \quad \ldots \quad \mathbf{l}_N] \tag{3.46}$$

We have assumed in equation (3.46), without loss of generality, that the N eigenvectors are ordered such that the corresponding eigenvalues are in decreasing order, that is,

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_N \geq 0$$

Then the KLT matrix is the transpose of $\mathbf{L}$, that is,

$$\mathbf{A}_{\mathrm{KL}} = \mathbf{L}^{\mathrm{T}} \tag{3.47}$$

Using equation (3.47) in (3.45), we obtain for the autocorrelation matrix of $\mathbf{y}$

$$\mathbf{R}_{\mathrm{YY}} = E\left(\mathbf{A}_{\mathrm{KL}}\mathbf{xx}^{\mathrm{T}}\mathbf{A}_{\mathrm{KL}}^{\mathrm{T}}\right) = \mathbf{L}^{\mathrm{T}}E\left(\mathbf{xx}^{\mathrm{T}}\right)\mathbf{L} = \mathbf{L}^{\mathrm{T}}\mathbf{R}_{\mathrm{XX}}\mathbf{L} = \boldsymbol{\Lambda} \tag{3.48}$$

Where

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & \lambda_N \end{bmatrix} \tag{3.49}$$

Thus, the KLT decorrelates the input vector. It is important to point out that one could use the covariance matrix rather than the correlation matrix [16]. In general, the pdf of a given image is not known. In this situation, we have to compute the autocorrelation matrix of the image empirically by converting the image into vectors of length N and then compute the autocorrelation matrix. We can then determine the eigenvectors and then the KLT matrix. Vectors of length N can be formed from the input image either by grouping pixels along columns or rows. It must also be mentioned that image vectors need not be taken along rows or columns only. One can take the pixels along the diagonal of an $N \times N$ matrix and form the N vectors and then determine the KLT matrix. When we use the KL matrix determined from vectors taken along the column or row dimension only, we have not exploited the true correlation in both dimensions. So, for obtaining 2D KLT of an image, we need to compute the KLT matrices—one using the row dimension and the other using the column dimension. The following example shows the true 2D KLT obtained from an image.

Example 3.5 For the Barbara image, calculate the 2D KLT matrices and then obtain the basis images and display. Observe the effect of reconstructing the image with 1 and 13 basis images. Also compute the resulting MSE due to using from 1 to N^2 basis images. Choose $N = 8$.

Solution First to find the covariance matrix of the image vectors of length N along the column dimension, we can segment the image in multiples of N pixels grouped along the column dimension. Then we proceed to determine the eigenvalues and eigenvectors as explained above. The eigenvector Φ_k satisfies the equation $\mathbf{R_{XX}}\Phi_k = \lambda_k \Phi_k$, $1 \le k \le N$, and the eigenvalues are obtained by solving the equation

$$|\mathbf{R_{XX}} - \lambda \mathbf{I}| = 0$$

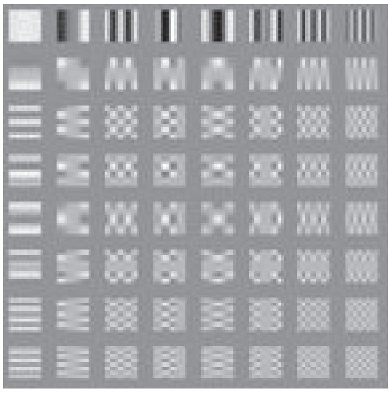
Let the corresponding KLT be denoted by $\mathbf{L}_\mathrm{C}$. Similarly, we proceed to compute the covariance matrix of the image vectors of length N along the row dimension by segmenting the image in multiples of N pixels grouped along the row dimension and then carrying out the rest. Let the KLT matrix obtained in the second case be denoted by $\mathbf{L}_\mathrm{R}$. For an $N \times N$ block of pixels, the KLT can be obtained as

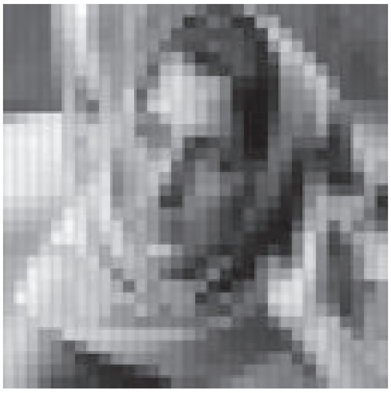
$$\mathbf{Y} = \mathbf{L}_\mathrm{R}^\mathrm{T} \mathbf{X} \mathbf{L}_\mathrm{C}$$

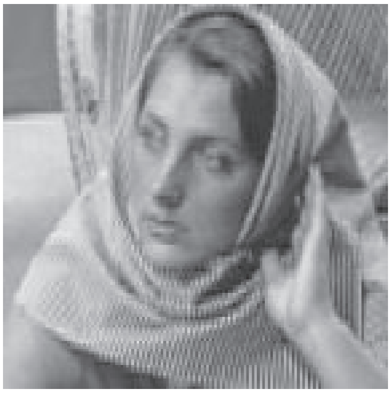
The basis images can now be obtained from $\mathbf{L}_\mathrm{R}$ and $\mathbf{L}_\mathrm{C}$ by multiplying a column of $\mathbf{L}_\mathrm{R}$ by the transpose of a column of $\mathbf{L}_\mathrm{C}$, that is,

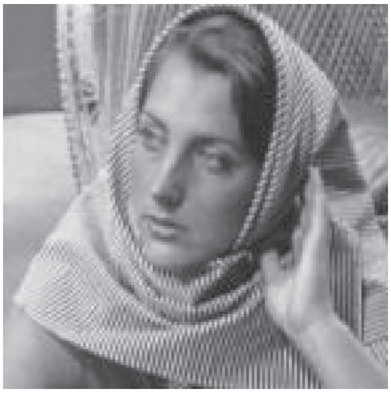
$$\mathbf{b}_{k,l} = \mathbf{l}_{\mathrm{R}k} \mathbf{l}_{\mathrm{C}l}^\mathrm{T}, \quad 0 \le k, l \le N - 1$$

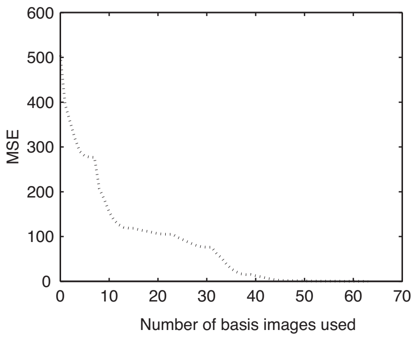
Following the MATLAB code listed below for the example, the 8×8 basis images of the KLT for the Barbara image is shown in Figure 3.6a. Figure 3.6b shows the image reconstructed with 1 basis image, and Figure 3.6c shows the reconstructed


(a)


(b)


(c)


(d)



(e)

Figure 3.6 Image reconstruction using 8 $\times$ 8 KL basis images: (a) basis images of KLT for Barbara image cropped to size 256 $\times$ 256 pixels, (b) image reconstructed with the first basis image, (c) image reconstructed with the first 13 basis images, (d) image reconstructed with all 64 basis images, and (e) MSE due to image reconstruction using from 1 to 64 basis images.

image with 13 out of 64 basis images. These two images are very similar to those obtained using the DCT. Figure 3.6d shows the image reconstructed using all the 64 basis images. As expected, the MSE due to reconstruction using all 64 basis images is zero. Finally, the MSE resulting from image reconstruction using from 1 to 64 KL basis images is shown in Figure 3.6e. Again, it is similar to the DCT case.

```
% Example3_5.m
% Karhunen-Loeve Transform of an image
%
N = 8; % N x N is the transform size
%{
A = imread('cameraman.tif');
[Height,Width,Depth] = size(A);
f = double(A(:,:,1));
%}
A = imread('barbara.tif');
Height = 256;
Width = 256;
f = double(A(59:58+Height,331:330+Width,1));% cropped image
%
b = zeros(N,N);
R = zeros(N,N);
Meanfc = zeros(1,N);% mean vector along column dimension
Meanfr = zeros(1,N);% mean vector along row dimension
% % compute the mean N-vectors along column & row dimensions
for n = 1:N:Width
    for k = 1:N
        Meanfc(k) = Meanfc(k) + sum(f(:,n+k-1));
        Meanfr(k) = Meanfr(k) + sum(f(n+k-1,:),2);
    end
end
Meanfc = floor(Meanfc/(Height*Width/N));
Meanfr = floor(Meanfr/(Height*Width/N));
% calculate the N x N covariance matrix of N-vectors
% along column dimension
for m = 1:Height
    for n = 1:N:Width
        b = (f(m,n:n+N-1)-Meanfc)'*(f(m,n:n+N-1)-Meanfc);
        R = R + b;
    end
end
R = R/max(R(:));
[EVc,Evalc] = eig(R);% eigen matrix of R
% reorder the covariance matrix in descending order of
% eigen values
EVc = EVc(:,N:-1:1);
EVc = EVc(N:-1:1,:);
%
% calculate the N x N covariance matrix of N-vectors
% along rows
R(:,:) = 0;
for n = 1:Width
    for m = 1:N:Height
        b = (f(m:m+N-1,n)-Meanfr')*(f(m:m+N-1,n)-Meanfr')';
        R = R + b;
    end
end
R = R/max(R(:));
[EVr,Evalr] = eig(R);% eigen matrix of R
% reorder the covariance matrix in descending order of
% eigen values
EVr = EVr(:,N:-1:1);
EVr = EVr(N:-1:1,:);
%
% Compute basis images and display
X = ones(192,192) * 16; % inscribe each N x N submatrix into X
T2 = zeros(N*N,N*N);
for k = 0:N-1
    rIndx = k*24 + 1;
    rInd = k*N + 1;
    for l = 0:N-1
        cIndx = l*24 +1;
        cInd = l*N +1;
        y = EVr(:,k+1) * EVc(:,l+1)';
        T = y;
        T2(rInd:rInd+N-1,cInd:cInd+N-1) = y;
        if k == 0
            T = T/max(T(:))*255;
        else
            T = (T-min(T(:)))/(max(T(:))-min(T(:)))*255;
        end
        T1 = imresize(T,[16 16],'bicubic'); % expand matrix
                                            % to 2N x 2N
        X(rIndx+3:rIndx+18,cIndx+3:cIndx+18) = T1;
    end
end
figure,imshow(X,[]),title([num2str(N) 'x' num2str(N) ' basis
                           images for KLT'])
%
% compute mse due to basis restriction
x1 = zeros(N,N);
x2 = zeros(Height,Width);
mse = zeros(N,N);
for m = 0:N-1   % take each basis image along rows
    mInd = m*N + 1;
    for n = 0:N-1   % take each basis image along columns
        nInd = n*N + 1;
        for r = 1:N:Height % take N rows at a time
```

```
            for c = 1:N:Width   % take N columns at a time
                x1 = EVr'*double( f(r:r+N-1,c:c+N-1))*EVc;
                sum1 = x2(r:r+N-1,c:c+N-1);
                x2(r:r+N-1,c:c+N-1) = sum1 +...
                    x1(m+1,n+1) * T2(mInd:mInd+N-1,
                                    nInd:nInd+N-1);
            end
        end

        if(m == 0 && n == 0) || (m == 1 && n == 4)
           figure,imshow(x2,[])
           title(['No. of basis images used = ' num2str(m*N+n+1)])
        end
        mse(m+1,n+1) = (std2(double(f) - x2)) ^2;

    end
end
figure,imshow(x2,[])
%
figure,plot(0:N^2-1,[mse(1,:) mse(2,:) mse(3,:) mse(4,:)
    mse(5,:) mse(6,:)... mse(7,:) mse(8,:)],'k:','LineWidth',2)
xlabel(' # of basis images used')
ylabel('MSE')
title('MSE Vs basis images for KLT')
```

3.4 PROPERTIES OF UNITARY TRANSFORMS

To reiterate our objective, the idea of using an orthogonal transform is to decorrelate the image in the transform domain so that we can individually quantize the transform coefficients to different levels to achieve as high a compression as possible. We will show in this section that that is indeed the property of such transforms.

3.4.1 Energy Is Conserved in the Transform Domain

The transformed vector $\mathbf{y}$ of an N vector $\mathbf{x}$ corresponding to the unitary transform $\mathbf{A}$ is

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

Denote the energies in $\mathbf{x}$ and $\mathbf{y}$ by $E_{\mathbf{x}}$ and $E_{\mathbf{y}}$, respectively. Then,

$$E_{\mathbf{y}} = \|\mathbf{y}\|^2 = \mathbf{y}^{*\mathrm{T}}\mathbf{y} = \mathbf{x}^{*\mathrm{T}}\mathbf{A}^{*\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{x}^{*\mathrm{T}}\mathbf{I}\mathbf{x} = \mathbf{x}^{*\mathrm{T}}\mathbf{x} = \|\mathbf{x}\|^2 = E_{\mathbf{x}}$$

That is, the unitary transform preserves the energy of the signal in the transform domain. The only effect the unitary transform has on $\mathbf{x}$ is to rotate the axes to a new set of orthogonal axes in which the signal is mapped with reduced correlation. A similar result holds for the 2D unitary transform.

3.4.2 Energy in the Transform Domain Is Compacted

This property implies that the signal energy is redistributed in the transform domain unevenly. This is advantageous because some transform coefficients may be insignificant and can be discarded resulting in compression. Again, in the 1D case, the variance of the transformed vector can be found from

$$E\left(\|\mathbf{y}-\boldsymbol{\mu}_{\mathbf{y}}\|^2\right)=E\left\{\left(\mathbf{y}-\boldsymbol{\mu}_{\mathbf{y}}\right)^{*\mathrm{T}}\left(\mathbf{y}-\boldsymbol{\mu}_{\mathbf{y}}\right)\right\} \tag{3.50}$$

Using $\mathbf{y}=\mathbf{Ax}$ in equation (3.50), we have

$$\sum_{k=1}^{N}\sigma_y^2(k)=E\left\{(\mathbf{x}-\boldsymbol{\mu}_{\mathbf{x}})^{*\mathrm{T}}\mathbf{A}^{*\mathrm{T}}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu}_{\mathbf{x}})\right\}=E\,\|\mathbf{x}-\boldsymbol{\mu}_{\mathbf{x}}\|^2=\sum_{n=1}^{N}\sigma_x^2(n) \tag{3.51}$$

Thus, we find that the sum of the variances of the transform coefficients equals the sum of the variances of the input sequence. However, the variances of the transform coefficients may be different even though the variances of the input sequence may all be the same. It is easily seen in the case of the KLT, where the coefficient variance equals the corresponding eigenvalue and that the eigenvalues are not equal in general.

The above property, namely, the energy compaction, holds good in the case of the 2D unitary transform also. As mentioned earlier, DCT has been proven to be an efficient compression vehicle. We, therefore, list below a few additional properties of the DCT without proof.

A few important properties of the DCT are as follows:

1. The DCT is real and orthogonal, which implies that $\mathbf{A}_{\mathrm{DCT}}=\mathbf{A}^*$ and $\mathbf{A}^{-1}=\mathbf{A}^{\mathrm{T}}$.
2. The 1D DCT of an N-point sequence can be computed with $\frac{N}{2}\left(\log_2 N\right)$ operations.
3. The DCT has excellent energy compaction property.
4. The DCT is very close to the KLT in decorrelating the input vector.

Finally, it is worth pointing out that the idea behind the unitary transform is to decorrelate the pixels and to this end, we showed that the KLT is the optimal unitary transform. Other unitary transforms have slightly different properties in this respect. We show here the degree to which the different unitary transforms achieve pixel decorrelation by way of an example.

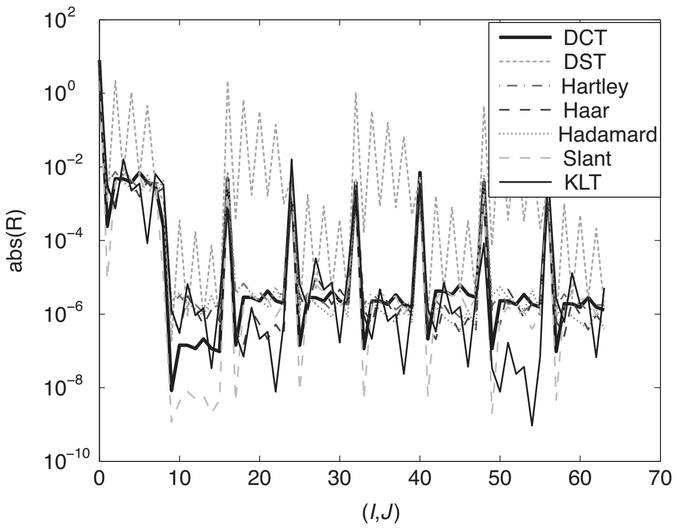


Figure 3.7 Plot of the 8 × 8 autocorrelation matrix of Barbara image for DCT, DST, Hartley, Haar, Hadamard, Slant, and KL transforms. The abscissa represents the element number from left to right, top to bottom order. The ordinate represents the absolute of the autocorrelation value in log scale. Note that the first element of the autocorrelation matrix has the highest value, implying that most of the energy resides in that transform coefficient. Even though other elements have smaller values, the difference in magnitude among the various unitary transforms is clear.

Example 3.6 For the Barbara image, compute the 8 × 8 covariance matrix. Then, calculate the DCT, DST, Hartley, Haar, Hadamard, Slant, and KL transforms. Next, calculate and plot the autocorrelation matrices of the transformed image.

Solution We have already defined the unitary transforms in the previous sections. Except the KLT, other unitary transforms do not depend on the image statistics. After reading in the Barbara image, we call the function named *AutoCorrelation* with arguments input image and N. It returns the $N \times N$ covariance matrix of the image by taking N vectors along the rows. For the KLT, we compute the eigenvectors and reorder them in descending eigenvalues and then obtain the autocorrelation matrix of the transformed image from $\mathbf{y}^{\mathrm{T}}\mathbf{R}_{\mathbf{XX}}\mathbf{y}$, where $\mathbf{R}_{\mathbf{XX}}$ is the covariance matrix of the input image and $\mathbf{y}$ is the KLT. For all other transforms, we call the function named *UnitaryTransform* with arguments N and transform type, which returns the 2D transform and the autocorrelation matrix of the transformed image is then computed from $\mathbf{y}^{\mathrm{T}}\mathbf{R}_{\mathbf{XX}}\mathbf{y}$. Figure 3.7 shows a plot of the autocorrelation matrix of the transformed image for all the unitary transforms specified in the exercise. The ordinate is the absolute value in logarithmic scale of the elements of the autocorrelation matrix taken along each column from left to right. Note that the first element has the highest value and the rest are much smaller in magnitude, especially the off diagonal elements. Also note that the DCT and KLT are very nearly the same and that

DST has less decorrelation than the others. The MATLAB code for this example is listed below.

```
% Example3_6.m
% computes the autocorrelation matrix of
% transformed image using unitary transforms
% Unitary transform types are: DCT, DST, Hartley,
% Haar, Hadamard, Slant, and KL
N = 8; % N x N is the transform size
%
% Barbara image is fairly large and so is cropped to
% size 256 x 256 pixels.
A = imread('barbara.tif');
Height = 256;
Width = 256;
x = double(A(59:58+Height,331:330+Width,1));% cropped image
R = AutoCorrelation(x,N); % call AutoCorrelation to compute Rxx
%
% Next, calculate the N x N unitary transform and then
% compute the autocorrelation matrix of the transformed image
% using Ryy = y*Rxx*y' for all except KL and
% Ryy = y'*Rxx*y for KL transform.
for k = 1:7
    if k == 1
        TransformType = 'DCT';
    elseif k == 2
        TransformType = 'DST';
    elseif k == 3
        TransformType = 'Hartley';
    elseif k == 4
        TransformType = 'Haar';
    elseif k == 5
        TransformType = 'Hadamard';
    elseif k == 6
        TransformType = 'Slant';
    else
        TransformType = 'KLT';
    end
    if k <=6
        % for all except KL
        y = UnitaryTransform(N,TransformType);
        Ry = y*R*y';
    else
        [y,Evalc] = eig(R);% eigen matrix of R
        % reorder the covariance matrix in descending order of
        % eigen values
        y = y(:,N:-1:1);
        y = y(N:-1:1,:);
        Ry = y'*R*y;
    end
    %
    Ry = abs(Ry);
    % plot the absolute of Ryy for each transform with
    % different color
    % and line style.
    if k == 1
        figure,semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'k','LineWidth',2)
        hold on
    elseif k == 2
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'g:','LineWidth',1)
    elseif k == 3
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'r-
.','LineWidth',1)
    elseif k == 4
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'b--
','LineWidth',1)
    elseif k == 5
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'m:','LineWidth',1)
    elseif k == 6
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'c--
','LineWidth',1)
    else
        semilogy(0:N^2-1,[Ry(1,:) Ry(2,:) Ry(3,:) Ry(4,:)
            Ry(5,:) Ry(6,:)... Ry(7,:) Ry(8,:)],'k','LineWidth',1)
    end
end
legend('DCT','DST','Hartley','Haar','Hadamard','Slant', 'KLT',0)
xlabel('(I,J)')
ylabel('abs(R)')
hold off


function R = AutoCorrelation(x,N)
% computes the covariance matrix of x
% x is the input image and
% R is the N x N covariance matrix of x

Meanf = zeros(1,N);
R = zeros(N,N);
[Height,Width] = size(x);
for n = 1:N:Width
```

```
    for k = 1:N
        Meanf(k) = Meanf(k) + sum(x(:,n+k-1));
    end
end
for m = 1:Height
    for n = 1:N:Width
        b = (x(m,n:n+N-1)-Meanf)'*(x(m,n:n+N-1)-Meanf);
        R = R + b;
    end
end
R = R/max(R(:));

function y = UnitaryTransform(N,Type)
% Returns the N x N 1D unitary transform kernel
%
% Input:
% N = size of the basis image, assumed square
% For Haar transform, only N = 8 is allowed
% Type = unitary transform type
% Permissible transforms are:
% DCT, DST, Hartley, Haar, Hadamard, and Slant
%
y = zeros(N,N);
switch Type
    case 'DCT'
        for k = 0:N-1
            y(k+1,:) = cos((2*(0:N-1)+1)*pi*k/(2*N));
        end
        y(1,:) = y(1,:)/sqrt(N);
        y(2:N,:) = y(2:N,:) * sqrt(2/N);
    case 'DST'
        for k = 0:N-1
            y(k+1,:) = sin(((0:N-1)+1)*pi*(k+1)/(N+1));
        end
        y = y*sqrt(2/(N+1));
    case 'Hartley'
        for k = 0:N-1
            y(k+1,:) = cos((0:N-1)*2*pi*k/N) + sin((0:N-
1)*2*pi*k/N);
        end
        y = y*sqrt(1/N);
    case 'Haar'
        sq2 = sqrt(2);
        y = [1 1 1 1 1 1 1 1; 1 1 1 1 -1 -1 -1 -1;...
            sq2 sq2 -sq2 -sq2 0 0 0 0; 0 0 0 0 sq2
            sq2 -sq2 -sq2;...
            2 -2 0 0 0 0 0 0; 0 0 2 -2 0 0 0 0;...
            0 0 0 0 2 -2 0 0; 0 0 0 0 0 0 2 -2];
        y = y/sqrt(8);
    case 'Hadamard'
        y = [1 1; 1 -1];
        y = y/sqrt(2);
        n = 2;
        while n < N
            n = 2*n;
            y = [y y; y -y];
            y = y/sqrt(2);
        end
    case 'Slant'
        a8 = sqrt(16/21);
        b8 = sqrt(5/21);
        s4 = [1 1 1 1; 3/sqrt(5) 1/sqrt(5) -1/sqrt(5) -
3/sqrt(5);...
            1 -1 -1 1; 1/sqrt(5) -3/sqrt(5) 3/sqrt(5) -
1/sqrt(5)];
        s4 = s4/2;
        h1 = [1 0 0 0 1 0 0 0];
        h2 = [a8 b8 0 0 -a8 b8 0 0];
        h3 = [0 0 1 0 0 0 1 0];
        h4 = [0 0 0 1 0 0 0 1];
        h5 = [0 1 0 0 0 -1 0 0];
        h6 = [-b8 a8 0 0 b8 a8 0 0];
        h7 = [0 0 1 0 0 0 -1 0];
        h8 = [0 0 0 1 0 0 0 -1];
        y = [h1;h2;h3;h4;h5;h6;h7;h8]*[s4 zeros(4,4);...
            zeros(4,4) s4]/sqrt(2);
end
```

3.5 SUMMARY

Unitary transforms play an important role not only in image compression but also in image enhancement, image classification, and so on. We briefly described such transforms and elaborated on DCT because of its role in image compression standards. A few examples were given to illustrate the important properties of the unitary transforms. As DCT is a popular transform, we have also shown its fast implementation in a signal flowgraph. We also discussed KLT, which is optimal in decorrelating an image and explained its property by way of a couple of examples. However, due to its dependence on the image in question, KLT is not a part of any image compression standard. But it is used as a yardstick of performance and we found that DCT is the unitary transform that comes close to the KLT in terms of the energy compaction property. For all of the examples in this chapter, MATLAB codes are listed with the file names corresponding to the exercises.

In a later chapter on transform coding, other important derivations such as optimal bit allocation, coding gain, and so on will be given. This chapter is more of an introduction to the various unitary transforms using both sinusoidal and nonsinusoidal basis functions, their properties and implementations. Even though basis images per se are not used in image compression, our interest here was to show how the image reconstruction is affected by the inclusion of restricted number of basis images in terms of visual quality. Again, the examples in this chapter proved that both DCT and KLT are very nearly the same in the amount of MSE introduced when fewer and fewer basis images are included in the reconstruction. Another important orthogonal transform known as the discrete wavelet transform, and its properties and implementation issues will be discussed in the next chapter.

REFERENCES

1. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19 (90), 297–301, 1965.
2. W. T. Cochran et al., "What is the fast fourier transform" *Proc. IEEE*, 55 (10), 164–1674, 1967.
3. N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE Trans. Comp.*, C-23, 90–93, 1974.
4. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
5. M. A. Sid-Ahmed, *Image Processing—Theory, Algorithms, and Architectures*, McGraw-Hill, New York, 1995.
6. M. J. Narasimha and A. M. Peterson, "On the computation of the discrete cosine transform," *IEEE Trans. Comm.*, COM-26(6), 934–936, 1978.
7. R. V. L. Hartley, "A more symmetrical Fourier analysis applied to transmission problems," *Proc. IRE*, 30, 144–150, 1942.
8. R. M. Bracewell, "The discrete Hartley transform," *J. Opt. Soc. Am.*, 73 (12), 1832–1835, 1983.
9. J. Hadamard, "Resolution d'une question relative aux determinants," *Bull. Sci. Math., Ser.* 2(**17**), Part I, 240–246, 1893.
10. A. Haar, "Zur Theorie der Orthogonalen Funktionen-System," *Inaugural Dissertation, Math. Annalen*, 5, 17–31, 1955.
11. J. E. Shore, "On the application of Haar functions," *IEEE Trans. Comm.*, COM-21, 209–216, 1973.
12. W. K. Pratt, H. C. Andrews, and J. Kane, "Hadamard transform image coding," *Proc. IEEE*, 57 (1), 58–68, 1969.
13. K. Karhunen, "Uber Lineare Methoden in der Wahrscheinlich-Kietsrechnung," *Ann. Acd. Sci. Fennicae, Ser. A.1.37*, 1947. English Translation by I. Selin, "On linear methods in probability theory," Doc. T-131, The RAND Corp., Santa Monica, CA, 1960.
14. M. Loève, "Fonctions Aleatoires de Second Ordre," In P. Levy, *Processus Stochastiques et Mouvement Brownien*, Hermann, Paris, 1948.
15. H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psychol.*, 24, 417–441, 498–520, 1933.
16. A. N. Netravali and B. G. Haskell, *Digital Pictures—Representation and Compression*, Plenum Press, New York, 1988.

PROBLEMS

3.1. Show that the 1D DCT of an N-point sequence $x[n], 0 \le n \le N-1$ is not the real part of the corresponding DFT.

3.2. Consider the sequence $x[n], 0 \le n \le N-1$. Show that its DCT can be obtained from $\mathrm{Re}\left\{\alpha(k) W_{2N}^{k/2} \mathrm{DFT}(\tilde{x}[n])\right\}$, where $\tilde{x}[n] = x[2n]$, $0 \le n \le (N/2)-1$, $\tilde{x}[N-1-n] = x[2n+1], 0 \le n \le (N/2)-1$, and

$$\alpha(k) = \begin{cases} \dfrac{1}{\sqrt{N}}, & k = 0 \\ \sqrt{\dfrac{2}{N}}, & 1 \le k \le N-1 \end{cases}$$

3.3. Show that the DST is not the imaginary part of the DFT.

3.4. **(a)** Determine the KLT of a 2×1 vector $\mathbf{x}$ whose covariance matrix is $\begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$.

(b) Find the covariance matrix of the transformed vector. (c) For $\mathbf{x} = [1 \quad 3]^{\mathrm{T}}$, expand it in terms of the basis vectors.

3.5. Consider the image described by $\begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix}$ and the transform $\mathbf{A} = \frac{1}{2}\begin{bmatrix} \sqrt{3} & -1 \\ 1 & \sqrt{3} \end{bmatrix}$. Compute the transformed image and the basis images.

3.6. Show that the determinant of the unitary transform is unity and that all the eigenvalues of the unitary matrix have unity magnitude.

3.7. Prove that the $N \times N$ DCT matrix is orthogonal.

3.8. Read one or two images of the same type, that is, black and white with 8 bpp, and compute the 8×8 DCT of the images. Then determine the histogram of the 64 transform coefficients and plot them. Describe the nature of the histograms. Repeat the procedure for 16×16 DCT.

3.9. Read one or two images in Problem 3.8. Compute the 8×8 unitary transform of the images. Then compute the variances of the 64 transform coefficients and plot them. The transforms you will consider are Haar, Hadamard, Slant, DCT, DST, Hartley, and KLT.

4

DISCRETE WAVELET TRANSFORM

4.1 INTRODUCTION

Wavelet transform has greatly impacted the field of signal analysis especially image analysis and compression. As we mentioned earlier, our interest in the study of wavelet transform centers on its exploitation in image compression. The amount of compression that can be achieved depends on the energy compaction property of the transform being used. We saw in the last chapter that discrete cosine transform is very nearly the same as the Karhunen–Loève transform, which is the optimal transform. Similarly, we will see that the wavelet transform also has a high-energy compaction property that makes it a very suitable candidate for image compression. In addition, it has other interesting properties such as progressive reconstruction that make wavelet transform a powerful tool for image and video compression.

The transforms that we discussed so far have sinusoidal or rectangular basis functions. A transform based on sinusoids, such as the DFT, can capture only frequencies in the steady state, that is, such a transform can enumerate the different frequencies present in the signal being analyzed from start to finish but cannot capture the moments that such frequencies begin or end. In the same manner, transforms based on sinusoids cannot pinpoint locations where activities occur in an image. Why is this feature important? When compressing an image, if we can distinguish areas of intense activity from flat regions, we may be able to allocate different number of bits of quantization to these different regions, thereby achieving a high compression without sacrificing visual quality. Even though this is possible with transforms based on sinusoids, wavelets are more efficient because of their inherent property of capturing transients.

Similar to the Fourier transform, wavelet transform comes in different flavors: continuous wavelet transform (CWT) and discrete wavelet transform (DWT) [1–8]. From a computational point of view, the discrete version of the wavelet transform will be very useful and meaningful. Therefore, we will concentrate on the *DWT*, its properties, and implementations and illustrate them with a few examples. For the sake of completeness, we will briefly describe the CWT and then proceed to a detailed discussion on the DWT.

4.2 CONTINUOUS WAVELET TRANSFORM

Unlike the continuous Fourier transform, which uses complex sinusoids as its basis functions, the wavelet transform uses *wavelets* as its basis functions. Each sinusoidal basis function in the Fourier transform exists over an infinite interval with constant amplitude. On the other hand, a wavelet is a fast decaying function even though it may exist over an infinite interval. The Fourier transform of a continuous-time signal is a function of the frequency only. But the wavelet transform of a continuous-time signal is a function of *scale* and *shift*. A scale as used in the wavelet transform is roughly related to the inverse frequency—small scale implies large frequencies and vice versa. In the Fourier transform of an a periodic signal, all the sinusoidal frequencies are not related to each other, that is, the frequencies are continuous. In wavelet transform, it is possible to have all the wavelet functions related to a single wavelet called the *mother wavelet*, from which all the wavelet functions are obtained by dilation and contraction. Figure 4.1 shows two wavelets scaled by different values. These two wavelets are called Daubechies wavelets. The formal definition of the *CWT* follows.

Given a continuous-time signal $f(t)$, $-\infty \le t \le \infty$, its CWT $W(s, \tau)$ is defined as

$$W(s,\tau) = \langle f(t), \psi_{s,\tau} \rangle = \int_{-\infty}^{\infty} f(u)\, \psi_{s,\tau}(u)\, du \tag{4.1}$$

In equation (4.1), s is the scale and τ is the shift and the basis functions $\psi_{s,\tau}(t)$ are the dilations and contractions of the mother wavelet given by

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{s}} \psi\left(\frac{t-\tau}{s}\right) \tag{4.2}$$

In equation (4.1), both the scale and shift parameters are continuous variables. *Scale* is proportional to the duration of the wavelet functions. Therefore, large scale corresponds to wavelets with a long duration and when a signal is expanded in terms of such wavelets, the effect is to capture the long-term behavior. On the other hand, small scale implies short duration for the wavelets and the signal expansion with small-scale wavelets captures short-term behavior of the signal. Thus, wavelet

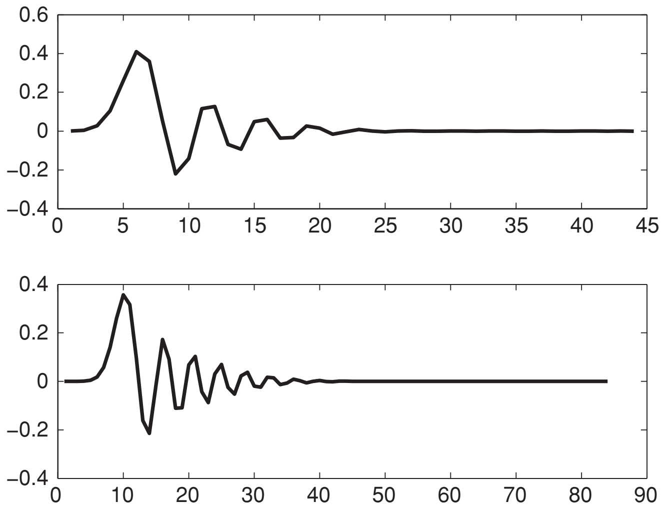


Figure 4.1 Scaling of a mother wavelet. The top figure shows Daubechies wavelet "db22", and the bottom figure shows Daubechies wavelet "db42."

analysis aids in capturing both long- and short-term trends in a signal whereas Fourier transform analysis captures only the long-term behavior of a signal since all the basis functions have infinite duration. The factor $1/\sqrt{s}$ guarantees that all the wavelets have the same energy, that is,

$$\int_{-\infty}^{\infty} \psi_{s,\tau}(t)\psi_{s,\tau}^{*}(t)\,dt = \int_{-\infty}^{\infty} \left|\psi_{s,\tau}(t)\right|^2 dt = \frac{1}{s}\int_{-\infty}^{\infty} \left|\psi\left(\frac{t-\tau}{s}\right)\right|^2 dt \qquad (4.3)$$

Using $u = (t-\tau)/s$ in equation (4.3), we get

$$\int_{-\infty}^{\infty} \left|\psi_{s,\tau}(t)\right|^2 dt = \int_{-\infty}^{\infty} |\psi(u)|^2 du \qquad (4.4)$$

Thus, all the wavelets have the same energy equal to that of the mother wavelet. From equation (4.1), we observe that the scale parameter gives a measure of the frequency of the signal at a time instant specified by the shift parameter.

The CWT is unique and we can, therefore, recover the signal from its CWT. Thus, the inverse CWT [9, 10] of $f(t)$ is given by

$$f(t) = \frac{1}{C_\Psi}\int_{0}^{\infty}\int_{-\infty}^{\infty} W(s,\tau)\,\psi_{s,\tau}(t)\,d\tau\frac{ds}{s^2} \qquad (4.5)$$

where the constant in equation (4.5) is given by

$$C_\Psi = \int_{-\infty}^{\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega \prec \infty \tag{4.6}$$

In equation (4.6), $\Psi(\omega)$ is the Fourier transform of $\psi(t)$. We notice from equation (4.6) that C_Ψ exists only if $\Psi(0) = 0$. This implies that the wavelets all have zero average value, that is, the wavelets are bandpass functions.

4.3 WAVELET SERIES

If the scale and shift parameters in equation (4.1) are discrete rather than continuous, then it is possible to expand $f(t)$ in a wavelet series, that is, summation rather than integral [11, 12]. More specifically, if the scale parameter is allowed to take on values that are integer power of two (called binary scaling) and the shift parameter is allowed to take on integer values (known as dyadic translation), then the wavelet series of $f(t)$ is expanded in terms of (1) infinite sum of scaling functions and (2) infinite sum of wavelets. Thus, the wavelet series of $f(t)$ is written as

$$f(t) = \sum_k a(j_0, k)\,\phi_{j_0,k}(t) + \sum_{j=j_0}^{\infty} \sum_{k=-\infty}^{\infty} d(j,k)\,\psi_{j,k}(t) \tag{4.7}$$

Note that $f(t)$ need not be periodic in order for us to expand it in a wavelet series. In equation (4.7), the first summation involves the basis functions $\phi_{j_0,k}(t)$ at a fixed scale j_0 and the summation is over all possible shift values. The functions $\phi_{j_0,k}(t)$ are called *scaling functions* and are obtained by binary scaling and shifting of a prototype function as given by equation (4.8):

$$\phi_{j,k}(t) = 2^{j/2}\phi\left(2^j t - k\right), -\infty \prec j, k \prec \infty; \quad j, k \in Z \tag{4.8}$$

Also, the first summation in equation (4.7) represents an approximation of $f(t)$ similar to the average value given in the Fourier series expansion of a periodic (or periodically extended) signal. The second summation in the wavelet series involves the *dyadic* wavelets, which are expressed as

$$\psi_{j,k}(t) = 2^{j/2}\psi\left(2^j t - k\right), -\infty \prec \quad j, k \prec \infty; \quad j, k \in Z \tag{4.9}$$

Similar to the Fourier series expansion, the second summation in equation (4.7) represents finer details of the signal $f(t)$. The wavelet series coefficients in equation

(4.7) can be evaluated from

$$a\left(j_0, k\right) = \left\langle f, \phi_{j_0,k}\right\rangle = 2^{j/2} \int_{-\infty}^{\infty} f\left(t\right) \phi_{j_0,k}\left(t\right) dt \tag{4.10a}$$

$$d\left(j, k\right) = \left\langle f, \psi_{j,k}\right\rangle = 2^{j/2} \int_{-\infty}^{\infty} f\left(t\right) \psi_{j,k}\left(t\right) dt \tag{4.10b}$$

Corresponding to the scaling and wavelet functions, the coefficients $a\left(j_0, k\right)$ are called the *scaling coefficients* and the $d\left(j, k\right)$ coefficients are called the detail coefficients.

4.4 DISCRETE WAVELET TRANSFORM

If the signal, scaling functions, and wavelets are discrete in time, then the wavelet series in equation (4.7) of the discrete-time signal is called the *DWT*. The DWT of a sequence consists of two series expansions, one corresponding to the approximation and the other to the details of the sequence. The formal definition of DWT of an N-point sequence $x\left[n\right], 0 \leq n \leq N-1$ [13] is given by

$$\text{DWT}\left\{f\left(t\right)\right\} = W_\phi\left(j_0, k\right) + W_\psi\left(j, k\right) \tag{4.11}$$

where

$$W_\phi\left(j_0, k\right) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x\left[n\right] \phi_{j_0,k}\left[n\right] \tag{4.12a}$$

$$W_\psi\left(j, k\right) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x\left[n\right] \psi_{j,k}\left[n\right], \quad j \geq j_0 \tag{4.12b}$$

The sequence $x\left[n\right], 0 \leq n \leq N-1$ can be recovered from the DWT coefficients W_ϕ and W_ψ as given by

$$x\left[n\right] = \frac{1}{\sqrt{N}} \sum_k W_\phi\left(j_0, k\right) \phi_{j_0,k}\left[n\right] + \frac{1}{\sqrt{N}} \sum_{j=j_0}^{\infty} \sum_k W_\psi\left(j, k\right) \psi_{j,k}\left[n\right] \tag{4.13}$$

The scale parameter in the second summation of equation (4.13) has an infinite number of terms. But in practice the upper limit for the scale parameter is usually fixed at some value say, J. The starting scale value j_0 is usually set to zero and corresponds to the original signal. Thus, the DWT coefficients for $x\left[n\right], 0 \leq n \leq N-1$ are computed for $j = 0, 1, \ldots, J-1$ and $k = 0, 1, \ldots, 2^j - 1$. Also, N is typically a power of 2, of the form $N = 2^J$.

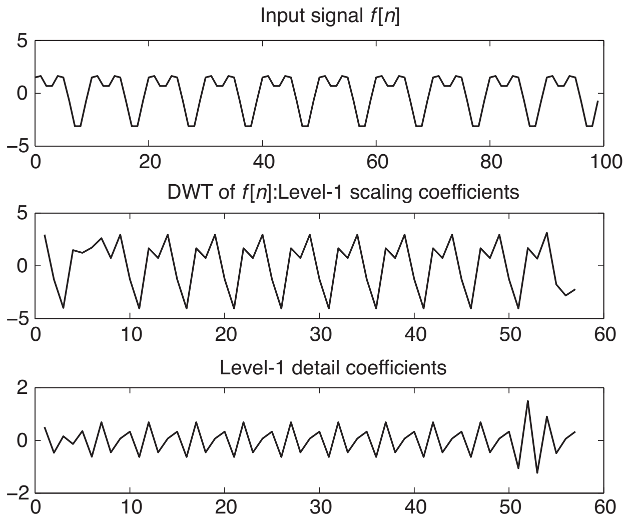


Figure 4.2 1D DWT of the sequence in Example 4.1. The top figure shows input sequence, the middle figure shows level-1 scaling coefficients, and the bottom figure shows level-1 detail coefficients.

Example 4.1 Consider a sequence described by $x[n] = 2\sin(2\pi \times 0.1n) + 1.5\cos(2\pi \times 0.2n), 0 \leq n \leq 99$. Let us compute its one-dimensional (1D) DWT up to two levels or scales using Daubechies's wavelet "db8", which corresponds to a 16-tap FIRfilter, using the MATLAB function *DWT*. Figure 4.2 shows the plots of the input sequence (top figure), the level-1 scaling coefficients (middle plot), and

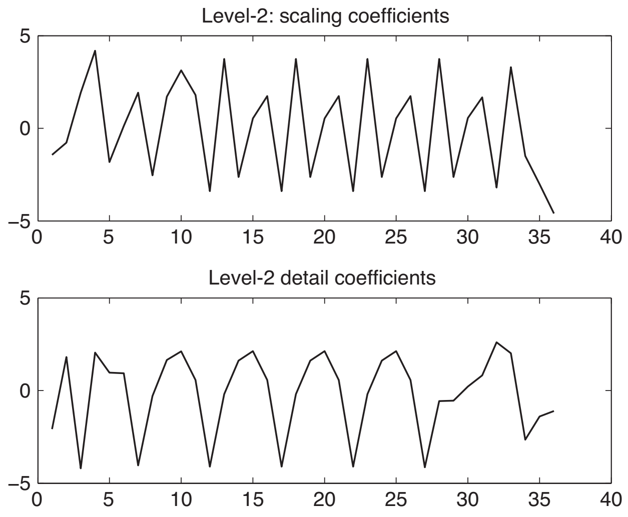


Figure 4.3 DWT of the level-1 scaling coefficients in Figure 4.2. The top figure shows level-2 scaling coefficients, and the bottom figure shows level-2 detail coefficients.

the detail or wavelet coefficients (bottom plot). As can be seen from the figure, the scaling coefficients are an approximation to the input sequence while the wavelet coefficients try to follow finer variations in the signal. A second-level DWT of the scaling coefficients obtained in level 1 is computed, which consists of level-2 scaling and wavelet coefficients. Figure 4.3 shows the plots of the scaling and wavelet coefficients at level 2. Again, the second-level scaling coefficients approximate those in level 1 and the wavelet coefficients in level 2 retain the finer or local variations in the scaling coefficients in level 1. We also notice from Figure 4.3 that the amplitudes of the wavelet coefficients at level 2 are amplified compared with those at level 1. Refer to the listing below for the MATLAB code for this example.

```
% Example4_1.m
% Computes 1D DWT of a sequence f[n] up to 2 levels
% and plots the DWT coefficients
% specify the name of the wavelet in ''WaveletName''
% f[n] = A1xsin(2*pi*F1*n) + A2xcos(2*pi*F2*n)
%
WaveletName = 'db8';
n = 0:99;% sequence length
A1 = 2.0; A2 = 1.5;% amplitudes of the two sinusoids
F1 = 0.1; F2 = 0.2;% frequencies normalized to sampling
%frequency
f = A1*sin(2*pi*F1*n) + A2*cos(2*pi*F2*n);
% compute level 1 DWT of f[n], which consists of the
% approximation coefficients ''a1'' and detail
% coefficients ''d1.''
[a1,d1] = dwt(f,WaveletName);
figure,subplot(3,1,1),plot(n,f,'k','LineWidth',1);
title('Input signal f[n]')
subplot(3,1,2),plot(a1,'k','LineWidth',1)
title('DWT of f[n]:Level-1 scaling coefficients')
subplot(3,1,3),plot(d1,'k','LineWidth',1)
title('Level-1 detail coefficients')
% Next, compute level 2 DWT of a1, the level 1 approximation
[a2,d2] = dwt(a1,WaveletName);
figure,subplot(2,1,1),plot(a2,'k','LineWidth',1)
title('level-2: scaling coefficients')
subplot(2,1,2),plot(d2,'k','LineWidth',1)
title('Level-2 detail coefficients')
```

4.5 EFFICIENT IMPLEMENTATION OF 1D DWT

Subband coding was first introduced for processing speech signals [14–16]. It is an efficient way of decomposing speech signals into a set of nested (octave band) frequency bands for the purpose of compression. It was later applied to image compression application. For the time being, let us consider a 1D sequence of length N. Subband coding consists of a sequence or stages of filtering and subsampling processes. In the first stage, the input sequence is filtered by two filters, a lowpass filter, and a highpass filter. The output of each filter is decimated by a factor of 2, that is, only every other output sample is retained at the filter output. In the frequency domain, the outputs of the filters occupy the frequency regions below and above half the sampling frequency. Note that the two outputs may have overlapping frequency regions due to the filter responses. Thus, in the first stage, there are two output sequences of length each $N/2$ corresponding to low and high frequencies. In the second stage, the output of the first stage lowpass filter is filtered by the same lowpass and highpass filters and decimated by a factor of 2 to give two length $N/4$ output sequences. This process of filtering the output of the lowpass filter and decimating is repeated M times. This type of subband coding is called the *octave-band subband coding* because each time only the lowpass region is split into two halves. This stage of subband coding where the input signal is decomposed into octave frequency bands is known as the *analysis* stage. In the *synthesis* or reconstruction stage, the procedures are repeated in the reverse direction using another set of lowpass and highpass filters. First, the output of the lowpass filter at the Mth stage is upsampled by 2 (zero value inserted between samples) and then filtered by a synthesis lowpass filter. Similarly, the output of the highpass filter at the Mth stage is upsampled by 2 and then filtered by a synthesis highpass filter. The two outputs are added. This process is repeated until a sequence of length N is obtained, which is the reconstructed sequence. A two-stage subband coding of a sequence is shown in Figure 4.4. Figure 4.4a is the analysis part and Figure 4.4b is the synthesis part.

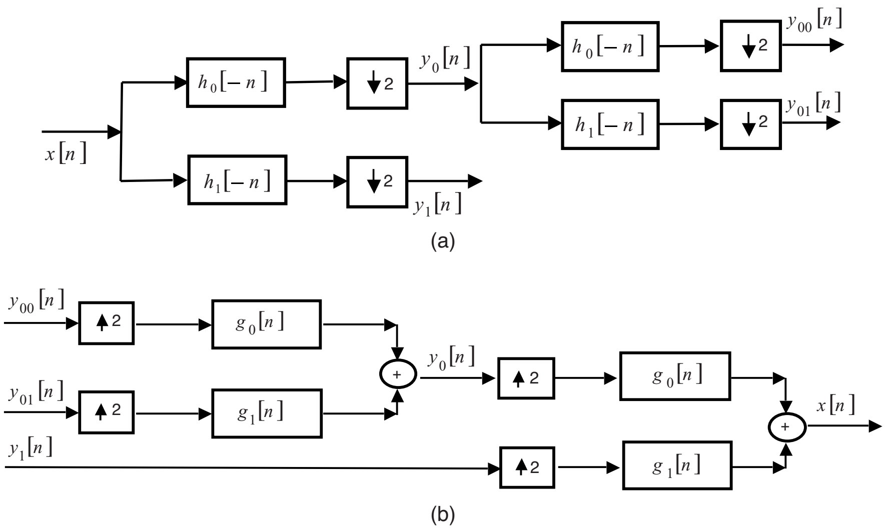


Figure 4.4 A two-stage subband coder: (a) analysis portion and (b) synthesis portion.

Due to filtering and subsampling by 2, the filtered signals suffer from *aliasing* distortion. Similarly, upsampling followed by filtering introduces *image* frequency. Therefore, use of lowpass and highpass filters with arbitrary frequency responses will result in a synthesized signal that suffers from aliasing and image distortions even without any signal quantization. This problem is negated when the analysis and synthesis filters satisfy certain properties. This results in what is known as the *quadrature mirror filter bank* [17, 18]. Because of certain symmetries that exist between the analysis and synthesis filters, the filtering becomes very efficient. In the following, we will show how the DWT of a sequence can be computed using the octave-band subband coding scheme.

Recall that the DWT of a sequence $x[n], 0 \leq n \leq N-1$ consists of the scaling and wavelet coefficients given in equations (4.11) and (4.12). Let us start with the lowest scale j_0. Typically, the lowest scale corresponds to the input sequence being analyzed. Since the scaling functions are nested, we can express the scaling functions at scale j as a linear combination of the scaling functions in scale $j+1$:

$$\phi\left[2^j n - k\right] = \sum_m h_0[m]\sqrt{2}\phi\left[2\left(2^j n - k\right) - m\right] \tag{4.14}$$

In equation (4.14), let $l = m + 2k$. Then, equation (4.14) can be written as

$$\phi\left[2^j n - k\right] = \sum_l h_0[l-2k]\sqrt{2}\phi\left[2^{j+1} n - l\right] \tag{4.15}$$

In equation (4.14), $\{h_0[m]\}$ are the weights used in the linear combination. Similarly, the wavelet function at scale j can be written in terms of the wavelet functions at scale $j+1$ using another set of weights $\{h_1[m]\}$ as

$$\psi\left[2^j n - k\right] = \sum_l h_1[l-2k]\sqrt{2}\psi\left[2^{j+1} n - l\right] \tag{4.16}$$

In equations (4.14) through (4.16), the factor $\sqrt{2}$ is the normalization factor so that all scaling functions and wavelets have unit energy. Substituting equation (4.15) into (4.12a), we can express the scaling coefficients of the DWT of the input sequence at scale j in terms of the scaling coefficients at scale $j+1$ as

$$\begin{aligned} W_\phi(j,k) &= \frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} x[n]\phi\left[2^j n - k\right] \\ &= \frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} x[n]\sum_l h_0[l-2k]\sqrt{2}\phi\left[2^{j+1}n - l\right] \\ &= \sum_l h_0[l-2k]\left\{\frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} x[n]\sqrt{2}\phi\left[2^{j+1}n - l\right]\right\} \end{aligned} \tag{4.17}$$

But,

$$\frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] \sqrt{2}\phi \left[2^{j+1}n - l\right] = W_{\phi}(j+1, l) \tag{4.18}$$

Therefore,

$$W_{\phi}(j, k) = \sum_{l} h_0 [l - 2k] W_{\phi}(j+1, l) \tag{4.19}$$

By a similar reasoning, we can write the wavelet or detail coefficients of the DWT of the input sequence at scale j in terms of those at scale $j+1$ as

$$W_{\psi}(j, k) = \sum_{l} h_1 [l - 2k] W_{\psi}(j+1, l) \tag{4.20}$$

By inspecting equation (4.19), we can interpret the scaling coefficients at scale j as being obtained by convolving (or filtering) the scaling coefficients at scale $j+1$ by a filter $h_0[-n]$ and retaining every other output sample. The filter with impulse response $h_0[-n]$ is the time-reversed version of the filter $h_0[n]$. Similarly, we can interpret the wavelet coefficients at scale j as being obtained by filtering the wavelet coefficients at scale $j+1$ by a filter $h_1[-n]$ and retaining every other output sample. One can iterate the process of filtering the scaling coefficients by $h_0[-n]$ and wavelet coefficients by $h_1[-n]$ followed by subsampling by 2 to as many stages as required. However, in practice the number of stages of iteration is fixed at some value such that we have at least as many samples left as there are taps in the filters.

To reconstruct the sequence from its scaling and wavelet coefficients, we start at the highest scale, upsample the scaling and wavelet coefficients by 2, filter them through two filters $g_0[n]$ and $g_1[n]$, and add the two filtered outputs. This results in the reconstruction of the scaling coefficients at the next lower scale. The process is continued until the full-length signal is obtained. The two reconstruction or synthesis filters are simply the time-reversed versions of the corresponding analysis filters.

So far we have not discussed how to design the four filters used in the analysis and synthesis parts of the DWT nor have discussed their relationship to the wavelets. In the following, we will briefly address these two issues.

4.6 SCALING AND WAVELET FILTERS

In order to be able to synthesize the signal exactly from its DWT, the FIR filters in the filter bank must satisfy certain conditions. We noted earlier that the distortions that will occur are the aliasing and image distortions. In order to cancel these distortions, the synthesis filters must possess what is called the *power complimentary* or

Smith–Barnwell property [19–24], which is

$$\left|G_0\left(e^{j\omega}\right)\right|^2 + \left|G_1\left(e^{j\omega}\right)\right|^2 = 2 \tag{4.21}$$

In equation (4.21), $G_0\left(e^{j\omega}\right)$ and $G_1\left(e^{j\omega}\right)$ are the discrete-time Fourier transform of the synthesis filters $g_0[n]$ and $g_1[n]$, respectively. The wavelets used in the DWT are either orthogonal or biorthogonal. Depending on the type of the DWT that is being used, the filter bank must satisfy a set of conditions. We will describe them in the following section.

4.6.1 Orthogonal DWT

In order to implement a two-channel perfect reconstruction orthogonal DWT, the FIR filters used in the filter bank possess the following properties:

1. The filter length L is even.
2. The filters $g_0[n]$ and $g_1[n]$ satisfy the power complimentary condition given in equation (4.21). Similarly, the filter pairs $\{h_0[n], h_1[n]\}$, $\{h_0[n], g_1[n]\}$, and $\{g_0[n], h_1[n]\}$ satisfy the power complimentary condition.
3. The filters $g_0[n]$ and $h_0[n]$ are time-reversed versions of each other, that is, $h_0[n] = g_0[-n]$.
4. The filters $g_1[n]$ and $h_1[n]$ are time-reversed versions of each other.
5. The filters $h_1[n]$ and $h_0[n]$ satisfy the condition $h_1[n] = (-1)^{n+1} h_0[L-1-n]$.
6. The filters $g_1[n]$ and $g_0[n]$ are time-reversed and modulated versions of each other, that is, $g_1[n] = (-1)^n g_0[L-1-n]$.
7. Finally, $\sum_n h_0[n] = \sum_n g_0[n] = \sqrt{2}$.

The filters $g_0[n]$ and $h_0[n]$ are lowpass, while the filters $g_1[n]$ and $h_1[n]$ are highpass. Obviously, the sum of the highpass filter coefficients equals zero. Although the filter bank satisfying conditions 1–7 realizes an orthogonal DWT, the FIR filters do not have linear phase response. All the four filters can be realized from a prototype lowpass filter. Thus, the task is to design a single lowpass prototype filter and then obtain all the four filters by making use of the properties listed above. As an example, the "db8" filter bank we used in Example 4.1 is listed in Table 4.1. The filters have even lengths. It can be easily verified that both the lowpass filters sum up to $\sqrt{2}$.

4.6.2 Biorthogonal DWT

As pointed out earlier, the FIR filters used in the orthogonal DWT cannot have linear phase. This is overcome in the biorthogonal DWT. In order to implement a

Table 4.1 Filter bank used in Example 4.1—“db8” wavelets

$h_0[n]$	$h_1[n]$	$g_0[n]$	$g_1[n]$
−0.0001	−0.0544	0.0544	−0.0001
0.0007	0.3129	0.3129	−0.0007
−0.0004	−0.6756	0.6756	−0.0004
−0.0049	0.5854	0.5854	0.0049
0.0087	0.0158	−0.0158	0.0087
0.0140	−0.2840	−0.2840	−0.0140
−0.0441	−0.0005	0.0005	−0.0441
−0.0174	0.1287	0.1287	0.0174
0.1287	0.0174	−0.0174	0.1287
0.0005	−0.0441	−0.0441	−0.0005
−0.2840	−0.0140	0.0140	−0.2840
−0.0158	0.0087	0.0087	0.0158
0.5854	0.0049	−0.0049	0.5854
0.6756	−0.0004	−0.0004	−0.6756
0.3129	−0.0007	0.0007	0.3129
0.0544	−0.0001	−0.0001	−0.0544

two-channel perfect reconstruction biorthogonal DWT, the FIR filters used in the filter bank possess the following properties:

1. The filter length L is even.
2. The filters have linear phase response.
3. $g_0[n]$ and $h_0[n]$ are not time-reversed versions of each other.
4. Similarly, $g_1[n]$ and $h_1[n]$ are not time-reversed versions of each other.
5. $h_0[n]$ and $g_1[n]$ are modulated versions of each other, that is, $g_1[n] = (-1)^n h_0[n]$.
6. Similarly, $g_0[n]$ and $h_1[n]$ are modulated versions of each other with one sign change, that is, $h_1[n] = (-1)^{n+1} g_0[n]$.
7. $\sum_n h_0[n] = \sum_n g_0[n] = \sqrt{2}$.

As an example, the biorthogonal “bior6.8” filter bank is listed in Table 4.2. The filter length is 18. It can be easily verified that both the lowpass filters sum up to $\sqrt{2}$.

4.6.3 Construction of Wavelets

In Section 4.5, we showed how to compute the DWT of a finite-length sequence by iterating the two-channel subband coding. In this scheme, the only thing that mattered was the filter bank. Since this chapter deals with wavelets, it is imperative that we should be able to construct a specific wavelet from the corresponding subband coding filter bank.

Table 4.2 Filter bank corresponding to biorthogonal wavelets "bior6.8"

$h_0[n]$	$h_1[n]$	$g_0[n]$	$g_1[n]$
0	0	0	0
0.0019	0	0	−0.0019
−0.0019	0	0	−0.0019
−0.0170	0.0144	0.0144	0.0170
0.0119	−0.0145	0.0145	0.0119
0.0497	−0.0787	−0.0787	−0.0497
−0.0773	0.0404	−0.0404	−0.0773
−0.0941	0.4178	0.4178	0.0941
0.4298	−0.7589	0.7589	0.4298
0.8259	0.4178	0.4178	−0.8259
0.4208	0.0404	−0.0404	0.4208
−0.0941	−0.0787	−0.0787	0.0941
−0.0773	−0.0145	0.0145	−0.0773
0.0497	0.0144	0.0144	−0.0497
0.0119	0	0	0.0119
−0.0170	0	0	0.0170
−0.0019	0	0	−0.0019
0.0019	0	0	−0.0019

Haar Wavelets Recall that the analysis lowpass filter must satisfy the conditions (1) $\sum_n h_0[n] = \sqrt{2}$ and (2) $\sum_n h_0[n]\, h_0[n-2k] = \delta(k)$. For the Haar wavelets with $N = 2$, condition (1) results in the equation

$$h_0[0] + h_0[1] = \sqrt{2} \tag{4.22}$$

and condition (2) yields the equation

$$h_0^2[0] + h_0^2[1] = 1 \tag{4.23}$$

From equations (4.22) and (4.23), the Haar scaling function is found to be $\{h_0[n]\} = \{1/\sqrt{2},\ 1/\sqrt{2}\}$. Using the modulation property, we find that the Haar wavelet is given by $\{h_1[n]\} = \{1/\sqrt{2},\ \ -1/\sqrt{2}\}$.

Daubechies's N = 4 Wavelet Again for $N = 4$, using the two conditions listed above, we get the following three equations:

$$h_0[0] + h_0[1] + h_0[2] + h_0[3] = \sqrt{2} \tag{4.24a}$$

$$h_0^2[0] + h_0^2[1] + h_0^2[2] + h_0^2[3] = 1 \tag{4.24b}$$

$$h_0[0]\, h_0[2] + h_0[1]\, h_0[3] = 0 \tag{4.24c}$$

In terms of the parameter θ, it is possible to express the solution to equation (4.24) as [11]

$$h_0[0] = \frac{1 - \cos(\theta) + \sin(\theta)}{2\sqrt{2}}$$

$$h_0[1] = \frac{1 + \cos(\theta) + \sin(\theta)}{2\sqrt{2}}$$

$$h_0[2] = \frac{1 + \cos(\theta) - \sin(\theta)}{2\sqrt{2}}$$

$$h_0[3] = \frac{1 - \cos(\theta) - \sin(\theta)}{2\sqrt{2}}$$

For the Daubechies's wavelet, $\theta = \frac{\pi}{3}$ and the wavelet results in the sequence $\{h_0[n]\} = \{(1+\sqrt{3})/4\sqrt{2},\ (3+\sqrt{3})/4\sqrt{2},\ (3-\sqrt{3})/4\sqrt{2},\ (1-\sqrt{3})/4\sqrt{2}\}$.

In general, one can obtain a wavelet by iterating many times the reconstruction filter $g_0[n]$. The exact procedure is as follows:

1. Start with the lowpass filter $g_0[n]$.
2. Modulate $g_0[n]$ to obtain the highpass filter $g_1[n] = (-1)^n g_0[n]$.
3. Upsample $g_1[n]$, that is, insert zeros between samples of $g_1[n]$.
4. Convolve the upsampled filter with $g_0[n]$.

If the above procedure is repeated many times, the resultant filter will approach the shape of a wavelet. The wavelet shape depends, of course, on the filter $g_0[n]$.

Example 4.2 Build the Daubechies's wavelet and scaling functions starting with the filter "db2." Plot the wavelets and scaling functions for 1 and 5 iterations of the procedure listed above. Repeat the procedure to build "bior2.2" wavelet and scaling functions.

Solution Use the MATLAB function called "wfilters" with arguments "db2" and "r." The second argument denotes reconstruction. This function returns two filters, lowpass and highpass filters of length 4. Then follow the wavelet construction procedure listed above to build the respective wavelet and scaling functions. The reconstruction scaling function (bottom plot) and wavelet (top plot) with one iteration are shown in Figure 4.5a. Figure 4.5b shows the same scaling and wavelet function with 5 iterations. We see from these figures that the wavelet and scaling function quickly converge to the final form. The analysis counterparts are shown in Figures 4.6a,b. Using the same procedure, we next build the "bior2.2" wavelet and scaling

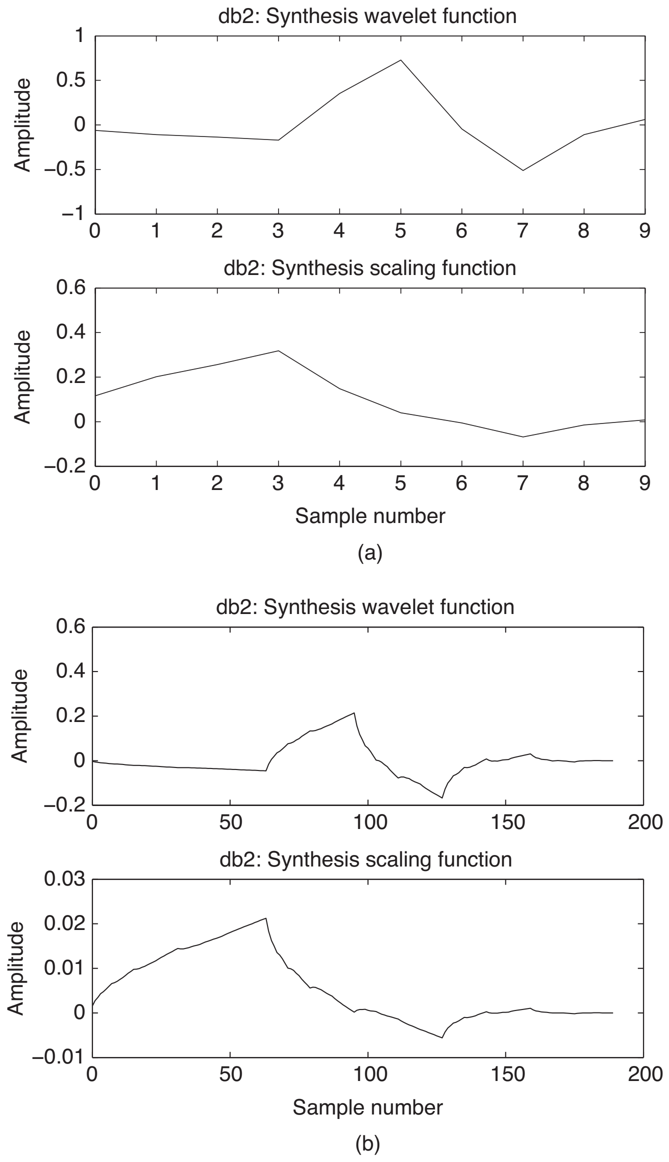
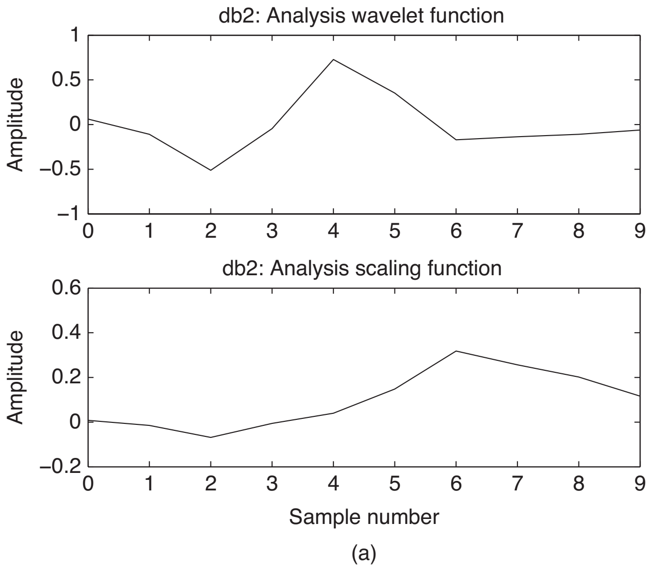


Figure 4.5 Construction of “db2” orthogonal wavelet: (a) reconstruction wavelet function after one iteration and (b) reconstruction scaling function after five iteration.



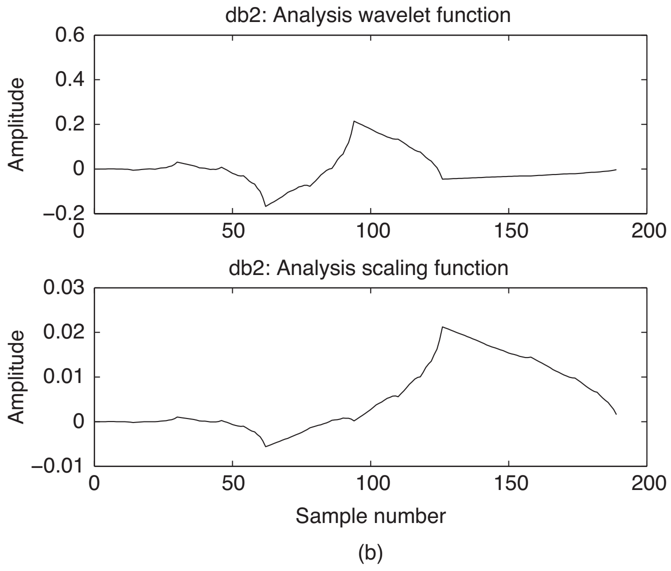
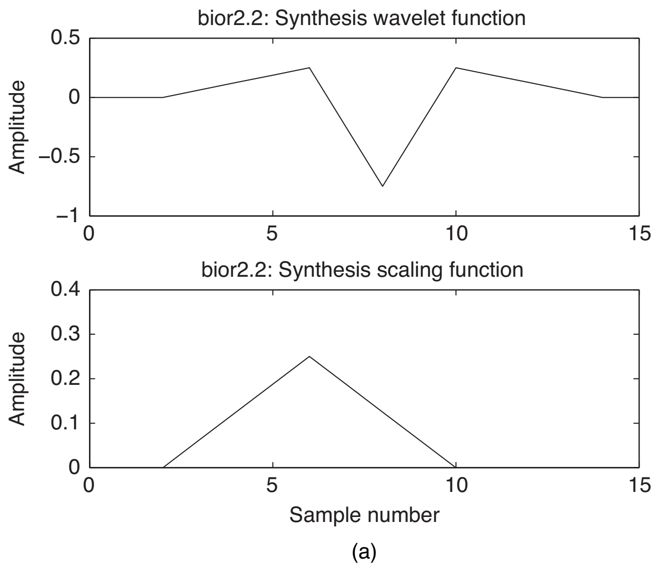


Figure 4.6 Construction of "db2" orthogonal wavelet: (a) analysis wavelet function after one iteration and (b) analysis scaling function after five iteration.



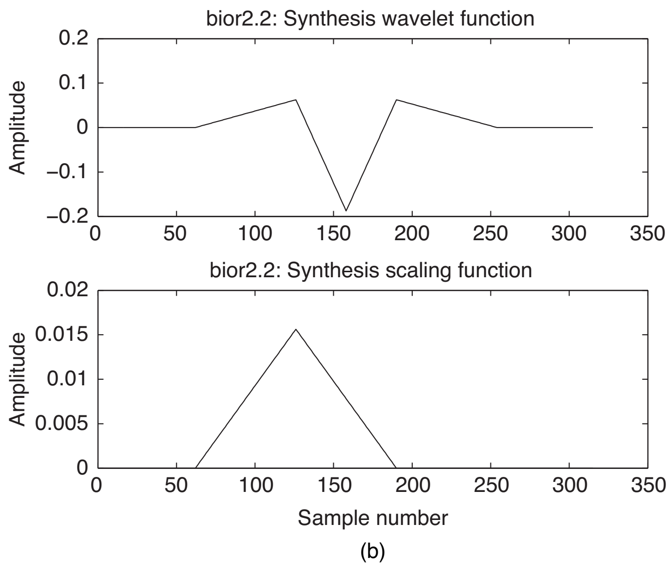
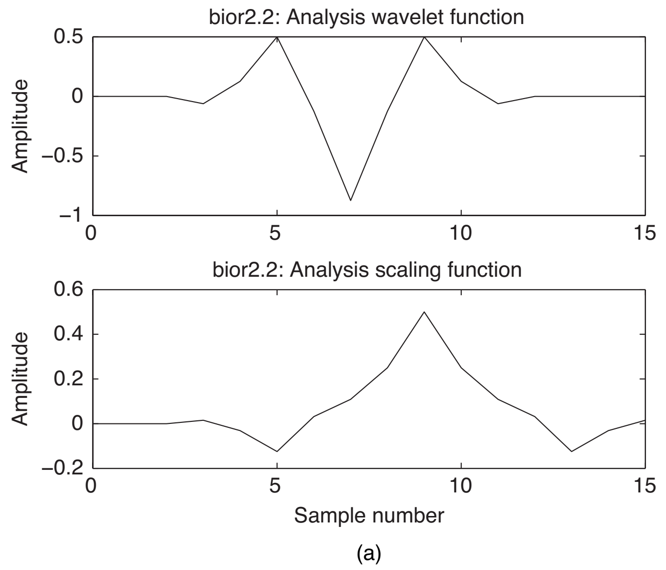


Figure 4.7 Construction of "bior2.2" biorthogonal wavelet: (a) reconstruction wavelet function after one iteration and (b) reconstruction scaling function after five iteration.



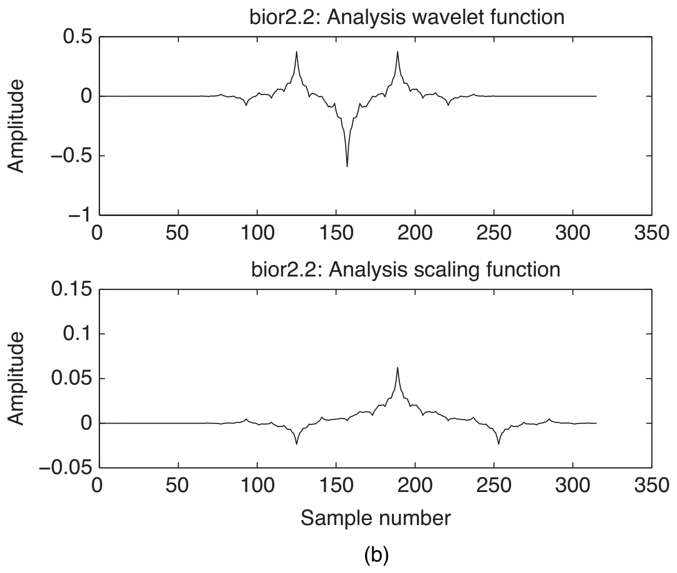


Figure 4.8 Construction of “bior2.2” biorthogonal wavelet: (a) analysis wavelet function after one iteration and (b) analysis scaling function after five iteration.

functions. These are shown in Figures 4.7a,b and Figures 4.8a,b, respectively. MATLAB code is listed below.

```
% Example4_2.m
% Builds ''db2'' and ''bior2.2'' wavelets.
% This script calls the function ''BuildWaveletShape'' to
% build the wavelets used in the analysis and synthesis
% portions of 2D DWT.
% Inputs to the ''BuildWaveletShape'' function are the
% name of the wavelet and number of iterations. For a
% list of wavelet names, refer to ''waveinfo'' in MATLAB.
%
% Build ''db2'' wavelet with 1 and 5 iterations.
BuildWaveletShape('db2',1);
BuildWaveletShape('db2',5);
% Next build ''bior2.2'' wavelet with 1 and 5 iterations.
BuildWaveletShape('bior2.2',1);
BuildWaveletShape('bior2.2',5);
```

```
function BuildWaveletShape(WaveletName,Iter)
% BuildWaveletShape(WaveletName,Iter)
% Builds the wavelet and scaling function for the wavelet
% WaveletName using Iter number of
% iterations.
% WaveletName can be one of 'db2','db4','db6', 'bior5.5', etc.


if ~exist('WaveletName','var')
  WaveletName = 'rbio5.5';
end
if ~exist('Iter','var')
  Iter = 1;
end
if nargin < 1
  error('at least one argument is required');
end
%
% Get the lowpass and highpass filters for the specified
  wavelet
[LP_D,HP_D] = wfilters(WaveletName,'d');
% upsample H and convolve it with L repeatedly Iter # times
% to obtain the wavelet function
for i = 1:Iter
  Hu = dyadup(HP_D,2);
  h = conv(Hu,LP_D);
  HP_D = h;
end
HP_D = h;
% upsample L and convolve it with L repeatedly Iter # times
% to obtain the scaling function
L1 = LP_D;
for i = 1:Iter
  Lu = dyadup(L1,2);
  L2 = conv(Lu,LP_D);
  L1 = L2;
end
LP_D = L2/sum(L2);% normalize the coefficients so the sum = 1
%
% Repeat the procedure for the wavelet
[LP_R,HP_R] = wfilters(WaveletName,'r');
clear h Hu
for i = 1:Iter
  Hu = dyadup(HP_R,2);
  h = conv(Hu,LP_R);
  HP_R = h;
end
HP_R = h;
L1 = LP_R;
for i = 1:Iter
  Lu = dyadup(L1,2);
  L2 = conv(Lu,LP_R);
  L1 = L2;
end
LP_R = L2/sum(L2);
%
N = length(LP_R); % length of wavelet
%
figure,subplot(2,1,1),plot(0:N-1,HP_R,'k')
ylabel('Amplitude')
title([WaveletName ' :' 'Synthesis' ' Wavelet function'])
subplot(2,1,2),plot(0:N-1,LP_R,'k')
ylabel('Amplitude')
title([WaveletName ' :' 'Synthesis' ' Scaling function'])
xlabel('Sample #')
%
figure,subplot(2,1,1),plot(0:N-1,HP_D,'k')
ylabel('Amplitude')
title([WaveletName ' :' 'Analysis' ' Wavelet function'])
subplot(2,1,2),plot(0:N-1,LP_D,'k')
xlabel('Sample #')
ylabel('Amplitude')
title([WaveletName ' :' 'Analysis' ' Scaling function'])
```

4.7 TWO-DIMENSIONAL DWT

When the two-dimensional (2D) DWT is separable, we can implement the 2D DWT in a row–column fashion. Even though one can implement the true 2D DWT to exploit the 2D correlation in an image, it is more efficient to implement the 2D DWT using 1D DWT because (a) a whole lot of 1D wavelet design materials exist in the literature and (b) fast implementation of 1D DWT is also available. We will, therefore, focus our attention on the implementation of 2D DWT of an image using 1D DWT in a row–column fashion.

Starting with the given image of size $N \times N$ pixels, we first filter it row by row by the filter $h_0[-n]$ and $h_1[-n]$, and retain every other sample at the filter outputs. This gives a set of two DWT coefficients each of size $N \times \frac{N}{2}$. Next, the DWT coefficients of the filter $h_0[-n]$ are filtered again along each column by the same two filters and subsampled by 2 to yield two other sets of DWT coefficients of each size $\frac{N}{2} \times \frac{N}{2}$. Finally, the DWT coefficients due to $h_1[-n]$ are filtered along the columns by the same two filters and subsampled to give two other sets of DWT coefficients of each size $\frac{N}{2} \times \frac{N}{2}$. We thus have one level of DWT of the image in question. Figure 4.9a depicts a one-level 2D DWT of an image of size $N \times N$ pixels. The four different DWT coefficients are labeled $y_{\text{LL1}}[m,n]$, $y_{\text{HL}_1}[m,n]$, $y_{\text{LH}_1}[m,n]$, and $y_{\text{HH}_1}[m,n]$. The suffix LL stands for the output of the $h_0[-n]$ filters in both dimensions, HL for the output of $h_1[-n]$ along the rows first and then of $h_0[-n]$ along the column, LH for the output of $h_0[-n]$ first and then of $h_1[-n]$, and finally HH stands for the outputs of $h_1[-n]$ along both dimensions. The suffix 1 refers to the level number. The labeling of the four DWT coefficients is shown in Figure 4.9b. The inverse 2D DWT of the DWT coefficients to reconstruct the image is shown in block diagram form in Figure 4.9c.

To obtain a second-level 2D DWT, one typically applies 2D DWT on the LL_1 coefficients using the same set of filters. This can be repeated iteratively. This is known as the tree-structured 2D DWT. Or, one can apply the computation of 2D DWT on each component at level 1 to obtain a level 2 *full tree* DWT.

Example 4.3 Compute the 2D DWT of the cameraman image up to two levels and display the approximation and detail coefficients as a single image. Apply 2D DWT on the first-level approximation coefficients to obtain the second-level DWT.

Solution MATLAB provides two functions to compute the 2D DWT of an image—"dwt2" which computes one level of 2D DWT and "wavedec2" to compute multilevel 2D DWT. The function "wavedec2" computes the 2D DWT in a tree structure. Since we want two levels, let us use the function "wavedec2." Let us use "db2" as the wavelet for this example.

"wavedec2" returns level-N 2D DWT coefficients along with the size of each component. We will use this feature to place all the DWT coefficients in a single array for display purpose. Finally, we will use the function "wcodemat" to enhance

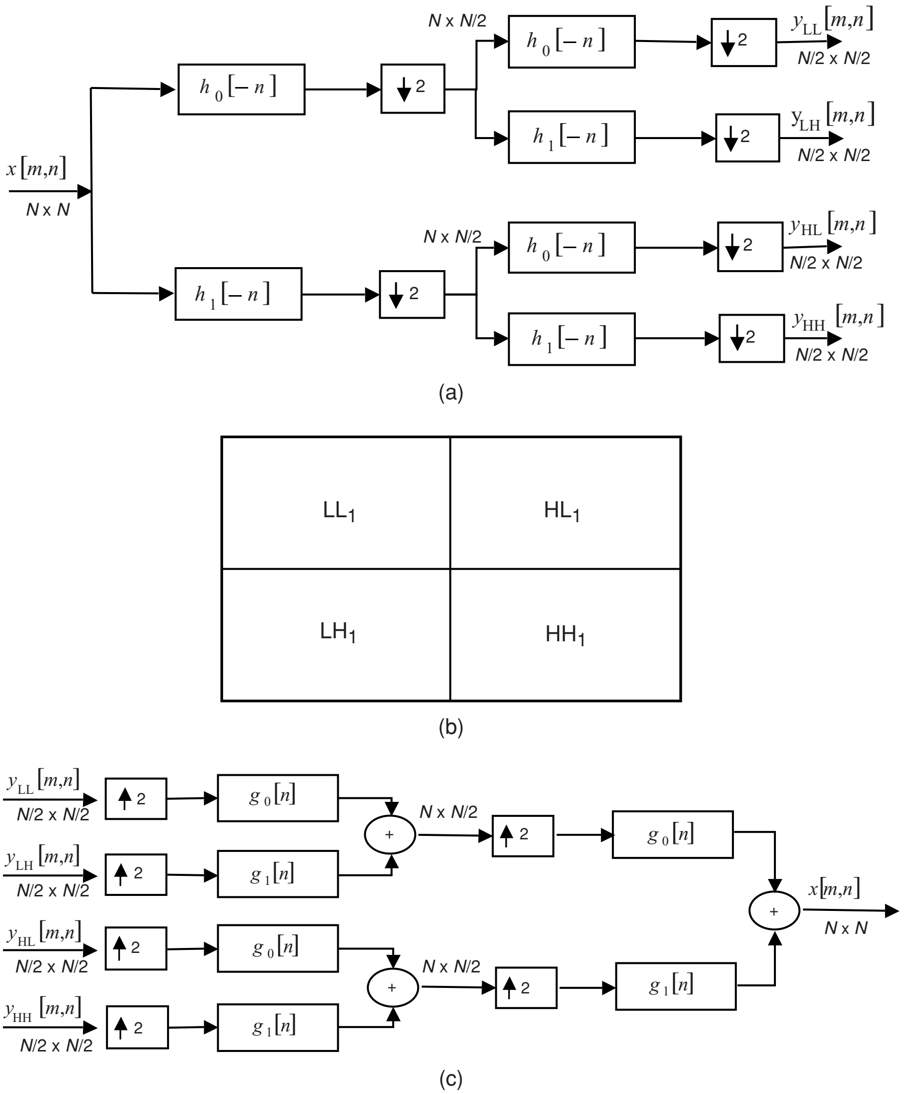


Figure 4.9 Computation of a one-level 2D DWT via subband coding scheme: (a) forward 2D DWT, (b) approximation and detail coefficient placement, and (c) inverse 2D DWT.

the visual quality of the displayed image. Figure 4.10 shows the two-level DWT of the cameraman image. The first level consists of LL, HL, LH, and HH coefficients. The HL coefficients correspond to highpass in the horizontal direction and lowpass in the vertical direction. Thus, the HL coefficients follow horizontal edges more than vertical edges. The LH coefficients follow vertical edges because they correspond to highpass in the vertical direction and lowpass in the horizontal direction. Finally,

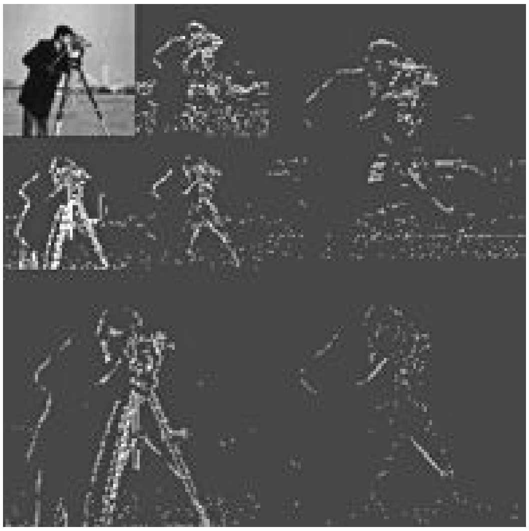


Figure 4.10 A two-level 2D DWT of the cameraman image of Example 4.3.

the HH coefficients preserve diagonal edges. The MATLAB code for this example is listed below.

```
% Example4_3.m
% Computes the Two-dimensional Discrete Wavelet Transform
% of an image up to two levels using ''db2'' wavelet. The
% DWT coefficients are placed in a large array for display
% purpose. The innermost 4 quadrants correspond to
% LL,HL,LH, and HH coefficients at level 2.
% the three outer quadrants correspond to HL,LH, and HH
% coefficients at level 1.
%
inFile = 'cameraman.tif';
%inFile = 'barbara.tif';
%inFile = 'liftingbody.png';
A = imread(inFile);
[Height,Width,Depth] = size(A);
if Depth = = 1
  f = double(A);
else
  f = double(A(:,:,1));
end
% 2-level 2D DWT
[DWT,S] = wavedec2(f,2,'db2');
% array to store all DWT coefficients for display
y = zeros(2*S(1,1)+S(2,1),2*S(1,2)+S(2,2));
```

```
% Extract approximation & detail coefficients @ level 2 and
% Place them in the large single array y for display
LL2 = appcoef2(DWT,S,2,'db2');
[HL2,LH2,HH2] = detcoef2('all',DWT,S,2);
y(1:S(1,1),1:S(1,2)) = LL2;
y(1:S(1,1),S(1,2)+1:2*S(1,2)) = HL2;
y(S(1,1)+1:2*S(1,1),1:S(1,2)) = LH2;
y(S(1,1)+1:2*S(1,1),S(1,2)+1:2*S(1,2)) = HH2;
% Now extract detail coefficients @ level 1 and
% Place them in the large single array y for display
[HL1,LH1,HH1] = detcoef2('all',DWT,S,1);
y(1:S(3,1),2*S(1,2)+1:2*S(1,2)+S(3,2)) = HL1;
y(2*S(1,1)+1:2*S(1,1)+S(3,1),1:S(3,2)) = LH1;
y(2*S(1,1)+1:2*S(1,1)+S(3,1),2*S(1,2)+1:2*S(1,2)+S(3,2)) = HH1;
% use wcodemat for visual enhancement
figure,imshow(wcodemat(y,32,'m',1),[]), title('2-Level
  2D DWT')
```

4.8 ENERGY COMPACTION PROPERTY

From a compression point of view, energy compacted into a few DWT coefficients will yield a high degree of image compression. We will demonstrate here how the DWT compacts energy in its coefficients by examples rather than by analytical means. We will also compute the histograms of the amplitudes of the DWT coefficients at different scales and orientations.

Example 4.4 Read an 8 bpp intensity image, compute its 2D DWT to three levels using the orthogonal wavelet "db2" and determine the percentage of energy in each scale and orientation. Also, compute the histograms of the amplitudes of the DWT coefficients at all three levels. Compare the percentage of energy in the coefficients with the orthogonal wavelets "haar," "db8," "coif2," and "sym2" as well as the biorthogonal wavelet "bior2.2."

Solution MATLAB provides a function "wenergy2" to compute the percentage of total energy in the DWT coefficients at different levels and orientations. First, we compute the 2D DWT of the cameraman image and then compute the energies in the approximation and detail coefficients. A listing of the MATLAB code follows.

The orthogonal wavelets do better than the biorthogonal wavelet in energy compaction as seen from Table 4.3. We also observe that almost all of the energy resides in the approximation coefficients, which indicates that a high degree of compression is achievable. The histograms for the DWT coefficients at the three levels for the "db2" wavelet are shown in Figures 4.11–4.13. From the figures, we notice that all the detail coefficients have histograms that resemble the double-sided exponential distribution. A similar phenomenon is seen for the biorthogonal case (Figures 4.14–4.16).

Table 4.3 Percentage of total energy in the DWT coefficients for the *cameraman* image

Level and Orientation	"db2"	"db8"	Haar	"coif2"	"sym2"	"bior2.2"
LL3	97.06	98.46	95.92	97.99	97.06	96.50
HL3	0.2020	0.0973	0.3392	0.1343	0.2020	0.1430
LH3	0.4044	0.1820	0.5733	0.2561	0.4044	0.2762
HH3	0.0709	0.0359	0.1094	0.0468	0.0709	0.0283
HL2	0.2349	0.1387	0.4566	0.1926	0.2349	0.3186
LH2	0.6644	0.3174	0.7827	0.3644	0.6644	0.6490
HH2	0.1107	0.0557	0.1437	0.0743	0.1107	0.1180
HL1	0.4518	0.2445	0.4891	0.2726	0.4518	0.4438
LH1	0.6691	0.3947	0.9946	0.5793	0.6691	1.3000
HH1	0.1364	0.0744	0.1889	0.0917	0.1364	0.2188

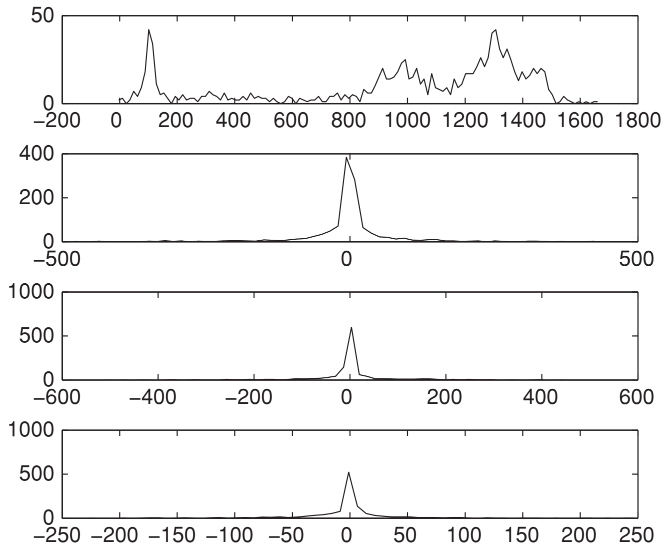


Figure 4.11 Histograms of the level-3 2D "db2" DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the scaling coefficients LL_3, the one below the top plot is the histogram of the detail coefficients HL_3, the one above bottom plot is the histogram of the detail coefficients LH_3, and the bottom plot is the histogram of the detail coefficients HH_3.

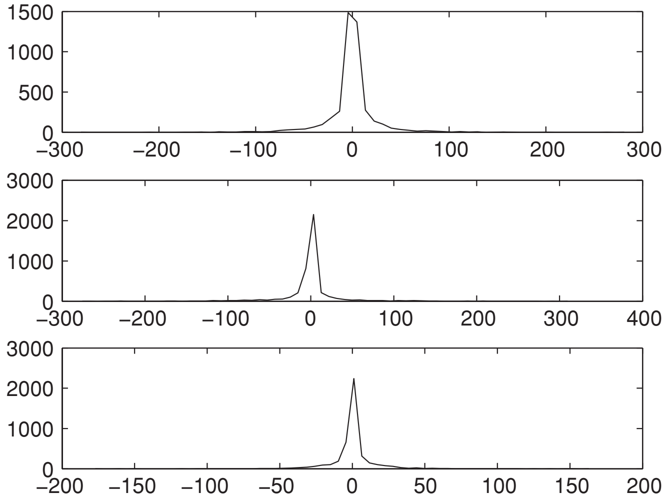


Figure 4.12 Histograms of the level-2 “db2” 2D DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the detail coefficients HL_2, the middle plot is the histogram of the detail coefficients LH_2, and the bottom plot is the histogram of the detail coefficients HH_2.

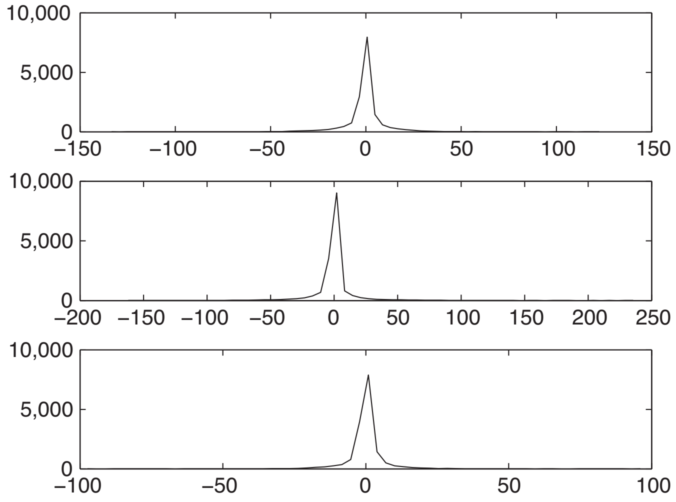


Figure 4.13 Histograms of the level-2 “db2” 2D DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the detail coefficients HL_1, the middle plot is the histogram of the detail coefficients LH_1, and the bottom plot is the histogram of the detail coefficients HH_1.

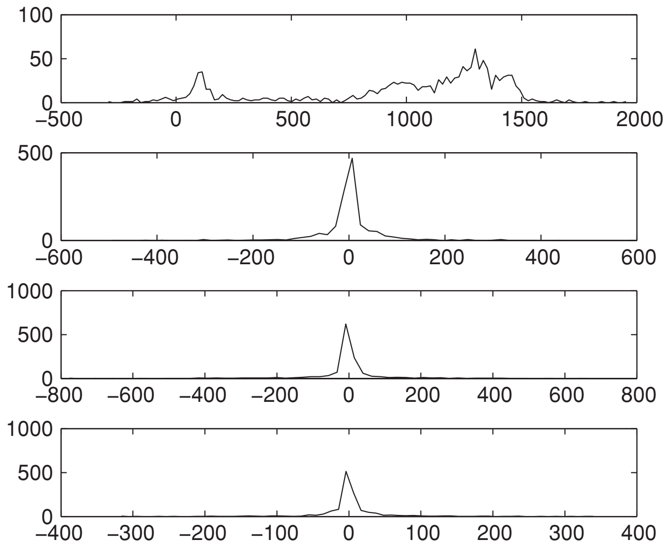


Figure 4.14 Histograms of the level-3 2D "bior2" DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the scaling coefficients LL_3, the one below the top plot is the histogram of the detail coefficients HL_3, the one above bottom plot is the histogram of the detail coefficients LH_3, and the bottom plot is the histogram of the detail coefficients HH_3.

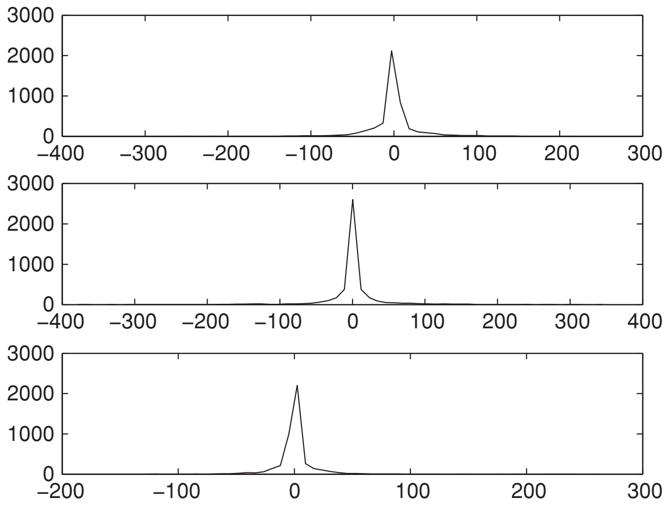


Figure 4.15 Histograms of the level-2 "bior2" 2D DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the detail coefficients HL_2, the middle plot is the histogram of the detail coefficients LH_2, and the bottom plot is the histogram of the detail coefficients HH_2.

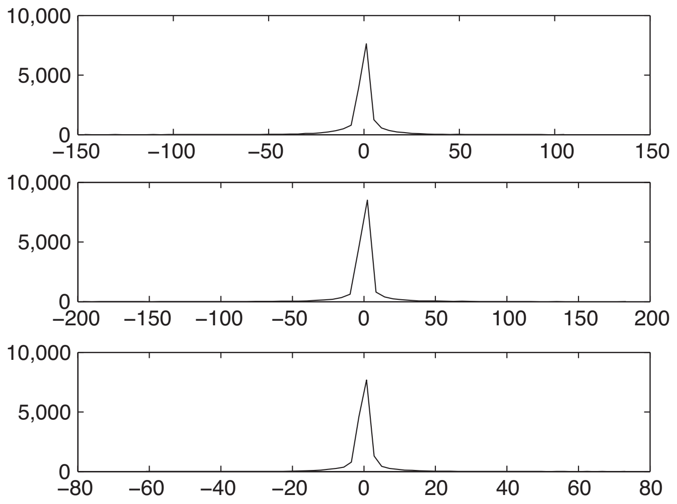


Figure 4.16 Histograms of the level-2 “bior2” 2D DWT coefficients of the cameraman image of Example 4.4. The top plot is the histogram of the detail coefficients HL_1, the middle plot is the histogram of the detail coefficients LH_1, and the bottom plot is the histogram of the detail coefficients HH_1.

```
% Example4_4.m
% Computes the Two-dimensional Discrete Wavelet Transform
% of an image up to three levels using orthogonal or
% biorthogonal wavelet.
% Then computes the energy in the coefficients at different
% levels and orientations as a percentage of total
  energy in the
% image. It also computes the histogram of the coefficients.
% dwtType = ''ortho'' or ''biortho''
% WaveletName = ''db2'' or ''bior2.2''. Type help ''wfilters''
% for the possible wavelets.

inFile = 'cameraman.tif';
%inFile = 'barbara.tif';
%inFile = 'liftingbody.png';
A = imread(inFile);
[Height,Width,Depth] = size(A);
if Depth == 1
   f = double(A);
else
   f = double(A(:,:,1));
end
dwtType = 'ortho';
WaveletName = 'db2';
```

```
switch dwtType
  case 'ortho'
    % Energy compaction in orthogonal DWT
    % 3-level 2D DWT
    [DWT,S] = wavedec2(f,3,WaveletName);
    % Extract the approximation & detail coefficients at
      all levels
    LL3 = appcoef2(DWT,S,3,WaveletName);
    [HL3,LH3,HH3] = detcoef2('all',DWT,S,3);
    [HL2,LH2,HH2] = detcoef2('all',DWT,S,2);
    [HL1,LH1,HH1] = detcoef2('all',DWT,S,1);
    % compute the percentage of energy in the coefficients
    [Ea,Eh,Ev,Ed] = wenergy2(DWT,S);
    sprintf('Energy in LL3 = %5.2f%%',Ea)
    sprintf('\t\t\t Energy in \nHL3 = %5.4f%% \t HL2 = %5.4f%%
         \t HL1 = %5.4f%% \n',Eh)
    sprintf('\t\t\t Energy in \nLH3 = %5.4f%% \t LH2 = %5.4f%%
         \t LH1 = %5.4f%% \n',Ev)
    sprintf('\t\t\t Energy in \nHH3 = %5.4f%% \t HH2 = %5.4f%%
         \t HH1 = %5.4f%% \n',Ed)
    [Na,Ba] = hist(LL3(:),128);
    [Nh3,Bh3] = hist(HL3(:),64);
    [Nv3,Bv3] = hist(LH3(:),64);
    [Nd3,Bd3] = hist(HH3(:),64);
    figure,subplot(4,1,1),plot(Ba,Na,'k')
    title(['Histo. of Approx. & detail coefficients @
        Level 3: ' WaveletName])
    subplot(4,1,2),plot(Bh3,Nh3,'k')
    subplot(4,1,3),plot(Bv3,Nv3,'k')
    subplot(4,1,4),plot(Bd3,Nd3,'k')
    [Nh2,Bh2] = hist(HL2(:),64);
    [Nv2,Bv2] = hist(LH2(:),64);
    [Nd2,Bd2] = hist(HH2(:),64);
    figure,subplot(3,1,1),plot(Bh2,Nh2,'k')
    title(['Histo. of detail coefficients @ Level 2:
        ' WaveletName])
    subplot(3,1,2),plot(Bv2,Nv2,'k')
    subplot(3,1,3),plot(Bd2,Nd2,'k')
    [Nh1,Bh1] = hist(HL1(:),64);
    [Nv1,Bv1] = hist(LH1(:),64);
    [Nd1,Bd1] = hist(HH1(:),64);
    figure,subplot(3,1,1),plot(Bh1,Nh1,'k')
    title(['Histo. of detail coefficients @ Level 1:
        ' WaveletName])
    subplot(3,1,2),plot(Bv1,Nv1,'k')
    subplot(3,1,3),plot(Bd1,Nd1,'k')
  case 'biortho'
    % Energy compaction in orthogonal DWT
    % 3-level 2D DWT
    [DWT,S] = wavedec2(f,3,'bior2.2');
    % Extract the approximation & detail coefficients at
        all levels
    LL3 = appcoef2(DWT,S,3,'db2');
    [HL3,LH3,HH3] = detcoef2('all',DWT,S,3);
    [HL2,LH2,HH2] = detcoef2('all',DWT,S,2);
    [HL1,LH1,HH1] = detcoef2('all',DWT,S,1);
    % compute the percentage of energy in the coefficients
    [Ea,Eh,Ev,Ed] = wenergy2(DWT,S);
    sprintf('Energy in LL3 = %5.2f%%',Ea)
    sprintf('\t\t\t Energy in \nHL3 = %5.4f%% \t HL2 = %5.4f%%
        \t HL1 = %5.4f%% \n',Eh)
    sprintf('\t\t\t Energy in \nLH3 = %5.4f%% \t LH2 = %5.4f%%
        \t LH1 = %5.4f%% \n',Ev)
    sprintf('\t\t\t Energy in \nHH3 = %5.4f%% \t HH2 = %5.4f%%
        \t HH1 = %5.4f%% \n',Ed)
    [Na,Ba] = hist(LL3(:),128);
    [Nh3,Bh3] = hist(HL3(:),64);
    [Nv3,Bv3] = hist(LH3(:),64);
    [Nd3,Bd3] = hist(HH3(:),64);
    figure,subplot(4,1,1),plot(Ba,Na,'k')
    title(['Histo. of Approx. & detail coefficients @
     Level 3: ' WaveletName])
    subplot(4,1,2),plot(Bh3,Nh3,'k')
    subplot(4,1,3),plot(Bv3,Nv3,'k')
    subplot(4,1,4),plot(Bd3,Nd3,'k')
    [Nh2,Bh2] = hist(HL2(:),64);
    [Nv2,Bv2] = hist(LH2(:),64);
    [Nd2,Bd2] = hist(HH2(:),64);
    figure,subplot(3,1,1),plot(Bh2,Nh2,'k')
    title(['Histo. of detail coefficients @ Level 2:
        ' WaveletName])
    subplot(3,1,2),plot(Bv2,Nv2,'k')
    subplot(3,1,3),plot(Bd2,Nd2,'k')
    [Nh1,Bh1] = hist(HL1(:),64);
    [Nv1,Bv1] = hist(LH1(:),64);
    [Nd1,Bd1] = hist(HH1(:),64);
    figure,subplot(3,1,1),plot(Bh1,Nh1,'k')
    title(['Histo. of detail coefficients @ Level 1:
        ' WaveletName])
    subplot(3,1,2),plot(Bv1,Nv1,'k')
    subplot(3,1,3),plot(Bd1,Nd1,'k')
end
```

Table 4.4 Le Gall's $\frac{5}{3}$ integer wavelets

	Analysis		Synthesis	
n	$h_0[n]$	$h_1[n]$	$g_0[n]$	$g_1[n]$
0	6/8	1	1	6/8
± 1	2/8	$-1/2$	1/2	$-2/8$
± 2	$-1/8$			$-1/8$

4.9 INTEGER OR REVERSIBLE WAVELET

The 1D and 2D DWT that we have discussed so far will give a perfect reconstruction of the signal or image as long as the arithmetic carried out is with full precision. However, in practice, only finite precision arithmetic is possible. Additionally, most embedded hardware uses integer arithmetic. In such cases, perfect image reconstruction using the DWT that has been discussed so far is not possible. When mathematically lossless compression is desired, one must be able to carry out the forward and inverse DWT of an image without introducing any arithmetic errors. This has been made possible with the so-called *integer* or *reversible* DWT. One such wavelet filter bank used in the motion JPEG2000 standard is called *Le Gall's* $\frac{5}{3}$ wavelet [25] and is listed in Table 4.4. These wavelets are also called *Cohen–Daubechies* $\frac{5}{3}$ wavelets. As can be seen from the table, these filter banks realize the biorthogonal wavelets (filters have symmetry and so have linear phase response).

4.10 SUMMARY

Wavelet transform, unlike the Fourier transform, has the ability to capture both short-term and long-term behaviors of a signal or image. This property has proved very useful in audio and image compression. In this chapter, we have discussed both CWT and DWT. From an image compression point of view, computational efficiency is an important factor to be considered. Fortunately, DWT can be implemented as a fast algorithm via subband coding scheme. Typically, DWT starts with the lowest scale (lowest level) corresponding to the given image and computes the DWT coefficients by iterating the filtering and subsampling processes. This results in one approximation and many detail coefficients. We observed from examples that most of the energy of the DWT coefficients is packed in the approximation coefficients, which is a factor in achieving high compression.

Wavelets come in two flavors—orthogonal and biorthogonal. Orthogonal wavelets cannot be implemented with linear phase FIR filters, while it is possible with biorthogonal wavelets. Both wavelets can be obtained by iterating the subband coding filter bank, and we showed the procedure and results by an example. In general, implementing the DWT with finite-precision arithmetic introduces irrecoverable error even without any quantization of the coefficients. But when lossless compression using DWT is desired, one must be able to carry out the computation with finite precision integer arithmetic for real-time processing. Such a wavelet is called an integer or reversible wavelet. An example is Le Gall's $\frac{5}{3}$ integer wavelet where all the filter coefficients are such that only integer multiplication by bit shifting are required in the forward and inverse DWT. Hence, the DWT is exactly reversible.

Having discussed both block transform and wavelet transform, which are the compression vehicles, we will next consider the coding of symbols in the next chapter. In transform coding or coding in the wavelet domain, the quantized coefficients that form the symbols must be coded losslessly both for transmission and storage. We will take up the topic of lossless coding in Chapter 5.

REFERENCES

1. Y. Meyer, "L'analyses par Ondelettes," *Pour la Science*, 1987.
2. Y. Meyer, *Ondelettes et ope'rateurs*, Hermann, Paris, 1990.
3. Y. Meyer, *Wavelets and Applications,* Proceedings of the Intl. Conf., Marseille, France, Mason, Paris, and Springer-Verlag, Berlin, 1992.
4. I. Daubechies, "Orthoginal bases of compactly supported wavelets," *Comm. Pure Appl. Math.*, 41, 909–996, 1988.
5. I. Daubechies, "The wavelet transform, time-frequency localization and signal analysis," *IEEE Trans. Inf. Theory*, 36 (5), 961–1005, 1990.
6. I. Daubechies, *Ten Lectures on Wavelets*, SIAM, Philadelphia, PA, 1992.
7. I. Daubechies, "Orthonormal bases of compactly supported wavelets II, variations on a theme," *SIAM J. Math. Anal.*, 24 (2), 99–519, 1993.
8. I. Daubechies, "Where do we go from here—a personal point of view," *Proc. IEEE*, 84 (4), 510–513, 1996.
9. P. Goupillaud, A. Grossman, and J. Morlet, "Cycle-octave and related transforms in seismic signal analysis," *Geoexploration*, 23, 85–102, 1984/1985.
10. A. Grossman and J. Morlet, "Decomposition of Hardy functions into square integrable wavelets of constant shape," *SIAM J. Math. Anal.*, 15 (4), 723–736, 1984.
11. M. Vetterli and J. Kovacevic, *Wavelets and Subband Coding*, Prentice Hall, Englewood Cliffs, NJ, 1995.
12. C. S. Burrus, R. A. Gopinath, and H. Guo, *Introduction to Wavelets and Wavelet Transforms*, Prentice Hall, Upper Saddle River, NJ, 1998.
13. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd edition, Prentice Hall, Upper Saddle River, NJ, 2002.
14. S. Mallat, "A compact multiresolution representation: the wavelet model," Proc. IEEE Computer Society Workshop on Computer Vision, *IEEE Computer Society Press*, Washington, DC, pp. 2–7, 1987.
15. S. Mallat, "A theory of multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-11, 674–693, 1989.
16. S. Mallat, *A Wavelet Tour of Signal Processing*, Academic Press, New York, 1998.
17. R. E. Crochiere, S. A. Weber, and J. L. Flanagan, "Digital coding of speech in sub-bands," *Bell Syst. Tech. J.*, 55 (8), 1069–1085, 1976.
18. A. Croisier, D. Esteban, and C. Galand, *Perfect Channel Splitting by Use of Interpolation/Decimation/Tree Decomposition Techniques*, Int. Conf. Inform. Sci. Syst., Patras, Greece, pp. 443–446, August 1976.

19. M. J. T. Smith and T. P. Barnwell III, “A procedure for designing exact reconstruction filter banks for tree structured sub-band coders,” IEEE Int. Conf. Acoust., Speech Signal Proc., San Diego, CA, March 1984.
20. M. J. T. Smith and T. P. Barnwell III, “Exact reconstruction for tree-structured subband coders,” *IEEE Trans. Acoust. Signal Speech Proc.*, 34 (3), 431–441, 1986.
21. F. Mintzer, “Filters for distortion-free two-band multirate filter banks,” *IEEE Trans. Acoust. Signal Speech Proc.*, 33 (3), 626–630, 1985.
22. P. P. Vaidyanathan, “Quadrature mirror filter banks, M-band extensions and perfect reconstruction techniques,” *IEEE ASSP Mag.*, 4 (3), 4–20, 1987.
23. P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, Englewood Cliffs, NJ, 1993.
24. T. Q. Nguyen and P. P. Vaidyanathan, “Two-channel perfect reconstruction FIR QMF structures which yield linear phase analysis and synthesis filters,” *IEEE Trans. Acoust. Signal Speech Proc.*, 37 (5), 676–690, 1989.
25. D. Le Gall and A. Tabatabai, “Subband coding of digital images using symmetric short kernel filters and arithmetic coding techniques,” Proc. IEEE Int. Conf. ASSP, New York, Vol. 2, pp. 761–764, April 1988.

PROBLEMS

4.1. Determine the coefficients of the wavelet series expansion (equation 4.7) of the signal $f(t) = 1 - t, 0 \le t \le 1$ up to two levels.

4.2. Show that Daubechies’s wavelet “db8” is orthonormal. Compute a two-level 2D DWT of a real intensity image using the wavelet “db8” and show that the energy is conserved in the DWT space.

4.3. Find the scaling and wavelet functions corresponding to Daubechies’s “db2.”

4.4. Compare the energy compaction property of the wavelets “db2,” “bior2.2,” “rbio2.2,” and Haar using a real intensity image with three levels. Which is more efficient from the point of view of image compression and why?

4.5. Read a real intensity image and perform a two-level 2D DWT using “db4” and “rbio4.4” wavelets. Plot the histograms of the various subbands and discuss their shapes and properties, such as mean and variance.

5

LOSSLESS CODING

5.1 INTRODUCTION

The input to a lossless coder is a symbol and the output is a codeword, typically a binary codeword. We are familiar with ASCII codes for the English alphabet. In an image compression system, a symbol could be a quantized pixel or a quantized transform coefficient. Then, the codeword assigned to the symbol by the lossless coder is either transmitted or stored. If the lengths of all codewords assigned to the different symbols are fixed or constant, then the lossless coder is called a fixed rate coder. On the other hand, if the code lengths of the codewords are variable, then the coder is a variable-length coder. A *pulse code modulator* assigns equal length codewords to its input symbols. Later in the chapter, we will see that a Huffman coder assigns variable-length codewords to its input symbols. In order for a lossless coder to assign codewords to its input symbols, it must first have the codewords precomputed and stored. We will discuss a method to generate the codebook of a Huffman coder later in the chapter.

A lossless decoder performs the inverse operation, which is to output a symbol corresponding to its input—a codeword. Therefore, it is imperative that a lossless decoder has the same codebook as the encoder. The practicality of a lossless coder depends on the size of the codebook it requires to encode its input symbols as well as its encoding complexity. A Huffman coder may require a large codebook, while an *arithmetic* coder requires no codebook at all, though it may be more complex than the Huffman coder. The choice of a particular lossless coder depends not only on the above-mentioned two factors but also on achievable compression.

Before we describe various lossless encoding methods and their implementation, we must have a rudimentary knowledge of information theory. Information theory gives us a means to quantify a source of symbols in terms of its average information content as well as the theoretical limit of its achievable lossless compression. This is important because such quantitative means enable us to compare the performance of different compression schemes. Therefore, we will briefly describe a few basic definitions of information theory and then proceed with the lossless coders.

5.2 INFORMATION THEORY

Let us consider a source of information, such as a transform coder. The information source produces a symbol at a time from a set of symbols called the source *alphabet*. For instance, an intensity image may have 256 possible grayscale values and each pixel value may be considered as a symbol. Thus, the source alphabet in this case has 256 symbols. Since we do not know ahead of time which symbol the source will produce, we may only describe the occurrence of the source symbols in a probabilistic fashion. Thus, we have an information source that produces a symbol at a time from its alphabet of symbols with each symbol having a certain probability of occurrence. The quantity of information conveyed by a symbol to the receiver must be nonnegative, must be large if its probability is small, and small if its probability is large. This agrees with our intuitive notion of information conveyed by a message. When we hear someone saying that the sun rises from the east, we turn our deaf ear to it because it is certain that the sun rises from the east only. So, there is no information in this statement. On the other hand, if someone says that a meteor is approaching the earth, then our eyebrows raise. Why, because the news is not very common, and in fact, it is indeed very rare and therefore conveys a great deal of information. This verbal information, of course, is semantic, but we are only interested in quantitative measure of information for engineering design purposes.

5.2.1 Self-Information

Let a source of information consist of an alphabet of M symbols $\mathrm{A} = \{a_i, 1 \leq i \leq M\}$ with probabilities $P(a_i) = p_i, 1 \leq i \leq M$. Then, the self-information I_i associated with the ith symbol is defined as [1]

$$I_i = \log_2\left(\frac{1}{p_i}\right) = -\log_2(p_i) \text{ (bits)} \tag{5.1}$$

Since $0 \leq p_i \leq 1$, the information content of a symbol is positive and increases monotonically with decreasing probability. Incidentally, bits stand for *binary* digits. A highly unlikely event (occurrence of a symbol) carries a large number of bits, while the most often occurring event carries the least number of bits. One interpretation of equation (5.1) is that a source symbol with a probability of occurrence p_i needs I_i number of bits for its representation. Note that I_i is not necessarily an integer.

Table 5.1 Entropy of a few BW images

Image	Entropy (bits/pixel)
Cameraman	7.0097
Barbara	5.6211
Trevor sequence: mean of first 10 frames	6.7177
Table tennis sequence: mean of first 10 frames	6.5470
Claire sequence: mean of first 10 frames	6.4036

5.2.2 Source Entropy

Memoryless Source An information source is not meant to produce just one symbol and keep quiet forever. It, in fact, produces a sequence of symbols, which must be conveyed to a receiver. In such cases, it is important to know how many bits are needed to represent (or code) each symbol of the source on an average. Thus, entropy of a source is the average number of bits required to code the source symbols. Let us assume that a source of information has M symbols with individual probability p_i and $\sum_{i=1}^{M} p_i = 1$. Let us also assume that the occurrence of a symbol is independent of other symbols. Such a source is referred to as a *memoryless* source. Entropy of a memoryless source of information is mathematically defined as [1–3],

$$H = -\sum_{i=1}^{M} p_i \log_2 p_i \text{ (bits/symbol)} \tag{5.2}$$

When a source probability model is not available a priori, as in the case of the symbol probabilities of real images, we can determine the entropy of such a source from its histogram. The entropies of a few black and white (BW) images are listed in Table 5.1. We have already seen the cameraman and Barbara images. The other images, one from each of the Trevor, Table Tennis, and Claire sequences, are shown in Figures 5.1a–c, respectively. A MATLAB function to calculate the entropy of an

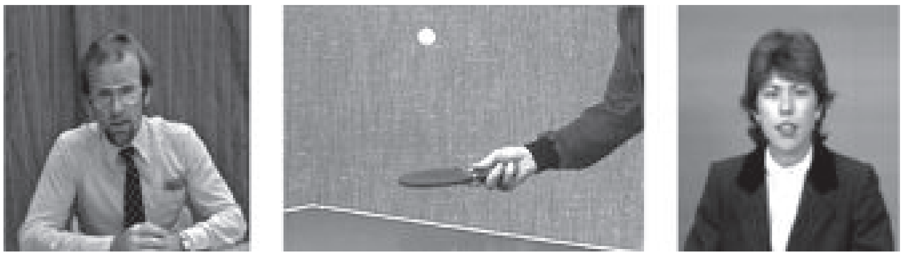


Figure 5.1 Images used in the calculation of entropy are listed in Table 5.1. (a) Frame 1 of Trevor sequence, (b) Frame 1 of Table tennis sequence, and (c) Frame 1 of Claire sequence.

image is listed below. One observation is that though the Barbara image has a lot of details, its entropy is only about 5.6 bpp. This is because its histogram has a lot of "holes," meaning that a lot of bins are empty.

```
function H = SourceEntropy(x)
% H = SourceEntropy(x)
% computes the entropy of the input intensity image x
% input image is assumed to be of type uint8



[Height,Width] = size(x);
[p,Binx] = imhist(x);% compute the histogram with
   256 bins.
%
p = p/(Height*Width);% normalize the histogram counts.
sprintf('Sum of hist. values = %g',sum(p))% verify
   that sum(p) = 1
figure,plot(Binx,p,'k')
xlabel('Pixel value'), ylabel('relative count')
% to avoid log(0), add a small positive value to p.
H = sum(-p.* log2(p+1e-08));% compute the entropy
```

The entropy of an information source is a maximum when the symbol probabilities are equal. If a source has M symbols with equal probabilities, then its entropy equals $\log_2 M$ bits/symbol. Thus, the source entropy is bounded by $0 \le H \le \log_2 M$. Let us consider a couple of examples to illustrate this statement.

Example 5.1 Consider a binary source ($M = 2$) with probabilities $p(m_1) = p$ and $p(m_2) = 1 - p$. Then, its entropy H is

$$H = p\log_2\left(\frac{1}{p}\right) + (1-p)\log_2\left(\frac{1}{1-p}\right) \tag{5.3}$$

When $p = 0$ or $p = 1$, the entropy is zero. When $p = \frac{1}{2}$, $H = 1$ bit/symbol. Thus, H is a convex function, as seen from Figure 5.2. The source *redundancy* is defined as the difference between the maximum value of the entropy and the entropy. The maximum value of the source entropy is sometimes known as the *capacity.* For instance, if $p = 0.2$, then $H = 0.7219$ bit/symbol. But the capacity is 1 bit/symbol. Therefore, the redundancy of this binary source is $1 - 0.7219 = 0.2781$ bit/symbol.

Example 5.2 Consider a memoryless source with 8 symbols. The symbol probabilities are 0.1, 0.2, 0.15, 0.3, 0.175, 0.02, 0.02, and 0.035. Calculate the source entropy, capacity, and redundancy.

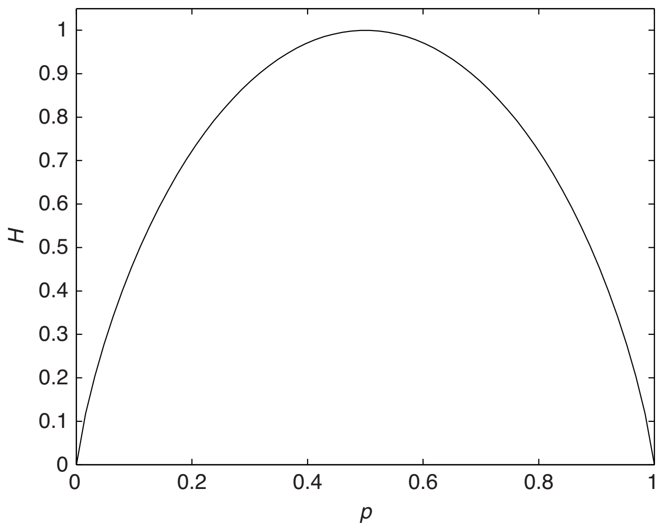


Figure 5.2 Entropy of a binary source.

Solution $H = -\sum_{i=1}^{8} p_i \log_2(p_i) = 2.5633$ bits/symbol. The capacity of the source is $C = \log_2 8 = 3$ bits/symbol. Therefore, the redundancy is 0.4367 bit/symbol.

Source with Memory In an information source with memory, the occurrence of a symbol at a given instant is dependent probabilistically on the occurrence of symbols at previous instants. In other words, there is a statistical dependency on the occurrence of successive symbols. This property is useful in achieving a higher compression. A suggestion by this statement is that we may be able to achieve a higher compression by grouping K symbols at a time rather than taking one symbol at a time.

Consider a source producing a sequence of alphabets $\{x_1, x_2, \ldots, x_n, \ldots\}$. Let us group K alphabets at a time to form a symbol $\mathbf{x}$ with a probability $p(\mathbf{x})$. Thus, we can consider any vector $\mathbf{x}$ as a realization of a random variable $\mathbf{X}$. The entropy of the length-K symbols is expressed as [4]

$$H_K(\mathbf{X}) = -\frac{1}{K} \sum_{\text{over all } \mathbf{x}} p(\mathbf{x}) \log_2(p(\mathbf{x})) \text{ (bits/symbol)} \tag{5.4}$$

The source entropy is obtained as the limiting value of $H_K(\mathbf{x})$, that is,

$$H(X) = \underbrace{\lim}_{K\to\infty} H_K(\mathbf{X}) \tag{5.5}$$

For a source with memory, the source entropy can also be expressed in terms of its conditional entropy as

$$H(\mathrm{X}) = \underbrace{\lim}_{K\to\infty} H\left(X_n \,|X_{n-1}, \ldots, \; X_{n-K+1}\right) \tag{5.6}$$

When comparing a memoryless source with a source with memory having the same alphabet and letter probabilities as the memoryless source, it can be shown in general that the entropy of the source with memory is less than that of the memoryless source [4], that is,

$$H_{\text{with memory}}(X) \prec H_{\text{memoryless}}(X) \tag{5.7}$$

The implication of equation (5.7) is that one can achieve a higher lossless compression for the source with memory by grouping letters to form symbols and then encoding the symbols. This is done routinely in Moving Picture Experts Group (MPEG) compression standard, where *run-length coding* (RLC) of the quantized *discrete cosine transform* (DCT) coefficients is used to form symbols that are then Huffman coded.

Example 5.3 A Markov source is a source with finite memory. It is said to have *states* with state transition probabilities. For instance, a first-order Markov source (or a binary Markov source) has two states, say, s_1 and s_2. Given that the current state is s_1, the probability of the system transitioning to the state s_2 denoted $p(s_2|\,s_1)$ is α. Similarly, the probability of the system transitioning from its current state s_2 to the state s_1 at the next instant is $p(s_1|\,s_2) = \beta$. Since there are only two states, $p(s_1|\,s_1) = 1 - \alpha$ and $p(s_2|\,s_2) = 1 - \beta$. Figure 5.3 is a graphical representation of a binary Markov source. The transition probabilities are expressed in a matrix called the *state transition matrix*. For the binary Markov source, it is given by

$$T = \begin{bmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{bmatrix} \tag{5.8}$$

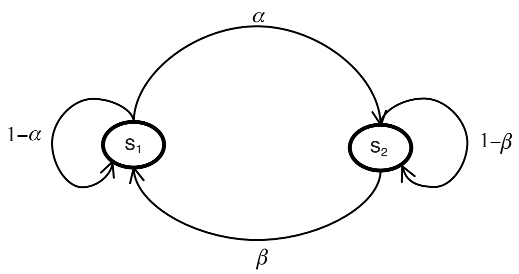


Figure 5.3 A graphical representation of a binary Markov source showing the state transitions with corresponding probabilities.

Let $\alpha = 0.2$ and $\beta = 0.3$. The individual state probabilities can be obtained from the following two equations:

$$\begin{aligned} p(s_1) &= p(s_1) \times (1 - p(s_2|\,s_1)) + p(s_2) \times p(s_1|\,s_2) \\ &= (1-\alpha)\,p(s_1) + \beta p(s_2) \end{aligned} \tag{5.9a}$$

$$p(s_1) + p(s_2) = 1 \tag{5.9b}$$

Solving for the individual probabilities, we find that $p(s_1) = 0.6$, and $p(s_2) = 0.4$. Next, we determine the entropy. The entropy can be expressed by

$$H(X) = P(s_1)\,H(X|\,s_1) + P(s_2)\,H(X|\,s_2) \tag{5.10}$$

where the conditional entropies are defined by

$$H(X|\,s_1) = -p(s_1|\,s_1)\log_2 p(s_1|\,s_1) - p(s_2|\,s_1)\log_2 p(s_2|\,s_1) \tag{5.11}$$

$$H(X|\,s_2) = -p(s_2|\,s_2)\log_2 p(s_2|\,s_2) - p(s_1|\,s_2)\log_2 p(s_1|\,s_2) \tag{5.12}$$

Using the appropriate numerical values, we obtain $H(X|\,s_1) = 0.7219$ and $H(X|\,s_2) = 0.8813$, and the source entropy $H(X) = 0.7857$ (bit/symbol). If we compare this with the entropy of a binary memoryless source with the same probabilities of 0.6 and 0.4 for $p(s_1)$ and $p(s_2)$, respectively, then we find that the memoryless binary source has the entropy of 0.971 bit/symbol! Thus, by having memory, the source entropy is reduced.

Shannon's Noiseless Coding Theorem

An important result of information theory is Shannon's noiseless coding theorem according to which it is possible to encode the symbols of an information source losslessly at a rate R as close to the source entropy as possible [5].

Shannon's noiseless coding theorem does not show how to achieve the theoretical limit of the source entropy. One possible method is to use variable-length entropy (lossless) coding to achieve the theoretical limit as close as possible.

5.2.3 Rate Distortion

The rate distortion theory provides a lower bound on achievable rate in bits per symbol for a given amount of distortion $R(D)$. The usual measure of distortion is the mean square error between the encoded and reconstructed samples. For a distortionless coder–decoder structure, the minimum achievable rate is the source entropy $R(0) = H(X)$. For a source that produces samples with continuum of amplitude whose probability density function is a zero-mean Gaussian function with variance

σ_x^2, the rate distortion function is given by [1]

$$R(D) = \max\left\{0, \quad \frac{1}{2}\log_2\frac{\sigma_x^2}{D}\right\} = \begin{cases} \frac{1}{2}\log_2\frac{\sigma_x^2}{D}, & 0 \le D \le \sigma_x^2 \\ 0, & D \succ \sigma_x^2 \end{cases} \tag{5.13}$$

When the distortion $D = \sigma_x^2$, there is no need to send any code for the compressed samples because we can assign zeros to the reconstructed samples and the distortion will be equal to the variance of the input samples. Therefore, the maximum value for the distortion is the signal variance. From equation (5.13), we can express the distortion in terms of the rate as given by

$$D(R) = \sigma_x^2 2^{-2R} \tag{5.14}$$

We can also express the signal-to-noise ratio (SNR) from equation (5.14) as

$$\text{SNR} = 10\log\left(\frac{\sigma_x^2}{D}\right) = R \times (20\log 2) \approx 6.02R\ (\text{dB}) \tag{5.15}$$

Thus, every bit increase in the rate yields a 6 dB improvement in SNR. Another way of looking at it is that every bit increase in the rate brings down the distortion (MSE) by a factor of 4. Figure 5.4 shows the plot of the distortion normalized to the signal variance versus rate as per equation (5.14).

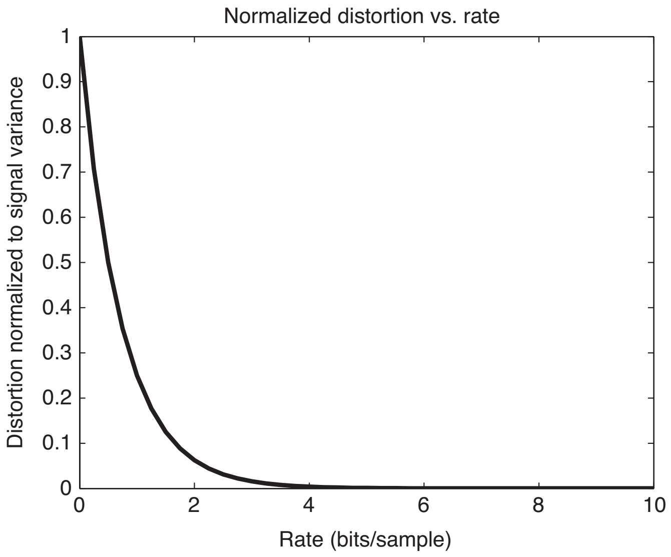


Figure 5.4 Distortion versus rate function of a Gaussian source.

If we encode N Gaussian random variables $\{x[1], x[2], \ldots, x[N]\}$ individually as in transform coding, and if the reconstructed values are $\{y[1], y[2], \ldots, y[N]\}$, then the average mean square distortion will be

$$D = \frac{1}{N}\sum_{j=1}^{N} E\left(|x[j] - y[j]|^2\right) \tag{5.16}$$

If the average distortion D in equation (5.16) is fixed, then the rate distortion function for the Gaussian random vector is given by [5]

$$R(D) = \frac{1}{N}\sum_{j=1}^{N} \max\left[0, \quad \frac{1}{2}\log_2\left(\frac{\sigma_j^2}{\theta}\right)\right] \tag{5.17}$$

where θ is determined from

$$D = \frac{1}{N}\sum_{j=1}^{N} \min\left[\theta, \quad \sigma_j^2\right] \tag{5.18}$$

In transform coding for instance, θ will be a threshold value and a coefficient value that exceeds the threshold will be quantized and a coefficient value below the threshold will be set to zero. The rate distortion function is useful when comparing practical compression systems.

5.3 HUFFMAN CODING

Huffman codes are variable-length codes and are optimum for a source with a given probability model [6]. Here, optimality implies that the average code length of the Huffman codes is closest to the source entropy. In Huffman coding, more probable symbols are assigned shorter codewords and less probable symbols are assigned longer codewords. The procedure to generate a set of Huffman codes for a discrete source is as follows.

Huffman Coding Procedure
Let a source alphabet consist of M letters with probabilities p_i, $1 \le i \le M$.

1. Sort the symbol probabilities in a descending order. Each symbol forms a leaf node of a tree.
2. Merge the two least probable branches (last two symbols in the ordered list) to form a new single node whose probability is the sum of the two merged branches. Assign a “0” to the top branch and a “1” to the bottom branch. Note that this assignment is arbitrary, and therefore, the Huffman codes are not unique. However, the same assignment rule must be maintained throughout.

3. Repeat step 2 until left with a single node with probability 1, which forms the root node of the Huffman tree.
4. The Huffman codes for the symbols are obtained by reading the branch digits sequentially from the root node to the respective terminal node or leaf.

The Huffman coding procedure is best explained by an example.

Example 5.4 For a source with discrete alphabet of eight letters whose probabilities are $P = [0.2\ 0.15\ 0.13\ 0.13\ 0.12\ 0.10\ 0.09\ 0.08]^T$, construct a Huffman code and calculate its average code length.

Solution The symbol probabilities are already in descending order, and therefore, let the symbols be numbered in the same order as the given symbol probabilities. Starting with the two least probable messages corresponding to the messages m_7 and m_8, generate the node whose probability equals the sum of 0.09 and 0.08. Assign a "0" to the top branch and a "1" to the bottom branch. Since the next two smallest probabilities are 0.1 and 0.12, form the node whose branches have these two probabilities and whose sum equals 0.22. Assign a "0" to the top branch and a "1" to the bottom branch. The next two probabilities to be combined are 0.13 and 0.13. Continuing further, we obtain the complete Huffman tree as shown in Figure 5.5.

Next, the codes for the messages are obtained by reading the digits from the root node to the terminal nodes sequentially. Thus, we get the Huffman codes for the letters as shown in Table 5.2. We can find the average code length $\overline{L}$ of the Huffman codes in Table 5.2 from

$$\overline{L} = \sum_{i=1}^{8} l_i p_i = 2.97 \text{ (bits/symbol)}$$

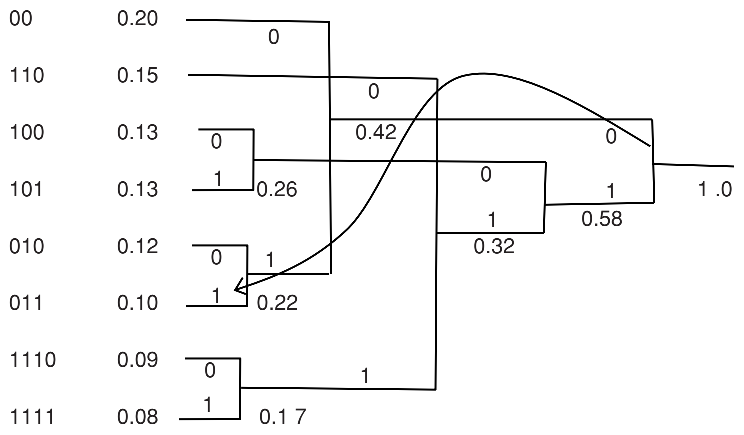


Figure 5.5 Construction of a Huffman tree for the probability vector in Example 5.4.

Table 5.2 Huffman codes for Example 5.4

Symbol	Probability	Huffman Code
1	0.20	00
2	0.15	110
3	0.13	100
4	0.13	101
5	0.12	010
6	0.10	011
7	0.09	1110
8	0.08	1111

The entropy of this source is found from equation (5.2) as 2.9436 bits/symbol. The Huffman coding efficiency is defined as the ratio of the entropy to the average code length, which is 99.11%. Thus, we find that the Huffman code is very nearly equal to the source entropy for this particular source. A MATLAB code for Huffman code generation is listed below. The procedure implemented by the MATLAB code is slightly different from the procedure described above [7, 8]. However, the code lengths are the same as well as the code efficiency.

```
% Example5_4.m
% Program to generate Huffman codes for the
% given probability vector p
% The Huffman codes and the corresponding code lengths are
% stored in the arrays HC and L, respectively.


%p = [1/8 1/4 1/10 1/10 1/16 1/16 1/12 13/60];
%p = [.2.15.13.12.1.09.08.13];
%p = [.25.25.125.125.0625.0625.0625.0625];
%p = [.5.1.1.1.1.1];
%p = [.2.18.1.1.1.06.06.04.04.04.04.03.01];
p = [.2.15.13.12.1.09.08.13];
% Check for negative value for any probability and
% if the sum of the probabilities do not add up to 1.
if any(find(p<0))
error('Negative value for probability\n')
end
if abs(sum(p)-1) >1.0e-10
error('Not a valid probability set\n')
end
% Compute the source entropy.
H = sum(-p.* log2(p+1e-08));
sprintf('Source Entropy = %5.2f\n',H)
% Call the function Huffman to generate the Huffman codes.
```

```
[HC,L] = Huffman(p);
% Compute the average code length of the Huffman codes.
AvgLength = sum(p.*double(L));
% compute the Huffman code efficiency as the ratio of
  entropy to
% the average code length.
E = 100*H/AvgLength;
sprintf('Average Code Length = %5.2f\n',AvgLength)
sprintf('Efficiency = %5.2f%%\n',E)
function [HC,L] = Huffman(p)
% [HC,L] = HUFFMAN(p)
% Generates Huffman codes for input symbols
% with a given probability distribution p
% The Huffman Codes are stored in the array HC
% The code lengths are stored in the array L
M = length(p);
N = M;
A = zeros(1,N-2);
p1 = sort(p,'descend');
% Part I of the code generation
for i = 1:N-2
p1(M-1) = p1(M-1) + p1(M);
s = find(p1(M-1) > = p1(1:M-2));
if ~isempty(s)
loc = s(1);
else
loc = M-1;
end
T = p1(M-1);
for k = M-2:-1:loc
p1(k+1) = p1(k);
end
p1(loc) = T;
A(N-2-i+1) = loc;
M = M -- 1;
end
% Part II of the code generation
HC = uint16(zeros(1,N));
L = uint16(ones(1,N));
HC(2) = 1;
M = 2; j = 1;
while j < = N-2
T = HC(A(j));
T1 = L(A(j));
for k = A(j):M-1
HC(k) = HC(k+1);
L(k) = L(k+1);
end
HC(N) = 0;
HC(M) = T;
L(M) = T1;
HC(M) = bitshift(HC(M),1);
HC(M+1) = HC(M) + 1;
L(M) = L(M) + 1;
L(M+1) = L(M);
M = M + 1;
j = j +1;
end
```

It can be shown that the average Huffman code length is within 1 bit of the source entropy, that is,

$$H(X) \leq \overline{L} \leq H(X) + 1 \tag{5.19}$$

It can also be shown that the average Huffman code length when K letters are grouped to form symbols is bounded by

$$H(X) \leq \overline{L} \leq H(X) + \frac{1}{K} \tag{5.20}$$

A difficulty with large K is that latency is introduced as there is a need for a buffer. We will discuss buffer control strategy in a later chapter.

5.4 ARITHMETIC CODING

Arithmetic coding is a variable-length lossless coding scheme that is gaining momentum in image compression standards [9–13]. It is more efficient for sources with small alphabets. In arithmetic coding, there is no requirement to store codebooks. Compare this with Huffman coding that requires codebooks of size K^M to be stored, where K is the source alphabet size and M is the symbol length. For a source with four letters and symbol length 8, the number of codewords to be stored is $4^8 = 65536$!

Arithmetic coding essentially amounts to computing the cumulative distribution function (CDF) of the probability of a sequence of symbols, and then representing the resulting numerical value in a binary code. The numerical value that is binary coded is called the *tag* or *identifier*. There is no need for the encoder or decoder to store codewords. Using the tag the decoder can uniquely decipher the encoded sequence. However, in order for the decoder to terminate decoding either it must know the sequence length or a special code indicating termination.

5.4.1 Arithmetic Coding a Sequence of Symbols

The arithmetic coding procedure can be divided into two parts: (a) generation of tag or identifier and (b) representing the tag as a truncated binary fraction. The source is assumed to have a probability model, that is, each source symbol has a certain probability of occurrence. The tag value will lie in the half open interval [0, 1) since the CDF lies between 0 and 1. Initially, the unit interval is divided into the number of source symbols with widths corresponding to the probabilities. Another way of looking at it is that the unit interval is divided such that the successive intervals correspond to the CDF of the source probability. With the arrival of the first symbol, the interval in which the symbol lies is divided into the number of source symbols in proportion to their probabilities and the rest of the intervals are discarded. After the second symbol arrives, the interval in which the second symbol lies is divided in proportion to the probabilities. This process is continued until the last symbol in the sequence. The tag value is located in the last interval. Any value in this interval may be used as the tag. One possible choice is to use the middle value of the last interval. Once the tag is determined, it is then represented as a binary fraction. Depending on the tag value, it may be necessary to truncate the binary fraction to a finite number of digits. Let us illustrate the arithmetic coding process by an example.

Example 5.5 Let us consider an information source with four letters in its alphabet $A = \{a_1, a_2, a_3, a_4\}$ and corresponding probabilities of 0.5, 0.3, 0.1, and 0.1, respectively. Let the sequence of symbols to be coded be $a_1a_3a_2a_4a_1$. Generate the tag and its corresponding binary fraction.

Solution Initially divide the unit interval in proportion to the probabilities starting with zero. Therefore, we find the boundary points at 0.5, 0.8, 0.9, and 1.0. Since the first symbol in the sequence is a_1, divide the interval [0, 0.5) in proportion to the four probabilities, as shown in Figure 5.6. At this point, the tag lies in the interval [0, 0.5). The boundary points after the first symbol are 0.25, 0.4, 0.45, and 0.5. As the second

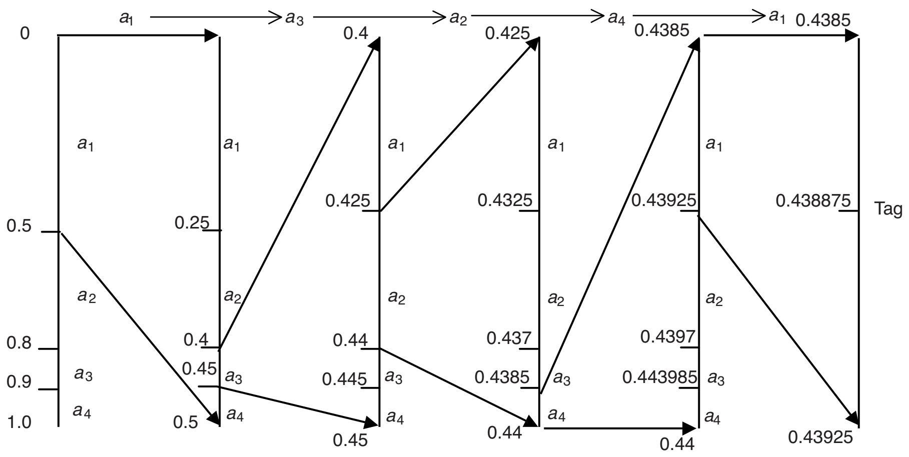


Figure 5.6 Arithmetic coding of a sequence of five symbols from a source alphabet of size four letters of Example 5.5.

symbol is a_3, which lies in the interval [0.4, 0.45), divide this interval in proportion to the probabilities. Again, at the end of the second symbol the tag lies in the interval [0.4, 0.45). Continuing further, we find the tag intervals to be [0.425, 0.440), [0.4385, 0.440), and [0.4385, 0.43925) corresponding to the last three symbols in the sequence in that order. The final tag resides in the last interval [0.4385, 0.43925). If we decide to choose the midpoint of the last interval as the tag, we obtain the tag value as 0.438875. If we represent this decimal fraction in binary fraction, we get 0.01110000010110 . . . . Truncating this binary fraction to 12 digits, the corresponding tag comes out as 0.4387207, which is in the interval [0.4385, 0.43925) and the tag is unique. Thus, the arithmetic code for the five-letter symbol sequence is the 12-bit binary fraction 0.011100000101. The entropy of the source is 1.6855 bits/symbol and the average length of the arithmetic code is $\frac{12}{5} = 2.4$ bits/symbol. In this particular case, the efficiency is only 70.23%. However, if we increase the symbol sequence length to 8 to give a sequence $a_1a_3a_2a_1a_4a_2a_1a_1$, then the tag comes out as 0.43215312 and the minimum number of bits required to represent the tag in a binary fraction is 16. Then average code length comes out as $\frac{16}{8} = 2$ bits/symbol and the efficiency goes to 84.27%.

We observe from Figure 5.6 that each successive interval is a subset of the previous interval and that it is unique. The only problem is that the fraction gets smaller and smaller in value and we need a high precision to represent the tag value in a binary code. This is one drawback of the arithmetic coding.

We can generalize the above procedure by sequentially computing the lower and upper boundaries of the interval in which the current symbol lies [14]. Let the symbol sequence to be coded be $\{a_1, a_2, a_3, \ldots, a_n\}$. Choosing the tag to be the midpoint of the final interval, the lower and upper boundaries of the tag are given by

$$\begin{aligned}
I_L(0) &= 0 \text{ and } I_U(0) = 1 \\
I_L(k) &= I_L(k-1) + (I_U(k-1) - I_L(k-1)) \times \mathrm{CDF}(m_k - 1) \\
I_U(k) &= I_L(k-1) + (I_U(k-1) - I_L(k-1)) \times \mathrm{CDF}(m_k) \\
\mathrm{Tag} &= \frac{I_U(k) + I_L(k)}{2}
\end{aligned}$$

In the Example 5.5, the symbol sequence takes on the values 1, 3, 2, 4, and 1 in that order. So, starting with $I_L(0) = 0$ and $I_U(0) = 1$, we have $m_1 = 1$:

$$\begin{aligned}
I_L(1) &= 0 + (1-0) \times \mathrm{CDF}(0) = 0 \\
I_U(1) &= 0 + (1-0) \times \mathrm{CDF}(1) = 1 \times 0.5 = 0.5
\end{aligned}$$

Next with $m_2 = 3$,

$$\begin{aligned}
I_L(2) &= 0 + (0.5-0) \times \mathrm{CDF}(2) = 0.5 \times 0.8 = 0.40 \\
I_U(2) &= 0 + (0.5-0) \times \mathrm{CDF}(3) = 0.5 \times 0.9 = 0.45
\end{aligned}$$

With the third symbol $m_3 = 2$, the lower and upper boundaries are

$$I_L(3) = 0.4 + (0.45 - 0.40) \times \text{CDF}(1) = 0.425$$
$$I_U(3) = 0.4 + (0.45 - 0.40) \times \text{CDF}(2) = 0.440$$

The fourth symbol is $m_4 = 4$. Therefore, the boundaries are

$$I_L(4) = 0.425 + (0.44 - 0.425) \times \text{CDF}(3) = 0.4385$$
$$I_U(4) = 0.425 + (0.44 - 0.425) \times \text{CDF}(4) = 0.440$$

The last symbol in the message sequence being $m_5 = 1$, we obtain the interval to be

$$I_L(5) = 0.4385 + (0.44 - 0.43855) \times \text{CDF}(0) = 0.4385$$
$$I_U(5) = 0.4385 + (0.44 - 0.43855) \times \text{CDF}(1) = 0.43925$$

This being the last interval in which the tag lies, we obtain the tag to be the midpoint of this interval. Thus, Tag = (0.43925 + 0.4385)/2 = 0.438875. The minimum number of binary digits $\overline{L}$ required to represent the tag is determined from

$$\overline{L} = \lceil -\log_2 \mathbf{p} \rceil + 1 \tag{5.21}$$

In equation (5.21), $\mathbf{p}$ is the probability of the symbol sequence and $\lceil x \rceil$ is the ceiling operation. For the memoryless source, $\mathbf{p}$ is the product of the probabilities of the symbols in the sequence.

5.4.2 Decoding the Symbol Sequence

The arithmetic encoding procedure is fairly simple. How about decoding? As we will see, the arithmetic decoding procedure is also fairly simple. The decoding procedure can be stated as follows:

1. Set the initial boundary values as: $I_L(0) = 0$ and $I_U(0) = 1$. Set $k = 1$. Let the symbol sequence length be M.
2. While $k \leq$ M, repeat the following steps.
3. Compute $T_1 = (\text{Tag} - I_L(k-1))/(I_U(k-1) - I_L(k-1))$, where Tag is the received tag value corresponding to the encoded symbol sequence.
4. Determine m_k such that $\text{CDF}(m_k - 1) \leq T_1 \prec \text{CDF}(m_k)$.
5. Now update the boundaries of the current interval using the equations $I_L(k) = I_L(k-1) + (I_U(k-1) - I_L(k-1)) \times \text{CDF}(m_k - 1)$ and $I_U(k) = I_L(k-1) + (I_U(k-1) - I_L(k-1)) \times \text{CDF}(m_k)$.
6. Increment k.

The MATLAB code implementing the arithmetic encoding and decoding procedures is listed below. The program also computes the minimum number of bits required to generate the truncated binary fraction of the tag. The binary fraction is returned in an unsigned integer. It further computes the source entropy, average arithmetic code length and the coding efficiency.

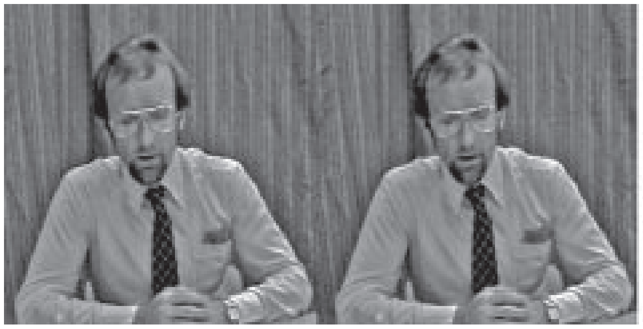
```
% Example5_5.m
% Example5_5 encodes a symbol sequence ''m'' using
  arithmetic coding.
% It generates a tag in a truncated binary fraction
% to a minimum number of bits ''TB'' of accuracy and
% decodes the input sequence.
% ''p'' is the probability vector of the source
% alphabet; ''m'' is the vector (of integers) whose
% elements are the symbols in the sequence. The values
% in ''m'' must be integers between 1 & length(p).
% ''Tag'' is the truncated value of the tag for the
% encoded sequence.
p = [0.5 0.3 0.1 0.1];% probability vector of input alphabet
if any(find(p<0))
error('Negative value for probability\n')
end
if abs(sum(p)-1) >1.0e-10
error('Not a valid probability set\n')
end
m = [1 3 2 4 1];% symbol sequence to be coded
%m = [1 3 2 1 4 2 1 1];
%TB is the significant number of bits of the Tag
TB = ceil(-log2(prod(p(m))))+1;
sprintf('Precision of Tag = %g bits\n',TB)
sprintf('Entropy = %4.3f;\tAvg. Code Length = %4.3f\n',...
sum(-p.*log2(p)),TB/length(m))
sprintf('Efficiency = %4.2f\n',(100*sum(-p.*log2(p))/
   TB*length(m)))
% compute the cumulative probability function (CDF) of p
CDF = zeros(1,length(p));
for i = 1:length(p)
CDF(i) = sum(p(1:i));
end
%
% Compute the Tag corresponding to the symbol sequence
% by calling the function ''ArithmeticEncode''
Tag = ArithmeticEncode(CDF,m);
% generate truncated code for the tag
AC = uint16(0);% contains the arithmetic code
k = 1;
T = Tag;
while k < = TB
```

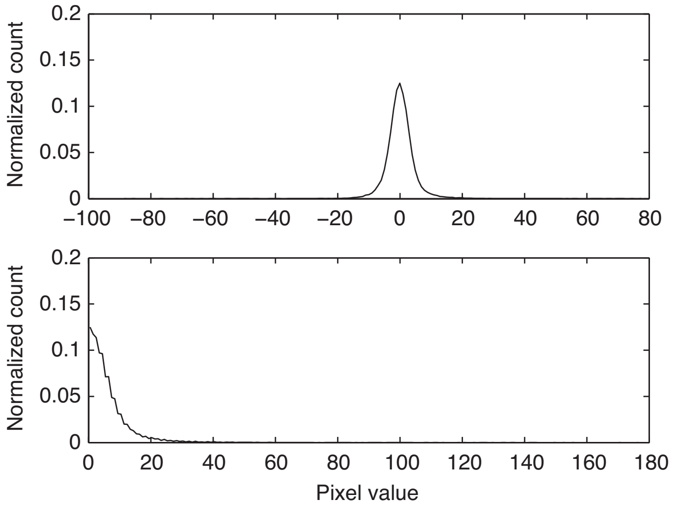
```
T = 2*T;
if T> = 1
AC = bitshift(AC,1);
AC = AC + uint16(1);
T = T -1;
else
AC = bitshift(AC,1);
end
k = k + 1;
end
% Now decode the symbol sequence
T1 = 0.0;
AC1 = AC;
TB1 = TB + 1;
for k = 1:TB
T1 = double(bitand(AC1,1))*(2^(k-TB1)) + T1;
AC1 = bitshift(AC1,-1);
end
sprintf('Tag = %10.8f\t\tTruncated Tag =
  %10.8f\t\tArithmetic code = %g\n',... Tag,T1,AC)
%
D = ArithmeticDecode(CDF,T1,length(m));
sprintf('\nEncoded Symbol Sequence :\n')
sprintf('%g\t',m)
sprintf('\nDecoded Symbol Sequence :\n')
sprintf('%g\t',D)
function Tag = ArithmeticEncode(CDF,m)
% Tag = ArithmeticEncode(p)
% Arithmetic coding procedure
% p is the probability vector of the source alphabet
% m is the symbol sequence to be coded
% CDF is the cumulative distribution



Low = 0; High = 1;% initial values of lower & upper intervals
%l and u are the next lower & upper intervals
k = 1;
while k < = length(m)
if m(k) = = 1
Lower = Low;
else
Lower = Low + (High-Low)*CDF(m(k)-1);
end
Upper = Low + (High-Low)*CDF(m(k));
Tag = (Upper+Lower)/2;
Low = Lower; High = Upper;
k = k + 1;
end
```

```
function DecodedSymbol = ArithmeticDecode(CDF,Tag,L)
% DecodedSymbol = ArithmeticDecode(m)
% Given the CDF, TAG, and the length of the
% symbol sequence, the function returns the
% symbol sequence in ''DecodedSymbol''
%
Low = 0; High = 1;% initialize lower & upper intervals
DecodedSymbol = int16(zeros(1,L));
k = 1;
while k < = L
T = (Tag -- Low)/(High-Low);
for j = 1:length(CDF)
if j = = 1
if T> = 0 && T<CDF(j)
DecodedSymbol(k) = j;
break
end
else
if T> = CDF(j-1) && T<CDF(j)
DecodedSymbol(k) = j;
break
end
end
end
if DecodedSymbol(k) = = 1
Lower = Low;
else
Lower = Low + (High-Low)*CDF(DecodedSymbol(k)-1);
end
Higher = Low + (High-Low)*CDF(DecodedSymbol(k));
Low = Lower; High = Higher;
k = k + 1;
end
```

5.5 GOLOMB–RICE CODING

Besides Huffman and arithmetic coding, there is another coding scheme called *Golomb–Rice* (GR) coding, which is simple and efficient for coding integers that have the *Laplacian* distribution [15,16]. Unlike the Huffman coding, there is no need to store codewords for the GR coding. In predictive coding of moving images, pixel differentials are encoded rather than the actual pixel values [17]. These differential pixels are integers and are typically distributed exponentially. Note that Laplacian distribution is another term for the exponential distribution. Figure 5.7a shows two consecutive frames (frames 8 and 9) from the Trevor sequence. The histogram of the differential frame (frame 9 − frame 8) is shown in Figure 5.7b, where the top plot is the histogram of the differential frame. It is (almost) symmetric about the zero value


(a)



(b)

Figure 5.7 GR coding of Example 5.6. (a) The top images show frames 8 (left) and 9 (right) of the Trevor sequence and (b) the bottom plots show the histogram of the differential frames 8 & 9 (top plot) and histogram of the mapped differential image (bottom plot).

and appears to have an exponential shape. We have also seen such a distribution for the detail coefficients of the discrete wavelet transform of images. It has been shown that encoding double-sided exponentially distributed integers using GR code is simple and more efficient than either Huffman or arithmetic coding. In what follows, we will briefly describe the GR coding.

GR coding encodes positive integers. Let n be a positive integer whose GR code we want to find and let $m = 2^k$, where k is a positive integer. Then $n = Q \times m + R$, where the quotient $Q = \left\lfloor \frac{n}{m} \right\rfloor$ and the remainder $R = n - mQ$. The GR code for n is obtained by concatenating the unary code of Q with the k-bit binary representation of the integer R. The unary code of 3, for example, is three zeros followed by a "1," that is, 0001. The digit "1" serves as the terminating code for the unary code. As an example, let $n = 7$ and $m = 2^2 = 4$. Then, $Q = \left\lfloor \frac{7}{4} \right\rfloor = 1$ and $R = 3$. The unary code of Q is 01 and the 2-bit binary representation of $R = 3$ is 11. Concatenating the unary code and the binary code, we get the GR code of 7 as 0111.

Although the GR coding applies only to positive integers, negative integers can also be coded using GR coding by first mapping it to a positive integer. Thus in GR coding, the first step is to map the integers into positive values. This mapping can be carried out by

$$\mathrm{M}(n) = \begin{cases} 2n, & n \geq 0 \\ 2\,|n| - 1, & n \prec 0 \end{cases} \tag{5.22}$$

Equation (5.22) maps positive integers into even integers and negative integers into odd integers. Thus, there is no need for an extra bit to denote the sign. The GR coding procedure can be describes as follows.

GR Coding Procedure

Given an integer n to be coded and an $m = 2^k$, where k is a positive integer,

1. Map n to $\hat{n}$ using (5.22).
2. Compute the quotient $Q = \left\lfloor \frac{\hat{n}}{m} \right\rfloor$ and the remainder $R = \hat{n} - mQ$.
3. Concatenate the unary code of Q with the k-bit binary code of R.

GR Decoding Procedure

Decoding the GR code is simple and can be described as follows.
Given the GR code of an integer and the value of m,

1. Compute $k = \log_2 m$.
2. Count the number of zeros starting from the most significant bit position until a "1" is encountered. The number of zeros gives the value of Q. Compute $P = m \times Q$.
3. Treat the remaining k digits in the given GR code as the k-bit binary representation of the integer R, $\hat{n} = P + R$.
4. The encoded integer

$$n = \begin{cases} \dfrac{\hat{n}}{2}, & \text{if } \hat{n} \text{ is even} \\[2ex] -\dfrac{\hat{n}+1}{2}, & \text{if } \hat{n} \text{ is odd} \end{cases}$$

For example, if $m = 4$ and the GR code is 0000110, then $Q = 4$, $P = 4 \times 4 = 16$, $R = 2$, and $\hat{n} = 16 + 2 = 18$. Therefore, $n = \frac{18}{2} = 9$. On the other hand, if the GR code is 0000101, then $\hat{n} = 16 + 1 = 17$, which is an odd integer. Therefore, $n = \frac{-(17+1)}{2} = -9$.

Example 5.6 Obtain the differential frame corresponding to the frames 8 and 9 of the Trevor sequence, and then, calculate the average length of the GR code for this differential frame. Plot the histograms of the differential and the mapped images and visually verify that the histogram of the differential image corresponds to the Laplacian distribution, while the histogram of the mapped differential frame has only positive integer values.

Solution The original frames 8 and 9 of the Trevor sequence are shown in Figure 5.7a. The histograms of the differential image and the mapped differential image are shown in Figure 5.7b. It is clearly seen from the top plot in Figure 5.7b that the differential image has the Laplacian distribution. After mapping, the differential image has a histogram that has only positive values, as seen from Figure 5.7b bottom plot. For this differential image, the GR code has an average length of 4.38 bits/pixel. The entropy of the differential image is 4.05 bits/pixel and the GR coding efficiency is 92.48%. A listing of the MATLAB code for this example is shown below.

```
%Example5_6.m
% Encode the differential image using Golomb-Rice
% coding.


A = imread('twy010.ras');% read frame #n
B = imread('twy011.ras');% read frame #n+1
%
FileNames = {'twy010.ras' 'twy011.ras'};
montage(FileNames)% function Montage displays the two
% image frames side by side in a single window.
%
% compute the differential frame = frame #(n+1) -- frame #n
E = double(B)-double(A);
%
% compute the histogram of the differential image
[H1,Bin1] = hist(E(:),max(E(:))-min(E(:))+1);
H1 = H1/sum(H1);% normalize the count
%
E1 = zeros(size(A));% array to store the mapped values
% map all negative values to odd positive integers
[y,x] = find(E<0);
for i = 1:size(x,1)
E1(y(i),x(i)) = 2*abs(E(y(i),x(i)))-1;
end
clear x y;
% map all positive values to even positive integers
[y,x] = find(E> = 0);
for i = 1:size(x,1)
E1(y(i),x(i)) = 2*E(y(i),x(i));
end
% determine m = 2^k
```

```
k = ceil(log10(mean(E1(:)))/log10(2));
m = 2^k;
% compute the histogram of the mapped differential image
N = max(E1(:)) + 1;
[H,Bin] = hist(E1(:),N);
H = H/sum(H);
figure, subplot(2,1,1),plot(Bin1,H1,'k')
title('Histogram of the differential image')
ylabel('normalized count')
subplot(2,1,2),plot(Bin,H,'k')
title('Histogram of mapped differential image')
xlabel('pixel value')
ylabel('normalized count')
% compute the average length of GR code = k+floor(n/m)+1
AvgCodeLn = sum((floor(Bin/m)).* H)+k+1;
sprintf('Average Golomb-Rice Code length = %3.2f\n',AvgCodeLn)
```

5.6 RUN-LENGTH CODING

RLC is typically used to code binary sources such as documents, maps, and facsimile images, all of which consist of BW dots or pixels. In these binary sources, a white pixel is represented by a “1” and a black pixel by a “0.” When coding such a binary source output, one encounters alternating runs of zeros and ones. Instead of coding the individual BW pixels, it is more efficient to code the runs of zeros and ones. A run-length of zeros represents the number of consecutive zeros between ones. Similarly, a run-length of ones is the number of consecutive ones between zeros. One way to code the run-lengths is to use codewords of fixed length “b” corresponding to a maximum run-length $L - 1$, where $L = 2^b$. Since we are using only one set of codewords, there must be a way of telling the decoder which run-length comes first. Typically, the first run is a white run of length zero. There on, the runs alternate and the decoding is straightforward.

A variable-length coding is more efficient than a fixed-length coding. Thus, the runs can be coded more efficiently by using Huffman coding. If the codebook is large, then truncated Huffman codes are more efficient. In truncated Huffman coding of RLC, variable-length codes are used up to separate maximum run-lengths for BW pixels [18]. The remaining longer run-lengths are coded by fixed-length codes, which are made up of prefix codes followed by a fixed number of bits.

Another situation where RLC is used is in image compression using transform and wavelet coding techniques. In transform coding using DCT, the DCT coefficients in each $N \times N$ block are first quantized to varying levels depending on the required compression. This quantization process results in a large number of zero-valued coefficients, which occur in bursts or in runs. Therefore, it is advantageous to encode the run of zeros using RLC rather than coding each zero-valued coefficient individually. This is a standard procedure in such compression standards as Joint Photographic Experts Group (JPEG) and MPEG. In fact, in such standards *run-length amplitude* or *run-length level* coding is used. Run-length amplitude represents a run of zeros followed by a nonzero amplitude of a DCT coefficient. For instance, one of the 8 × 8 DCT blocks of the *cameraman* image is shown below with the coefficients being rounded.

Rounded 8 × 8 DCT Coefficients

873	−698	106	94	−58	56	−10	−70
−9	13	13	−7	−12	10	−1	−10
−8	−1	−3	3	3	8	4	8
−1	−3	−6	4	5	2	4	6
−2	3	1	−2	−2	−1	−1	−1
4	2	1	−1	−3	−3	−3	−2
−1	3	0	0	2	−3	−2	3
−2	−2	−1	−1	1	3	1	0

After the quantization by a factor of 18 with rounding, the quantized 8 × 8 DCT block looks like this:

Quantized DCT Coefficients

49	−38	6	6	−3	4	0	−3
0	1	1	0	0	1	0	0
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0
0	1	1	1	1	0	0	1
0	0	0	0	1	1	1	0

If the above quantized DCT block is scanned left-to-right, top-to-bottom, and excluding the DC (top left) coefficient, we obtain the number of run-lengths as shown in Table 5.3. On the other hand if we use a zigzag scanning pattern (Figure 5.8), then the number of different run-lengths obtained for the same quantized DCT block is shown on the right side of Table 5.3.

Table 5.3 Length of zero runs and counts for the quantized coefficients

Zigzag Scanning		Raster Scanning	
RL	Count	RL	Count
1	7	1	4
2	7	2	2
3	1	3	1
4	0	4	1
5	0	5	2
6	1	6	1

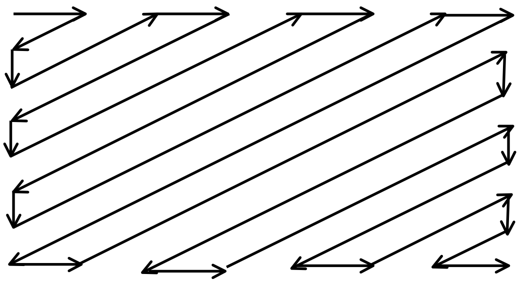


Figure 5.8 A zigzag scanning pattern of an 8 × 8 DCT block.

An alternative to using RLC is the so-called *run/level*, which is what is used in JPEG and MPEG standards. Using run/level coding, we get the different run/levels for the same quantized DCT coefficients after zigzag scanning, as shown in Table 5.4. These can then be coded using Huffman coding.

5.7 SUMMARY

Lossless coding is needed in converting source alphabets into efficient codes for either transmission or storage. In image compression, source alphabets could be pixel intensities or quantized transform coefficients or differential pixel values or quantized DWT coefficients. In this chapter, we have described several lossless coding methods, all of which generate codes of varying length. Information theory deals with quantifying the source information, which is very useful in the design of digital communications systems. As all information sources produce long sequences of discrete output symbols for transmission or storage, information theory defines the source entropy, which is the average number of bits per symbol required to encode the data. The entropy of a source is the lower bound on achievable lossless compression by a practical coding scheme.

Table 5.4 Run/level and counts for the quantized coefficients using zigzag scanning

Run/Level	Count
0/1	12
0/4	1
0/6	2
1/1	5
1/−3	2
2/1	7
3/1	1
6/−3	1

We described three variable-length coding schemes in this chapter. Huffman coding is very efficient in the sense that its average code length is very close to the source entropy. However, it requires the storage of codewords at both the encoder and decoder sides. This is especially inefficient when the source alphabet is large. An alternative to the Huffman coding scheme, arithmetic coding is a variable-length coding that is more efficient when the source alphabet is small. It does not require the codes to be stored. The drawbacks of the arithmetic coding are that it is a bit more complex than the Huffman coding and that higher arithmetic precision is required to represent the tag. A third lossless coding procedure known as the GR coding was also described. GR coding is simple and efficient and is well suited for differential images with integer pixel values. Several examples are given illustrating the coding principles of these schemes along with corresponding MATLAB codes.

The succeeding three chapters will deal with compressing images using predictive, transform, and DWT coding methods, where lossless coding will be incorporated at the encoding side. As mentioned earlier, lossless coding not only encodes source symbols but also achieves additional compression by exploiting redundancy in the source symbols, that is, by exploiting the redundancy in the quantized source symbols such as the transform coefficients.

REFERENCES

1. C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, 379–423 (Part I) and 623–656 (Part II), 1948.
2. R. G. Gallager, *Information Theory and Reliable Communication*, Halsted Press, John Wiley, New York, 1968.
3. A. J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding*, McGraw-Hill, New York, 1979.
4. N. S. Jayant and P. Noll, *Digital Coding of Waveforms—Principles and Applications to Speech and Video*, Prentice Hall, Englewood, NJ, 1984.
5. C. E. Shannon, "Communication in the presence of noise," *Proc. IRE*, 10–21, 1949.
6. D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, 40, 1098–1101, 1951.
7. M. A. Sid-Ahmed, *Image Processing—Theory, Algorithms and Architecture*, McGraw-Hill, New York, 1995.
8. J. G. Proakis and M. Salehi, *Contemporary Communication Systems Using MATLAB*, PWS Publishing Company, 1998.
9. N. Abramson, *Information Theory and Coding*, McGraw-Hill, New York, 1963.
10. F. Jelinek, *Probabilistic Information Theory*, McGraw-Hill, New York, 1968.
11. R. Pasco, *Source Coding Algorithms for Fast Data Compression*, Ph.D. thesis, Stanford University, 1976.
12. J. J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM J. Res. Dev.*, 20, 198–203, 1976.
13. J. J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM J. Res. Dev.*, 23 (2), 149–162, 1979.
14. K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, San Francisco, 1996.
15. S. W. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory*, IT-12, 399–401, 1966.

16. R. F. Rice, "Some practical universal noiseless coding techniques," Jet Propulsion Laboratory, JPL Publication, Pasadena, CA, pp. 79–22, 1979.
17. K. S. Thyagarajan, "DCT compression using Golomb–Rice coding," US Patent, 6,735,254, 2001.
18. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

PROBLEMS

5.1. If $\mathbf{X}$ is a random variable with L possible outcomes, show that its entropy is $0 \le H(\mathbf{X}) \le \log_2(L)$.

5.2. Consider the discrete source with alphabets $A = \{a_1, a_2, a_3, a_4\}$. Determine the entropy of the source if (a) $P(a_i) = 2^{-i}$, $1 \le i \le 4$, (b) $P(a_1) = 0.5$, $P(a_2) = 0.15$, $P(a_3) = 0.1$, and $P(a_4) = 0.25$.

5.3. Consider a first-order binary Markov source with state probabilities $P(s_1) = 0.3$ and $P(s_2) = 0.7$. If the state transition probabilities are $P(s_2 | s_1) = 0.15$ and $P(s_1 | s_2) = 0.45$, determine the source entropy.

5.4. Design a Huffman code for a discrete source with an alphabet of five letters and probabilities 0.35, 0.2, 0.15, 0.1, and 0.2. Compute the average code length and redundancy of the Huffman codes. Note that the redundancy is the difference between the entropy and average code length.

5.5. Read in 3 or 4 real intensity images (8 bpp) of your choice and compute the histogram of the combined images with 255 bins. Generate a Huffman code from the histogram.

5.6. For a discrete source with an alphabet $A = \{a_1, a_2, a_3, a_4\}$ and letter probabilities 0.4, 0.2, 0.25, and 0.15, respectively, determine the tag corresponding to the symbol sequence $a_1\ a_3\ a_2\ a_4$.

5.7. For two consecutive frames of a sequence image (8 bpp), calculate the difference image. Show that the histogram of the difference image is roughly Laplacian. Calculate the average bit rate in bpp of the difference image using GR code.

6 PREDICTIVE CODING

6.1 INTRODUCTION

As mentioned in Chapter 1, image compression is achieved by removing pixel redundancies. For instance, pixels along a scan line are highly correlated (see Figure 1.4). In a similar manner, pixels in a rectangular neighborhood are also highly correlated. A high correlation implies that a pixel is predictable based on the values of the pixels in its neighborhood. Therefore, one can transmit or store the pixel differences rather than the actual pixels themselves to achieve compression because the differential pixels are very nearly zero with a high probability and require less number of bits to code. This is the basis behind predictive coders. Predictive coding is used in Moving Picture Experts Group (MPEG) standards to code moving pictures.

Consider a discrete sequence of samples $\{x[n], n = 0, 1, \ldots\}$. This sequence may well be pixels along a particular row of an image. Then, the current sample or pixel $x[n]$ can be predicted or estimated from the previous pixels, which may be expressed as

$$\hat{x}[n] = f(x[n-1],\ x[n-2], \ldots) \tag{6.1}$$

In equation (6.1), $\hat{x}[n]$ is the estimate of $x[n]$, the current pixel, and $f(\cdot)$ is the prediction rule to be determined. The differential pixel $e[n]$ corresponding to the current pixel is the difference between $x[n]$ and its estimate $\hat{x}[n]$. That is,

$$e[n] = x[n] - \hat{x}[n] \tag{6.2}$$

Still Image and Video Compression with MATLAB By K. S. Thyagarajan.
Copyright © 2011 John Wiley & Sons, Inc.

The differential pixel is quantized and transmitted or stored. Thus, the transmitted differential pixel $e_q[n]$ is written as

$$e_q[n] = Q(e[n]) \tag{6.3}$$

where $Q(\bullet)$ is the quantization rule. which is nonlinear. Most often the input pixels are *pulse code modulator* (PCM) samples with 8 or more bits of quantization and the differential pixels are quantized to fewer bits (3 or 4 bits). Note that since we are only transmitting the quantized differential pixels, the predicted pixel value depends on the previously reconstructed pixels $\{\tilde{x}[n-1], \tilde{x}[n-2], \ldots\}$. This forms a closed loop prediction as shown in Figure 6.1. Therefore, equation (6.1) takes the form

$$\hat{x}[n] = f(\tilde{x}[n-1],\ \tilde{x}[n-2], \ldots) \tag{6.4}$$

The reconstructed pixel $\tilde{x}[n]$ is simply the sum of the predicted pixel and quantized differential pixel [1]:

$$\tilde{x}[n] = \hat{x}[n] + e_q[n] \tag{6.5}$$

The reconstructed pixel error $\varepsilon[n]$, which is the difference between the actual and reconstructed pixels, is

$$\varepsilon[n] = x[n] - \tilde{x}[n] \tag{6.6}$$

Using equations (6.2) and (6.5) in (6.6), we get

$$\varepsilon[n] = e[n] - e_q[n] = q[n] \tag{6.7}$$

Equation (6.7) implies that the overall error in reconstructing the pixels depends on the design of the quantizer. This is rather interesting because equation (6.7) says that no matter what the prediction rule is, the overall reconstruction error depends on the quantizer performance. But be careful! Recall from the scalar quantizers in Chapter 2 that the Lloyd–Max quantizer design is based on optimizing with respect to both the probability density function (PDF) of the input and mean square error (MSE). Hence, the performance of the predictive coder depends on both the predictor and the quantizer! In Figure 6.1, the decoder is also shown. Observe that the encoder has

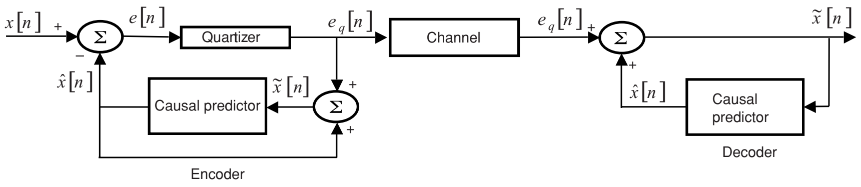


Figure 6.1 Block diagram of a differential pulse code modulator.

the decoder in a closed loop. This type of predictive coder is called the *differential pulse code modulator* (DPCM).

If the prediction amounts to a simple differencing, then the residual or differential pixel values are integers when the input pixels are integers. In this case, the pixel differentials may be entropy coded without quantization using any one of the lossless coding schemes described in Chapter 5 [2–4]. Then, the resulting DPCM is a lossless one. In fact, the MPEG standard uses simple differencing and entropy coding for coding moving pictures.

6.2 DESIGN OF A DPCM

There are two subsystems in the simple DPCM of Figure 6.1: (1) the predictor and (2) the quantizer. Thus, to design a DPCM for image compression, one must design the predictor and the quantizer. The design of an optimal quantizer was discussed in detail in Chapter 2. Therefore, we will focus our attention on the design of the predictor. In what follows, we will discuss in detail the design of a linear predictor. In most cases, linear prediction is used because it lends itself to analytical solution. While nonlinear prediction may prove to perform better, it is difficult to analyze and a closed form solution may not be feasible. Hence, our discussion on prediction will be constrained to linear case only.

6.2.1 One-Dimensional Linear Prediction

Consider a discrete sequence $\{x[n], n = 0, 1, \ldots\}$, which may be pixels along a row of an image. If the pixels are highly correlated, then it makes good sense to estimate the value of the current pixel $x[n]$ in terms of the previous pixels. That is to say that we can express the estimate or the predicted value of the current pixel as a weighted sum of the previous pixels. Thus,

$$\hat{x}[n] = \sum_{m=1}^{p} \alpha_m x[n-m] \tag{6.8}$$

Equation (6.8) describes a pth-order predictor because p previous pixels are used in estimating the current pixel value. Even though simple differencing is a degenerate case of linear prediction, it has no degree of freedom to optimize a cost function. Therefore, we will consider the general case where the predictor weights have to be determined to satisfy a chosen cost function.

First-Order Predictor For the first-order linear predictor, equation (6.8) reduces to

$$\hat{x}[n] = \alpha x[n-1] \tag{6.9}$$

In terms of the predicted pixel and the differential pixel, the actual pixel takes the form of

$$x[n] = \hat{x}[n] + e[n] \tag{6.10}$$

Since the predicted pixel is some multiple of the previous pixel, we observe from equation (6.10) that the error or differential pixel and the previous pixel are orthogonal. That is,

$$E\{e[n]\,x[n-1]\} = 0 \tag{6.11}$$

Equation (6.11) is formally written as $e[n] \perp x[n-1]$. The prediction error is also called the innovation [5]. Note that we are using the actual previous pixel in predicting the current pixel instead of using the previously reconstructed pixel $\tilde{x}[n-1]$. This will simplify the analysis. The value of α is chosen so as to minimize a cost function. The usual cost function used in practice is the *MSE* between the actual and predicted pixels over the whole image. The MSE is written as

$$\text{MSE} = E\left\{e^2[n]\right\} = E\left\{(x[n] - \hat{x}[n])^2\right\} \tag{6.12}$$

In equation (6.12), E denotes the expectation operation. Using equation (6.9) in (6.12), the MSE is

$$\text{MSE} = \text{E}\left\{(\text{x}[\text{n}] - \alpha x[n-1])^2\right\} \tag{6.13}$$

Minimization of the MSE is achieved by differentiating equation (6.13) with respect to α and setting the derivative to zero. Thus,

$$\frac{\partial \text{MSE}}{\partial \alpha} = \frac{\partial E\left\{(x[n] - \alpha x[n-1])^2\right\}}{\partial \alpha} = 0 \tag{6.14}$$

Since the expectation is linear, we can write equation (6.14) as

$$E\left\{\frac{\partial (x[n] - \alpha x[n-1])^2}{\partial \alpha}\right\} = 0 \tag{6.15}$$

We, therefore, obtain the optimum value for the weight as

$$\alpha = \frac{E\{x[n]\,x[n-1]\}}{E\left\{(x[n-1])^2\right\}} = \frac{R_x(1)}{R_x(0)} = \rho_1 \tag{6.16}$$

In equation (6.16), $R_x(0)$ and $R_x(1)$ are the autocorrelation function of the input sequence at lags 0 and 1, respectively, and ρ_1 is the corresponding correlation coefficient. Using equation (6.16) in (6.11), we obtain the minimum MSE for the first-order linear predictor as

$$\begin{aligned} \mathrm{MSE}_{\min} &= E\left\{e[n]\left(x[n]-\alpha x[n-1]\right)\right\} \\ &= E\left\{e[n]\,x[n]\right\} - \alpha E\left\{e[n]\,x[n-1]\right\} \end{aligned} \tag{6.17}$$

Since $e[n] \perp x[n-1]$, equation (6.17) reduces to

$$\begin{aligned} \mathrm{MSE}_{\min} &= E\left\{\left(x[n]-\alpha x[n-1]\right)x[n]\right\} = R_{\mathbf{x}}(0) - \alpha R_{\mathbf{x}}(1) \\ &= R_{\mathbf{x}}(0)\left(1-\alpha\frac{R_{\mathbf{x}}(1)}{R_{\mathbf{x}}(0)}\right) = R_{\mathbf{x}}(0)\left(1-\rho_1^2\right) \end{aligned} \tag{6.18}$$

If the input sequence has zero mean, then the minimum MSE of the optimum prediction is

$$\mathrm{MSE}_{\min} = \sigma_x^2\left(1-\rho_1^2\right) \tag{6.19}$$

The implication of equation (6.19) is that by employing optimal first-order prediction, the differential signal of the DPCM has a variance that is reduced from that of the input PCM signal by a factor of $\left(1-\rho_1^2\right)$. This reduction in the variance of the differential signal can be expressed as the gain in performance due to prediction. Thus, the *prediction gain* G_{P} of the optimal first-order predictor is defined as the ratio of the input signal variance to that of the prediction *error* or differential signal [6]. Therefore,

$$G_{\mathrm{P}} = \frac{\sigma_x^2}{\sigma_e^2} = \frac{1}{1-\rho_1^2} \tag{6.20}$$

When the number of levels of quantization is large for both the PCM and DPCM coders, the performance improvement in terms of the signal-to-noise ratio (SNR) in going from PCM to DPCM can be expressed as [6, 7]

$$\mathrm{SNR}_{\mathrm{DPCM}}\,(\mathrm{dB}) = \mathrm{SNR}_{\mathrm{PCM}}\,(\mathrm{dB}) + 10\ \log G_{\mathrm{P}} \tag{6.21}$$

For typical intensity (BW) images, $\rho \approx 0.95$ and the corresponding SNR improvement over PCM can be as much as 10 dB. Alternatively, the bit rate reduction can be shown to be

$$R_{\mathrm{PCM}} - R_{\mathrm{DPCM}} = \frac{1}{2}\ \log_2 \frac{1}{1-\rho_1^2} \tag{6.22}$$

Then, for typical intensity images, the rate reduction can be anywhere between 1.7 and 2.8 bits. In addition to being reduced in its variance, the differential signal has a PDF that is Laplacian.

Second-Order Linear Predictor The second-order linear predictor is characterized by

$$\hat{x}[n] = \alpha_1 x[n-1] + \alpha_2 x[n-2] \tag{6.23}$$

As in the case of the first-order predictor, the prediction error $e[n] = x[n] - \hat{x}[n]$ of the second-order linear predictor is orthogonal to both $x[n-1]$ and $x[n-2]$, that is, $e[n] \perp x[n-1]$ and $e[n] \perp x[n-2]$. Therefore, we obtain the following equations:

$$E\{e[n]x[n-1]\} = E\{(x[n] - \alpha_1 x[n-1] - \alpha_2 x[n-2])x[n-1]\} = 0 \tag{6.24}$$

$$E\{e[n]x[n-2]\} = E\{(x[n] - \alpha_1 x[n-1] - \alpha_2 x[n-2])x[n-2]\} = 0 \tag{6.25}$$

Expanding and rearranging equations (6.24) and (6.25), the minimum MSE results in the equations

$$\begin{bmatrix} R_x(0) & R_x(1) \\ R_x(1) & R_x(0) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} R_x(1) \\ R_x(2) \end{bmatrix} \tag{6.26}$$

The solution to equation (6.26) gives the optimal predictor coefficients, which are

$$\alpha_1 = \frac{\rho_1(1-\rho_2)}{1-\rho_1^2} \tag{6.27}$$

$$\alpha_2 = \frac{\rho_2 - \rho_1^2}{1-\rho_1^2} \tag{6.28}$$

where

$$\rho_1 = \frac{R_x(1)}{R_x(0)} \tag{6.29a}$$

$$\rho_2 = \frac{R_x(2)}{R_x(0)} \tag{6.29b}$$

The resulting minimum MSE of the second-order linear predictor can be found by substituting the values for the optimal predictor coefficients in the expression for the MSE as

$$\text{MSE}_{\min} = E\{e[n]x[n]\} = E\{(x[n] - \alpha_1 x[n-1] - \alpha_2 x[n-2])x[n]\} \tag{6.30}$$

Using equations (6.27) and (6.28) in (6.30), the minimum MSE of the second-order linear predictor is found to be

$$\text{MSE}_{\min} = R_x(0) - \alpha_1 R_x(1) - \alpha_2 R_x(2) = R_x(0)(1 - \alpha_1\rho_1 - \alpha_2\rho_2) \tag{6.31}$$

Substituting for α_1 and α_2 in equation (6.31) and with some algebraic manipulation, the minimum MSE of the second-order linear predictor can be shown to be

$$\mathrm{MSE}_{\min} = \sigma_x^2 \left\{ 1 - \rho_1^2 - \frac{\left(\rho_1^2 - \rho_2\right)^2}{1 - \rho_1^2} \right\} \tag{6.32}$$

The resulting prediction gain is [6]

$$G_{\mathrm{P}} = \frac{\sigma_x^2}{\mathrm{MSE}_{\min}} = \frac{1}{1 - \rho_1^2 - \dfrac{\left(\rho_1^2 - \rho_2\right)^2}{1 - \rho_1^2}} \tag{6.33}$$

As a comparison with the first-order predictor, if the correlation coefficient decreases exponentially, that is, if $\rho_2 = \rho_1^2$ [7], then the prediction gain of the second-order predictor is the same as that of the first-order predictor and there is no advantage in using a second-order linear predictor.

Linear Predictor of Order *p* A *p*th-order linear predictor is described by

$$\hat{x}[n] = \sum_{i=1}^{p} \alpha_i x[n-i] \tag{6.34}$$

The optimal p coefficients are determined by minimizing the MSE between the actual and predicted pixel values. Minimization of the MSE is achieved by setting the partial derivatives of the MSE with respect to the predictor coefficients to zero. Alternatively, using the orthogonality principle, minimization of the MSE is equivalent to setting the joint expectation of the prediction error and the previously predicted pixels to zero. Thus,

$$E\{e[n]\,x[n-i]\} = 0, \; 1 \le i \le p \tag{6.35}$$

Since

$$e[n] = x[n] - \hat{x}[n] = x[n] - \sum_{j=1}^{p} \alpha_j x[n-j] \tag{6.36}$$

Equation (6.35) results in p equations, which are

$$E\left\{ \left(x[n] - \sum_{j=1}^{p} \alpha_j x[n-j] \right) x[n-i] \right\} = 0, \; 1 \le i \le p \tag{6.37}$$

In matrix form, equation (6.37) can be written as

$$\begin{bmatrix} R_x(0)\, R_x(1) \ldots\ldots\ldots\ldots\ldots\ldots R_x(p-1) \\ R_x(1)\, R_x(0) \ldots\ldots\ldots\ldots\ldots\ldots R_x(p-2) \\ R_x(2)\, R_x(1) \ldots\ldots\ldots\ldots\ldots\ldots R_x(p-3) \\ \vdots \\ \vdots \\ R_x(p-1) \ldots\ldots\ldots\ldots\ldots\ldots R_x(0) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \vdots \\ \vdots \\ \alpha_p \end{bmatrix} = \begin{bmatrix} R_x(1) \\ R_x(2) \\ \vdots \\ \vdots \\ \vdots \\ R_x(p) \end{bmatrix} \tag{6.38}$$

Note that a unique solution to equation (6.38) exists because the autocorrelation matrix in equation (6.38) is positive definite.

Example 6.1 Design a first-order line-by-line DPCM (encoder only) for an intensity (BW) image with 8 bpp. Calculate its prediction gain and bit rate reduction. Display the histograms of the unquantized and quantized differential images as well as the reconstructed image. Finally, calculate the SNR in dB due to DPCM encoding. Use a 5-bit Lloyd–Max quantizer. Compare its performance with that of a uniform quantizer with the same number of bits as the nonuniform quantizer.

Solution Let us read the cameraman image, which is of size 256 × 256 pixels at 8 bpp. The first-order linear predictor coefficient is found to be 0.9362 for this image. Table 6.1 lists the various quantities of interest. The histograms of the prediction error for both unquantized (top figure) and quantized (bottom figure) differential images corresponding to the nonuniform quantizer are shown in Figure 6.2. The reconstructed image using the 5-bit nonuniform quantizer is shown in Figure 6.3. As a comparison, the corresponding histograms and reconstructed image for the uniform quantizer are shown in Figures 6.4 and 6.5, respectively. There are no apparent visual differences in the reconstructed images between the two cases at 5 bpp. However, nonuniform quantizer performs better with respect to prediction gain, rate reduction, and SNR. In Figure 6.6, the SNR is plotted versus the number of bits of quantization for the two quantizers. From the figure is seen that the nonuniform quantizer outperforms the uniform quantizer at higher bit rates. This is because the quantization step size is large at low bit rates. It should be pointed out that at 4 bpp rate, the

Table 6.1 First-order 1D DPCM: 5 bpp; cameraman image

Performance Parameter	Uniform Quantizer	Nonuniform Quantizer
Gp	8.91 dB	8.91 dB
Rate reduction	1.48 bits	1.48 bits
SNR (dB)	24.02	24.83

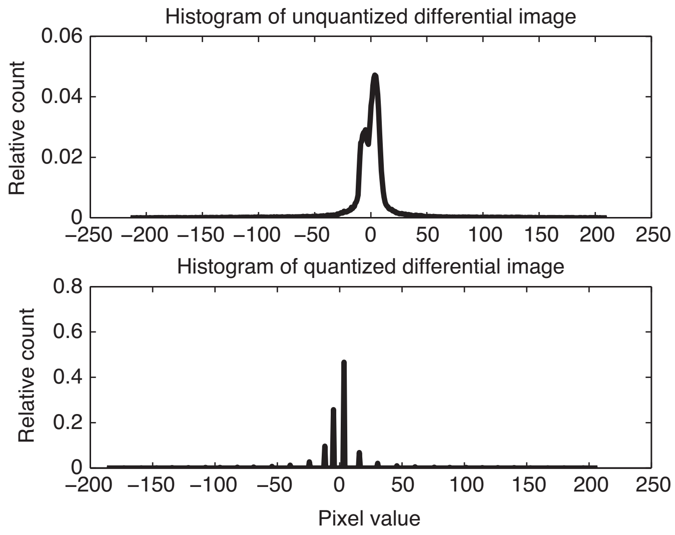


Figure 6.2 Histogram of the prediction error of the DPCM in Example 6.1. The top histogram is unquantized at 5 bpp using the nonuniform quantizer and the bottom is quantized.

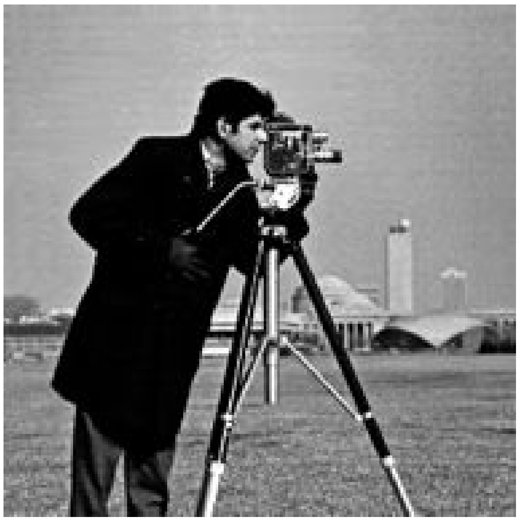


Figure 6.3 Reconstructed cameraman image at 5 bpp using the nonuniform quantizer.

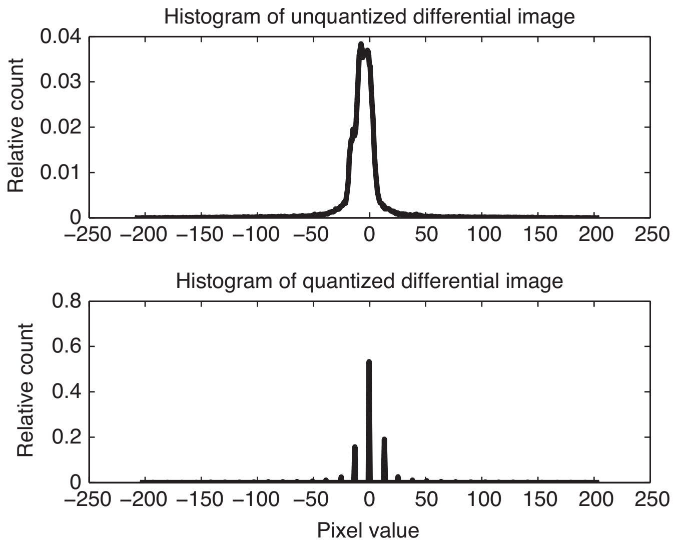


Figure 6.4 Histogram of the prediction error of the DPCM in Example 6.1. The top histogram is unquantized at 5 bpp using the uniform quantizer and the bottom is quantized.

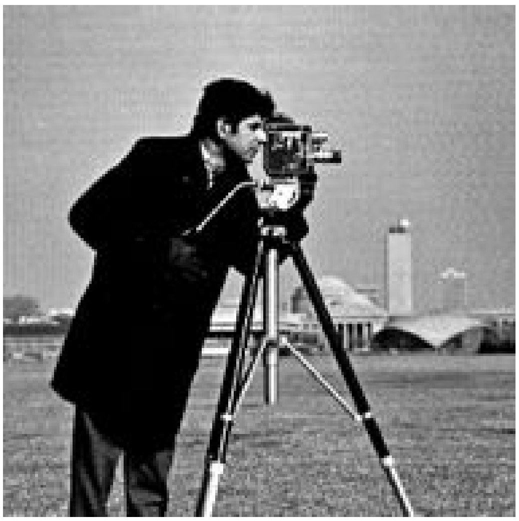


Figure 6.5 Cameraman image at 5 bpp using the uniform quantizer.

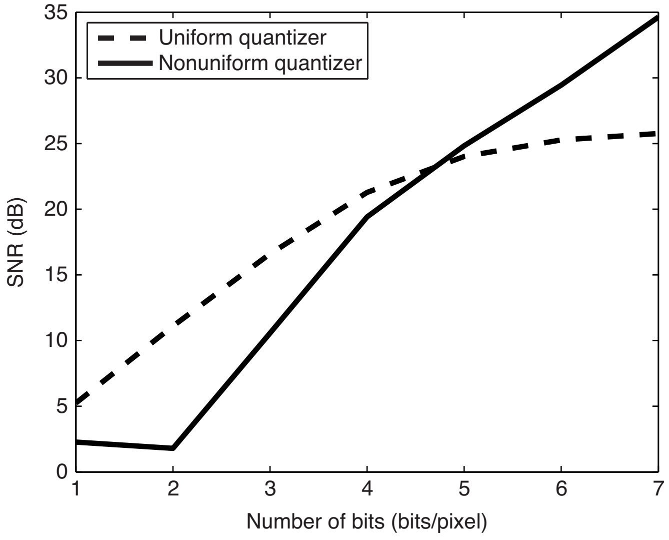


Figure 6.6 SNR of the DPCM in Example 6.1 for both uniform and nonuniform quantizers.

nonuniform quantizer performs better than the uniform quantizer visually even though the SNR is lower. The MATLAB code for this example is listed below.

```
% Example6_1.m
% Designs a 1st-order DPCM (encoder only)using both
% uniform and non-uniform quantizers with user specified
% number of bits of quantization. It also calculates
% the prediction gain and bit rate reduction. It further
% displays the histograms of the quantized and unquantized
% prediction error images as well as the reconstructed
% image.Finally, the SNR(dB) due to DPCM encoding is
% calculated.

clear, %close all
A = imread('cameraman.tif');
%A = imread('barbara.tif');
[Height,Width,Depth] = size(A);
if Depth > 1
    A1 = double(A(:,:,1));
else
    A1 = double(A);
end
Qtype = 'uniform'; % 'uniform' and 'nonuniform'
B = 5; % # bits of quantization
L = 2^B;
[alfa,E] = LinearPredict(A1);% design the linear predictor
switch Qtype
    case 'uniform'
```

```
        dMin = mean2(E) - 5*std2(E);
        dMax = mean2(E) + 5*std2(E);
        q = 2*dMax/L;
        q2 = q/2;
        dR = linspace(dMin,dMax,L+1);
        rL = zeros(L,1);
        for k = 1:L
            rL(k) = dR(k)+q2;
        end
    case 'nonuniform'
        [DR,C] = dpcmQuantizer(E,B);% design a B-bit non-uniform
          quantizer
end
%
Mu = mean2(A1);% mean value of the image
% Implement the DPCM
pe = zeros(Height,Width);% array to store differential image
A2 = zeros(Height,Width);% array to store reconstructed image
peq = zeros(Height,Width);% array to store quantized
      differential image
A1 = A1 - Mu;% mean removed input image
for r = 1:Height
    A2(r,1) = A1(r,1) + Mu; % first pixel unquantized
    x1 = A1(r,1); % previous pixel
    for c = 2:Width
        xhat = alfa*x1; % predicted pixel
        pe(r,c) = A1(r,c) - xhat; % differential pixel
        switch Qtype
            case 'uniform'
                for k = 1:L
                    if pe(r,c)>dR(k) && pe(r,c)<=dR(k+1)
                        peq(r,c) = rL(k);
                    elseif pe(r,c)<= dR(1)
                            peq(r,c) = rL(1);
                    elseif pe(r,c) > dR(L+1)
                        peq(r,c) = rL(L);
                    end
                end
            case 'nonuniform'
                for k = 1:L
                    if (pe(r,c)>DR(k) && pe(r,c)<=DR(k+1))
                            peq(r,c) = C(k);
                    end
                end
        end
        x1 = peq(r,c) + xhat; % previously reconstructed pixel
        A2(r,c) = x1 + Mu;% mean added reconstructed pixel
    end
end
```

```
% display histogram of quantized & unquantized differential images
[H,B] = hist(pe(:),512);
H = H/sum(H);
[H1,B1] = hist(peq(:),512);
H1 = H1/sum(H1);
figure,subplot(2,1,1),plot(B,H,'k','LineWidth',2)
title('Histogram of unquantized differential image')
ylabel('relative count')
subplot(2,1,2),plot(B1,H1,'k','LineWidth',2)
title('Histogram of quantized differential image')
xlabel('pixel value'),ylabel('relative count')
% display the reconstructed image
figure,imshow(A2,[]), title('Reconstructed image')
% Compute Gp and rate reduction
%Evar = sum(sum(pe(:,2:Width).* pe(:,2:Width)))/(Height*(Width-1));
Evar = sum(sum(pe.* pe))/(Height*Width);
SigVar = (std2(A1+Mu))^2;
sprintf('Pred. Coeff. = %5.4f\n',alfa)
Gp = 10*log10(SigVar/Evar);
RateRed = 0.5*log2(SigVar/Evar);
sprintf('Gp = %4.2f dB\tRateRed = %4.2f bits\n',Gp,RateRed)
SNR = 10*log10(SigVar/((std2(A1+Mu - A2))^2));
sprintf('SNR = %4.2f\n',SNR)


function [alfa,pe] = LinearPredict(I)
% [alfa,pe] = LinearPredict(I)
% I = input intensity image
% n = predictor order
% alfa = predictor coefficient vector
% pe = prediction error image of the same size as I
% Uses 1st-order optimal linear prediction to
% create the differential image


[Height,Width,Depth] = size(I);
if Depth > 1
    I1 = double(I(:,:,1));
else
    I1 = double(I);
end
%
Mu = mean2(I1);% mean value of the image
I1 = I1 - Mu;% mean removed input image
R0 = mean2(I1 .* I1);
R1 = sum(sum(I1(:,2:Width).* I1(:,1:Width-1)))/(Height*(Width-1));
alfa = R1/R0;
% Implement the Linear Predictor
pe = zeros(Height,Width);% array to store prediction errors
```

```
I1 = padarray(I1,[0 1],'symmetric','pre');
for r = 1:Height
    for c = 2:Width
        xhat = alfa*I1(r,c-1);% predicted pixel
        pe(r,c) = I1(r,c) - xhat;% differential pixel
    end
end


function [DR,C] = dpcmQuantizer(x,b)
%   [DR,C] = dpcmQuantizer(x,b)
% Design of a non-uniform quantizer for the DPCM
% x = input differential image
% b = number of bits of quantization
% DR = input decision boundaries
% C = output reconstruction levels


[Height,Width] = size(x);
L = 2^b;
C = linspace(min(x(:)),max(x(:)),L);
%C = linspace(min(x(:))+2,max(x(:))-2,L);
PartitionRegion = zeros(L,Height,Width);
RegionCount = zeros(L,1);
RegionSum = zeros(L,1);
Distortion = zeros(L,1);
TotalDistortion = 1.0e+5; % total minimum distortion
err = 0.1e-01; % accuracy to terminate iteration
TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
% start iteration
It = 1; % iteration number
while TD >= err
    for r = 1:Height
        for c = 1:Width
            d1 = abs(x(r,c)-C(1));
            J = 1;
            for k = 2:L
                d2 = abs(x(r,c)-C(k));
                if d2<d1
                    d1 = d2;
                    J = k;
                end
            end
            RegionCount(J) = RegionCount(J) + 1;
            RegionSum(J) = RegionSum(J) + x(r,c);
            PartitionRegion(J,r,c) = x(r,c);
            Distortion(J) = Distortion(J)+(x(r,c)-C(J))*
                                          (x(r,c)-C(J));
        end
    end
    Index = logical(RegionCount == 0);% check for empty regions
    RegionCount(Index) = 1;% set empty regions to count 1
    Distortion = Distortion ./RegionCount;% average minimum
       distortions
    %TD = abs(TotalDistortion-sum(Distortion));
    % alternate convergence measure
    TD = (TotalDistortion-sum(Distortion))/TotalDistortion;
    C = RegionSum ./RegionCount; % refined output levels
    TotalDistortion = sum(Distortion);% total minimum distortion
    It = It + 1;
end
clear Index
DR = zeros(L+1,1);
DR(L+1) = 255;
for k = 2:L
    DR(k) = (C(k-1)+C(k))/2;
end
```

6.2.2 Two-Dimensional DPCM

As we have seen before, a pixel in intensity images has correlation in both spatial dimensions. Therefore, it is meaningful to exploit the two-dimensional (2D) correlation to achieve a higher compression. This may be accomplished by using a 2D linear predictor in the DPCM loop [8–11]. Let the 2D first-order predicted pixel be expressed as

$$\begin{aligned}\hat{x}[m,n] &= \alpha_1 x[m,n-1] + \alpha_2 x[m-1,n] + \alpha_3 x[m-1,n-1] \\ &\quad + \alpha_4 x[m-1,n+1] \end{aligned} \tag{6.39}$$

Equation (6.39) refers to a causal prediction (Figure 6.7) [7]. Then the optimal 2D predictor is one which has the minimum MSE. That is,

$$E\left\{e^2[m,n]\right\} = E\left\{\begin{pmatrix} x[m,n] - \alpha_1 x[m,n-1] - \alpha_2 x[m-1,n] \\ -\alpha_3 x[m-1,n-1] - \alpha_4 x[m-1,n+1] \end{pmatrix}^2\right\} \tag{6.40}$$

is a minimum. As in the case of one-dimensional (1D) linear prediction, the differential or error $e[m,n]$ is orthogonal to $x[m-i,n-j], 0 \le i, j \le 1$ as well as $x[m-1,n+1]$ and, as a result, we can write

$$E\{e[m,n]\,x[m-i,n-j]\} = 0, 0 \le i, j \le 1 \tag{6.41a}$$

$$E\{e[m,n]\,x[m-1,n+1]\} = 0, \tag{6.41b}$$

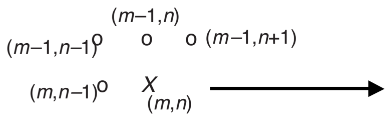


Figure 6.7 A first-order 2D linear predictor showing the pixel ordering.

Since

$$e[m,n] = x[m,n] - \alpha_1 x[m,n-1] - \alpha_2 x[m-1,n] \\ -\alpha_3 x[m-1,n-1] - \alpha_4 x[m-1,n+1] \tag{6.42}$$

Equation (6.41) yields the following equation in matrix form:

$$\begin{bmatrix} r_x(0,0) & r_x(1,1) & r_x(1,0) & r_x(1,-2) \\ r_x(1,1) & r_x(0,0) & r_x(0,1) & r_x(0,-1) \\ r_x(1,0) & r_x(0,1) & r_x(0,0) & r_x(0,2) \\ r_x(1,2) & r_x(0,-1) & r_x(0,2) & r_x(0,0) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} r_x(0,1) \\ r_x(1,0) \\ r_x(1,1) \\ r_x(1,-1) \end{bmatrix} \tag{6.43}$$

where

$$r_x(i,j) = E\{x[m,n]\,x[m-i,n-j]\}, 0 \le i \le 1, 0 \le j \le 2 \tag{6.44}$$

is the covariance function of the image in question. The covariance function if separable, can be written as

$$r(m,n) = r_1(m)\,r_2(n) \tag{6.45}$$

An often used model of a separable image covariance function is of the form

$$r(m,n) = \sigma_x^2 \rho_1^{|m|} \rho_2^{|n|}, |\rho_i| \prec 1, i = 1, 2 \tag{6.46}$$

where σ_x^2 is the variance of the image, $\rho_1 = r(1,0)/\sigma_x^2$ and $\rho_2 = r(0,1)/\sigma_x^2$ are the correlation coefficients along the two spatial dimensions. Then, equation (6.43) reduces to

$$\begin{bmatrix} 1 & \rho_1\rho_2 & \rho_1 \\ \rho_1\rho_2 & 1 & \rho_2 \\ \rho_1 & \rho_2 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} \rho_2 \\ \rho_1 \\ \rho_1\rho_2 \end{bmatrix} \tag{6.47}$$

From equation (6.47), we find that the 2D predictor coefficients are given by $\alpha_1 = \rho_1, \alpha_2 = \rho_2$, and $\alpha_3 = -\rho_1\rho_2$. The resulting minimum MSE can be determined to be

$$\begin{aligned}\mathrm{MSE}_{\min} &= E\{e[m,n]\,x[m,n]\}\\ &= r_x(0,0) - \alpha_{10}r_x(1,0) - \alpha_{01}r_x(0,1) - \alpha_{11}r_x(1,1)\\ &= \sigma_x^2\left(1-\rho_1^2-\rho_2^2+\rho_1^2\rho_2^2\right)\end{aligned} \tag{6.48}$$

Therefore, the prediction gain of the separable 2D DPCM is found to be

$$G_{\mathrm{P}} = \frac{\sigma_x^2}{\mathrm{MSE}_{\min}} = \frac{1}{1-\rho_1^2-\rho_2^2+\rho_1^2\rho_2^2} = \frac{1}{\left(1-\rho_1^2\right)\left(1-\rho_2^2\right)} \tag{6.49}$$

which implies that the prediction gain is the product of the prediction gains along the two dimensions. Let us illustrate the difference between the 1D and 2D DPCM by the following example.

Example 6.2 Implement the 2D DPCM using the predictor defined in equation (6.39) for the cameraman image. Compare its performance with the 1D DPCM at 4 bpp. Use a uniform quantizer.

Solution The 2D linear prediction to be used is given by the following equation:

$$\begin{aligned}\hat{x}[m,n] = \alpha_1 x[m,n-1] + \alpha_2 x[m-1,n] + \alpha_3 x[m-1,n-1]\\ +\alpha_4 x[m-1,n+1]\end{aligned}$$

In order to determine the predictor coefficients analytically, we will use the actual image pixels rather than the quantized and reconstructed pixels. For the cameraman image, the predictor coefficients are found to be $\alpha_1 = 0.6252$, $\alpha_2 = 0.7232$, $\alpha_3 = -0.4271$, and $\alpha_4 = 0.0713$. Once the predictor is designed, we next have to design the uniform quantizer. Since the prediction error image has a Laplacian PDF, the theoretical minimum and maximum values of the prediction error are strictly negative infinity and infinity, respectively. For the given image, the lower and upper limits for the prediction error are -161 and 180, respectively. However, the probability of an error pixel having the extreme value is about 0.002%. It is, therefore, efficient to set the extreme values to plus and minus 3 to 5 times the standard deviation. A value of 5 standard deviations proves to yield good visual quality at 4 bpp. Thus, the quantization step size with L levels is $\Delta = 2 \times (\mu + 5\sigma)/L$. Of course, the mean prediction error is zero, as can be seen from the histogram of the prediction error

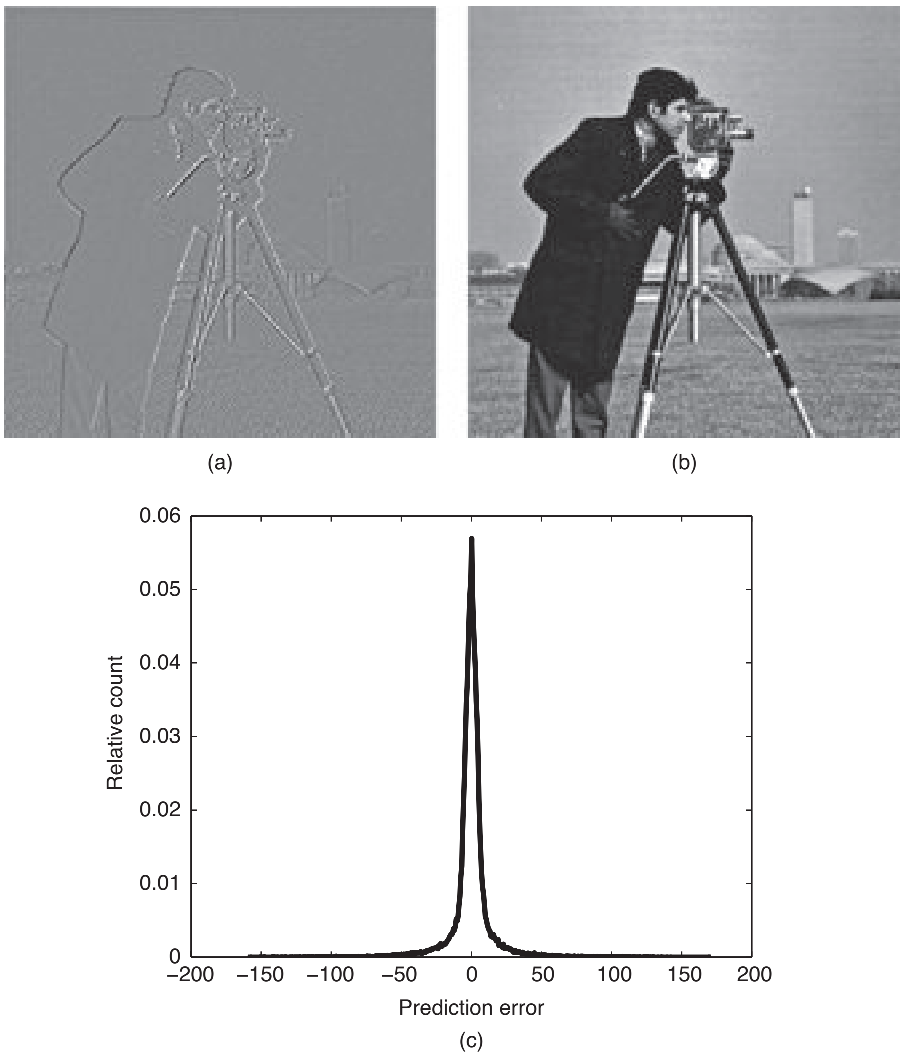


Figure 6.8 Result of 2D DPCM encoding of Example 6.2 using uniform quantization: (a) differential image, (b) reconstructed image at 4 bpp with SNR = 23.87 dB, and (c) histogram of the prediction error.

(Figure 6.8c). The step size with $L = 16$ comes out to be 8.92. The SNR at 4 bpp is found to be 23.87 dB with good visual quality. The differential image and the reconstructed image at 4 bpp are shown in Figures 6.8a,b, respectively. As a comparison to the 1D DPCM, Figures 6.9a,b display the differential and reconstructed images at 4 bpp, respectively. The SNR for the 1D case is 21.28 dB. One can see some streaks in the black coat, which are not visible in the 2D case. Figure 6.10 shows

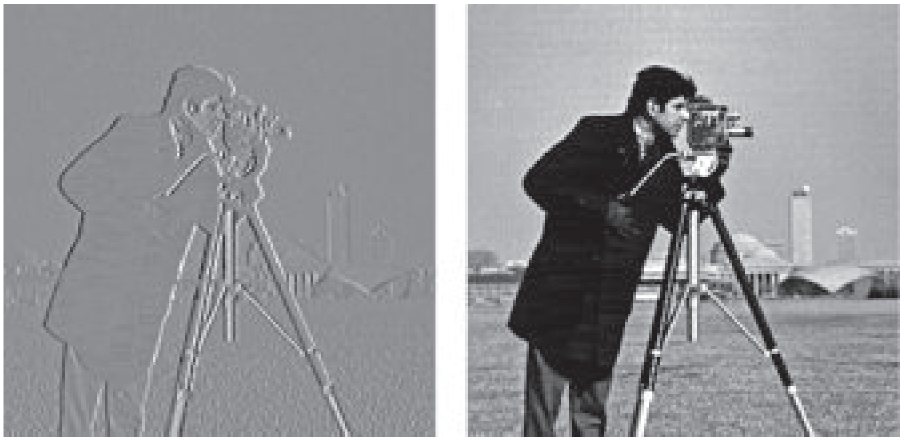


Figure 6.9 Result of 1D DPCM encoding using uniform quantization at 4 bpp: (a) differential image and (b) reconstructed image with SNR = 21.28 dB.

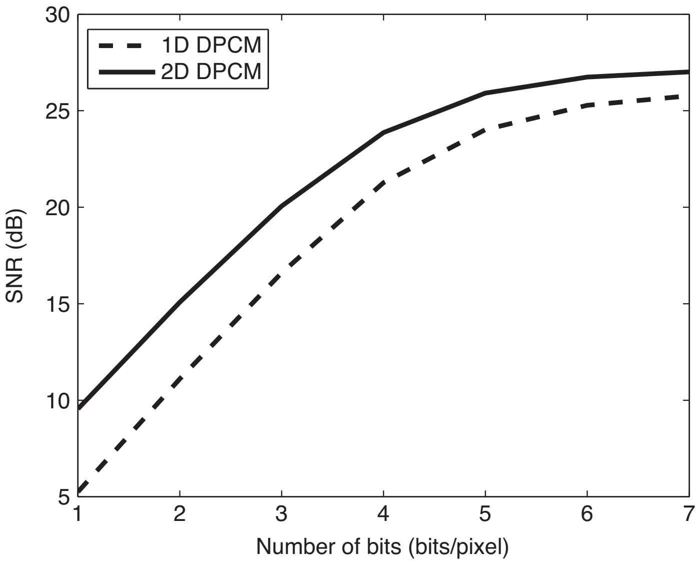


Figure 6.10 Comparison of SNR for 1D and 2D DPCM coders of Example 6.2.

the SNR in dB versus the bpp for both 1D and 2D DPCMs. There is a 3 dB gain at 3 bpp and a 2.59 dB gain at 4 bpp for the 2D DPCM, though the 2D implementation is not that much complex.

```
% Example6_2.m
% Design and implement the 2D DPCM where
% xhat = a1*x(m,n-1) + a2*x(m-1,n) + a3*x(m-1,n-1)...
% + a4*x(m-1,n+1)is the predictor
% Use a uniform quantizer with 4 bits of quantization

clear, %close all
A = imread('cameraman.tif');
[Height,Width,Depth] = size(A);
if Depth > 1
    A1 = double(A(:,:,1));
else
    A1 = double(A);
end
%
Qtype = 'uniform'; % 'uniform' and 'nonuniform'
B = 5; % # bits of quantization
DPCM_2D(A1,Qtype,B);

function DPCM_2D(f,Qtype,B)
% DPCM_2D(f,Qtype,B)
% Designs and implements 2D DPCM where
% xhat = a1*x(m,n-1) + a2*x(m-1,n) + a3*x(m-1,n-1)...
% + a4*x(m-1,n+1)is the predictor
%
% Input:
%   f = intensity image to be encoded
%   Qtype = quantizer type: "uniform" or "nonuniform"
%   B = # bits of quantization
%
%   alfa = predictor coefficient
%   E = unquantized prediction error image
%   pe = unquantized differential image
%   peq = quantized differential image
%   y = DPCM reconstructed image
%
% This function implements only the encoder.


L = 2^B; % # levels in the quantizer
[Height,Width] = size(f);
%
% compute optimal predictor coefficients
[alfa,E] = LinearPredict_2D(f);
%
```

```
% Design the uniform quantizer using 5*std dev as the limits
switch Qtype
    case 'uniform'
        dMin = mean2(E) - 5*std2(E);
        dMax = mean2(E) + 5*std2(E);
        q = 2*dMax/L; % step size
        q2 = q/2;
        dR = linspace(dMin,dMax,L+1); % decision intervals
        rL = zeros(L,1); % reconstruction levels
        for k = 1:L
            rL(k) = dR(k)+q2;
        end
    case 'nonuniform'
        [DR,C] = dpcmQuantizer(E,B);% design a B-bit
                 non-uniform quantizer
end
Mu = mean2(f);% mean value of the image
f = f - Mu;% remove mean value
% Implement the 2D DPCM
y = zeros(Height,Width);% array to store reconstructed image
pe = zeros(Height,Width);% array to store differential image
peq = zeros(Height,Width);% array to store quantized
      differential image
x1 = zeros(Height+1,Width+2); % array to store
     reconstructed image
y(1,:) = f(1,:) + Mu;
y(:,1) = f(:,1) + Mu;
%
f = padarray(f,[1 2],'symmetric','pre');
% First row, first column no prediction
x1(1,:) = f(1,:);% store previously reconstructed pixels
x1(:,1) = f(:,1);
for r = 2:Height
    for c = 2:Width
        xhat = alfa(1)*x1(r,c-1) + alfa(2)*x1(r-1,c) + ...
                alfa(3)*x1(r-1,c-1)+ alfa(4)*x1(r-1,c+1);
        pe(r,c) = f(r,c) - xhat;
        switch Qtype
            case 'uniform'
                for k = 1:L
                    if pe(r,c)>dR(k) && pe(r,c)<=dR(k+1)
                        peq(r,c) = rL(k);
                    elseif pe(r,c)<= dR(1)
                            peq(r,c) = rL(1);
                    elseif pe(r,c) > dR(L+1)
                        peq(r,c) = rL(L);
                    end
                end
            case 'nonuniform'
```

```
                %{
                for k = 1:L
                    if (pe(r,c)>DR(k) && pe(r,c)<=DR(k+1))
                            peq(r,c) = C(k);
                    end
                end
                %}
                d1 = abs(pe(r,c)-C(1));
                for k = 2:L
                    d2 = abs(pe(r,c)-C(k));
                    if d2<d1
                        d1 = d2;
                        J = k;
                    end
                end
                peq(r,c) = C(J);
        end
        x1(r,c) = peq(r,c) + xhat;% previously
                  reconstructed pixel
        y(r,c) = x1(r,c) + Mu;% mean added reconstructed pixel
    end
end
% Display differential and reconstructed images
figure,imshow(pe,[]), title('Differential image')
figure,imshow(y,[]), title('Reconstructed image')
[H,B] = hist(pe(:),512); % Histogram of differential image
H = H/sum(H);
[H1,B1] = hist(peq(:),512); % Histogram of quantized
          differential image
H1 = H1/sum(H1);
figure,subplot(2,1,1),plot(B,H,'k','LineWidth',2)
title('Histogram of unquantized differential image')
ylabel('relative count')
subplot(2,1,2),plot(B1,H1,'k','LineWidth',2)
title('Histogram of quantized differential image')
xlabel('pixel value'),ylabel('relative count')
% Compute SNR
SigVar = (std2(f(1:Height,1:Width)))^2;
SNR = 10*log10(SigVar/(std2(f(1:Height,1:Width) - y)^2));
sprintf('SNR = %4.2f\n',SNR)
function [alfa,pe] = LinearPredict_2D(I)
% [alfa,pe] = LinearPredict_2D(I)
% I = input intensity image
% alfa = predictor coefficient vector
% pe = prediction error image of the same size as I
% Uses 2D linear prediction to
% create the differential image
% xhat(m,n) = alfa1(1)*x(m,n-1)+alfa(2)*x(m-1,n)+...
%    alfa(3)*x(m-1,n-1)+alfa(4)*x(m-1,n+1)
```

```
[Height,Width,Depth] = size(I);
if Depth > 1
    I1 = double(I(:,:,1));
else
    I1 = double(I);
end
%
Mu = mean2(I1);% mean value of the image
I1 = I1 - Mu;% mean removed input image
r00 = mean2(I1 .* I1);
r01 = sum(sum(I1(:,2:Width).* I1(:,1:Width-1)))/
      (Height*(Width-1));
r10 = sum(sum(I1(2:Height,:).* I1(1:Height-1,:)))/
      ((Height-1)*Width);
r11 = sum(sum(I1(2:Height,2:Width).* I1(1:Height-1,1:
      Width-1)))/((Height-1)*(Width-1));
r12 = sum(sum(I1(2:Height,3:Width).* I1(1:Height-1,1:
      Width-2)))/((Height-1)*(Width-2));
r02 = sum(sum(I1(1:Height,3:Width).* I1(1:Height,1:Width-2)))/
      (Height*(Width-2));
alfa = inv([r00 r11 r10 r12; r11 r00 r01 r01;...
    r10 r01 r00 r02; r12 r01 r02 r00])*[r01 r10 r11 r11]';
% Implement the Linear Predictor
pe = zeros(Height,Width);% array to store prediction errors
%I1 = padarray(I1,[1 1],'symmetric','pre');
I1 = padarray(I1,[1 2],'symmetric','pre');
for r = 2:Height+1
    for c = 2:Width+1
        xhat = alfa(1)*I1(r,c-1)+alfa(2)*I1(r-1,c)+ ...
            alfa(3)*I1(r-1,c-1) + alfa(4)*I1(r-1,c+1);
        pe(r-1,c-1) = I1(r,c) - xhat;% differential pixel
    end
end
```

6.3 ADAPTIVE DPCM

A DPCM works on the basis that pixels in a neighborhood are highly correlated and that the differential pixels have a relatively small range of values. A fixed quantizer, uniform or nonuniform, will perform adequately because the statistics of the differential image remain essentially unchanged. An intensity image may have several regions with finer details. In such cases, the statistics of the differential image will change according to the details in the image. Therefore, a fixed quantizer will not be efficient. That is to say that a better performance could be achieved if the quantizer somehow adapts to the changing statistics of the image being encoded via a DPCM. When a DPCM adapts itself to the changing statistics of the input image, it is called an *adaptive DPCM* (ADPCM) [12]. As we will see in this section, an ADPCM can achieve significant improvements over a non-ADPCM in terms of SNR, or equivalently in terms of bit rate reduction.

There are two possible approaches to the design of an ADPCM. In one case, we can use a fixed quantizer but adapt its input to produce a unit variance differential pixel. This is known as the *adaptive gain control* DPCM [13, 14]. In the second case, one can use a set of fixed quantizers and choose a particular one to quantize a given differential pixel based on the class into which the given pixel falls. This is known as the *adaptively classified* DPCM [15].

6.3.1 Adaptive Gain Control DPCM

Figure 6.11 shows the ADPCM in which the input to the quantizer is adapted to always produce a unit variance differential signal. Basically, based on the previously quantized differential pixel, an estimate of the variance of the current difference pixel is obtained. The quantizer input is then divided by the square root of the estimated variance so that the input difference pixel has a unit variance. Thus, the quantizer remains fixed while its input is adjusted from pixel to pixel to have a unit variance.

Gain Adaptation Rule
For the current pixel, the variance $\sigma_e^2(k)$ of the differential pixel at the quantizer input is estimated from the previous variance and the quantized differential pixel as given by

$$\sigma_e^2(k) = (1-\gamma)\,e_q^2(k-1) + \gamma\sigma_e^2(k-1) \tag{6.50}$$

where $0 \le \gamma \le 1$ and refer to Figure 6.11 for other quantities in equation (6.50).

We will illustrate the implementation of a gain-controlled ADPCM using a real intensity image in the following example.

Example 6.3 Design a gain-controlled 1D DPCM to encode an intensity image. Use a uniform quantizer and calculate the resulting SNR at 3 bpp. Compare the results with a 2D gain-controlled DPCM at the same bit rate of a uniform quantizer.

Solution Let us use the same cameraman image in this example. The ADPCM for this example is shown in Figure 6.11. The predictor corresponds to the 1D case. Figure 6.12a is the unquantized differential image of the ADPCM and the quantized version is shown in Figure 6.12b. Notice the slat & pepper type of grain in the quantized differential image. This shows that the differential image is nearly a white noise. The reconstructed image at 3 bpp is shown in Figure 6.12c. The SNR at 3 bpp for the 1D ADPCM is 26.80 dB with excellent visual quality.

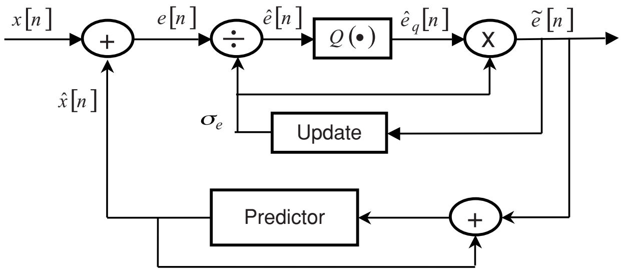


Figure 6.11 Block diagram of a gain-controlled ADPCM. The linear predictor may be 1D or 2D.

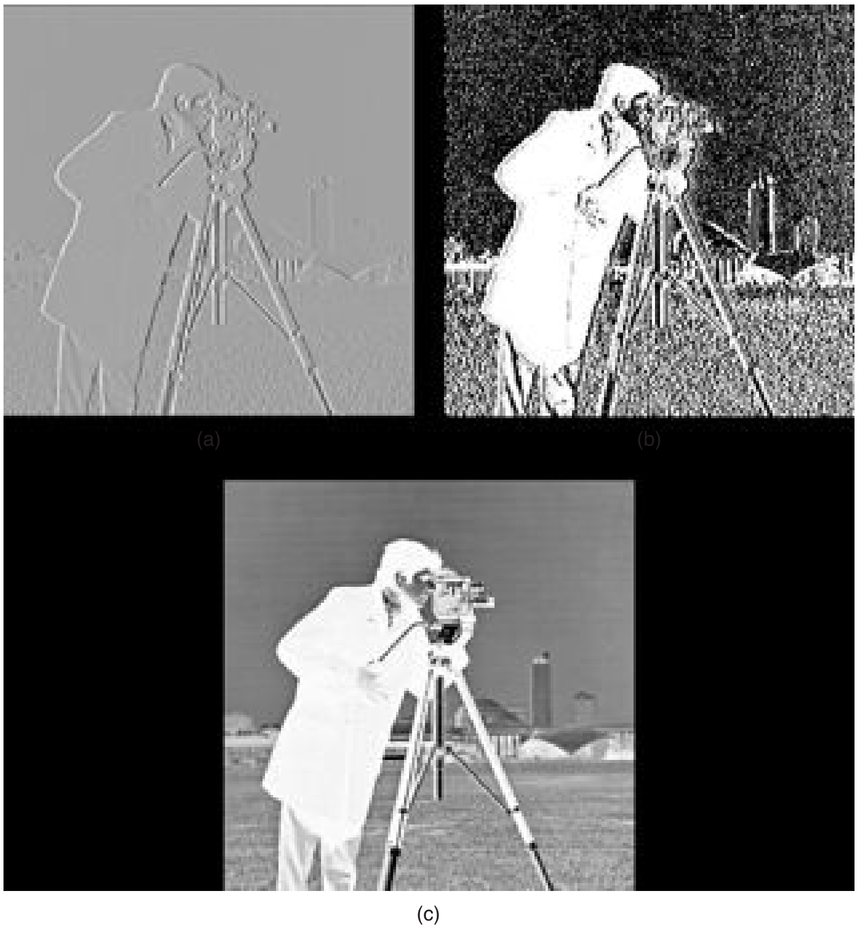
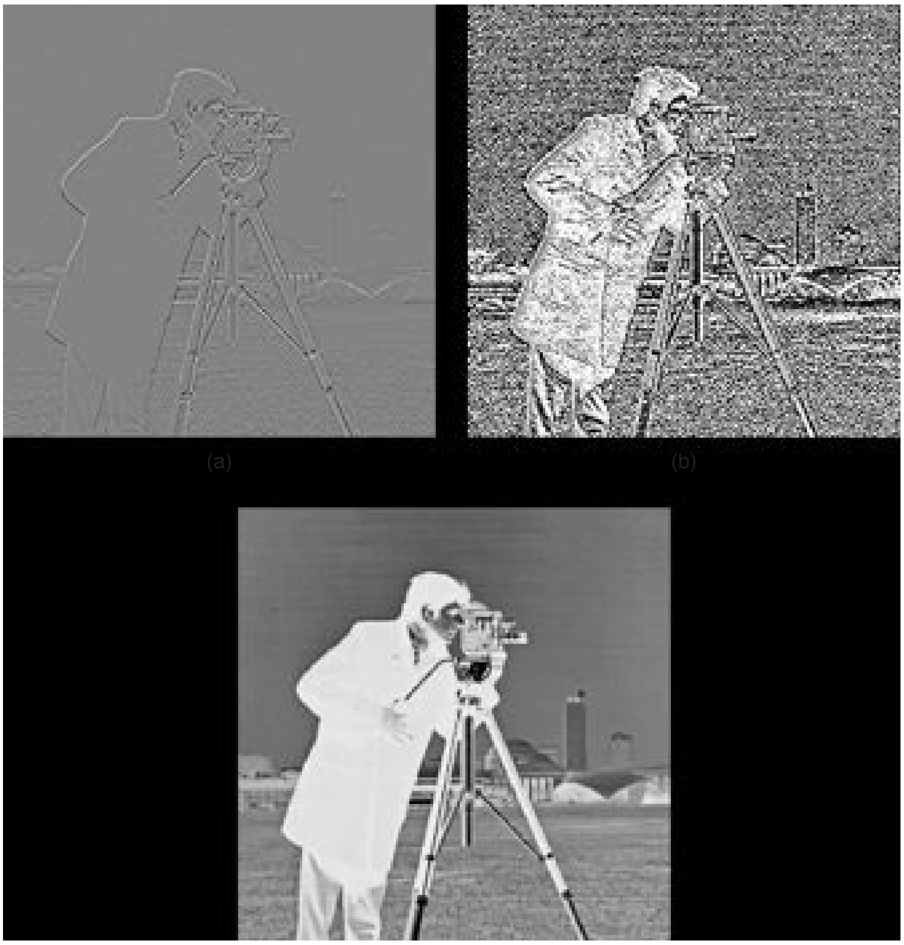


Figure 6.12 Results of encoding the cameraman image with the 1D ADPCM of Figure 6.11: (a) unquantized differential image using 1D ADPCM, (b) quantized differential image using a 3-bit uniform quantizer, and (c) reconstructed image with an SNR of 26.80 dB.



(c)

Figure 6.13 Results of encoding the cameraman image with the 2D ADPCM of Figure 6.11: (a) unquantized differential image using 2D ADPCM, (b) quantized differential image using a 3-bit uniform quantizer, and (c) reconstructed image with an SNR of 29.52 dB.

The same ADPCM is implemented with 2D linear prediction referring to equation (6.39) and the unquantized, quantized versions of the differential images as well as the reconstructed image are shown in Figures 6.13a–c, respectively. The quantized differential image in the 2D case is much more noise-like than that in the 1D case. This is because the 2D prediction removes a greater amount of pixel redundancy by exploiting 2D correlation in the image. This naturally results in a higher SNR (29.52 dB) at the same bit rate (3 bpp) as in the 1D case. It is important to compare the result with the 1D *non*-ADPCM, where we found that even the nonuniform quantizer at

5 bpp yields only 24.83 dB of SNR. Thus, the ADPCM outperforms its nonadaptive counterpart without much complexity.

```
% Example6_3.m
% Gain controlled DPCM
% Implements both 1D and 2D DPCM with a uniform quantizer
% whose input variance is adapted to the changing
% statistics of the differential input to the
% quantizer.

clear, %close all
A = imread('cameraman.tif');
B = 3; % # bits of quantization
dpcmType = '2D'; % options: "1D" and "2D".
% Call ADPCM function to implement the gain controlled DPCM
ADPCM_Gain(A,dpcmType,B);
function ADPCM_Gain(A,dpcmType,B)
% ADPCM_Gain(A,dpcmType)
% Designs a 1st-order adaptive 1D DPCM (encoder only).
% The prediction error variance is adapted so that the
% uniform quantizer is fixed. The adaptation procedure
% updates the current error pixel variance and accordingly
% adjusts the quantizer input (refer to Fig.6-11).

[Height,Width,Depth] = size(A);
if Depth > 1
    A1 = double(A(:,:,1));
else
    A1 = double(A);
end
%B = 3; % # bits of quantization
L = 2^B; % # levels of quantization
switch dpcmType
    case '1D'
        [alfa,E] = LinearPredict_1D(A1);% design the 1D
                   1st-order predictor
    case '2D'
        [alfa,E] = LinearPredict_2D(A1); % design the
                   2nd-order predictor
end
% Design a uniform quantizer with input as the prediction error
% with unit variance
E = E/std2(E);
dMin = mean2(E) - 3*std2(E);
dMax = mean2(E) + 3*std2(E);
q = 2*dMax/L;
q2 = q/2;
dR = linspace(dMin,dMax,L+1);
rL = zeros(L,1);
for k = 1:L
    rL(k) = dR(k)+q2;
end
```

```
% Adaptive rule: sigma(k)^2 = (1-gama)/(1-fB)*sigma(k-1)^2
                              +gama*e^2
% Coefficients of adaptive algorithm
gama = 0.2; %b = 1.1711; fB = 2^(-b*B);
%a1 = (1-gama)/(1-fB);
a1 = (1-gama);
%
% Implement the DPCM
Mu = mean2(A1);% mean value of the image
pe = zeros(Height,Width);% array to store differential image
A2 = zeros(Height,Width);% array to store reconstructed image
peq = zeros(Height,Width);% array to store quantized
      differential image
A1 = A1 - Mu;% mean removed input image
SigVar = (std2(A1))^2;
switch dpcmType
    case '1D'
        for r = 1:Height
            A2(r,1) = A1(r,1) + Mu; % first pixel unquantized
            x1 = A1(r,1); % previous pixel
            varE = 0; % initial estimate of prediction error
                   variance
            for c = 2:Width
                xhat = alfa*x1; % predicted pixel
                pe(r,c) = A1(r,c) - xhat; % differential pixel
                varE = a1*pe(r,c)*pe(r,c) + gama*varE;
                sigmaE = sqrt(varE);
                e1 = pe(r,c)/sigmaE;
                for k = 1:L
                    if e1>dR(k) && e1<=dR(k+1)
                        peq(r,c) = rL(k);
                    elseif e1<= dR(1)
                            peq(r,c) = rL(1);
                    elseif e1 > dR(L+1)
                        peq(r,c) = rL(L);
                    end
                end
                x1 = peq(r,c)*sigmaE + xhat; % previously
                     reconstructed pixel
                A2(r,c) = x1 + Mu;% mean added reconstructed
                          pixel
            end
        end
        SNR = 10*log10(SigVar/((std2(A1 - A2))^2));
    case '2D'
        x1 = zeros(Height+1,Width+2); % array to store
             reconstructed image
        A2(1,:) = A1(1,:) + Mu;
        A2(:,1) = A1(:,1) + Mu;
        A1 = padarray(A1,[1 2],'symmetric','pre');
        % First row, first column no prediction
        x1(1,:) = A1(1,:);% store previously reconstructed pixels
        x1(:,1) = A1(:,1);
        for r = 2:Height
            varE = 0;
            for c = 2:Width
                xhat = alfa(1)*x1(r,c-1) + alfa(2)*x1
                       (r-1,c) + ...
                        alfa(3)*x1(r-1,c-1)+ alfa(4)*x1(r-1,c+1);
                pe(r,c) = A1(r,c) - xhat;
                varE = a1*pe(r,c)*pe(r,c) + gama*varE;
                sigmaE = sqrt(varE);
                e1 = pe(r,c)/sigmaE;
                for k = 1:L
                    if e1>dR(k) && e1<=dR(k+1)
                        peq(r,c) = rL(k);
                    elseif e1<= dR(1)
                            peq(r,c) = rL(1);
                    elseif e1 > dR(L+1)
                        peq(r,c) = rL(L);
                    end
                end
                x1(r,c) = peq(r,c)*sigmaE + xhat;% previously
                          reconstructed pixel
                A2(r,c) = x1(r,c) + Mu;% mean added
                          reconstructed pixel
            end
        end
        SNR = 10*log10(SigVar/(std2(A1(1:Height,1:Width) -
                                    A2)^2));
end
% display histogram of quantized & unquantized differential
% images
[H,B] = hist(pe(:),512);
H = H/sum(H);
[H1,B1] = hist(peq(:),512);
H1 = H1/sum(H1);
figure,subplot(2,1,1),plot(B,H,'k','LineWidth',2)
title(['Histogram of unquantized differential
       image: ' dpcmType])
ylabel('relative count')
subplot(2,1,2),plot(B1,H1,'k','LineWidth',2)
title(['Histogram of quantized differential
       image: ' dpcmType])
xlabel('pixel value'),ylabel('relative count')
% display the reconstructed image & calculate SNR
figure,imshow(pe,[]), title(['Unquantized differential im-
age: ' dpcmType])
figure,imshow(peq,[]), title(['quantized differential im-
age: ' dpcmType])
figure,imshow(A2,[]), title(['Reconstructed image: ' dpcmType])
sprintf('SNR = %4.2f\n',SNR)
```

6.3.2 Adaptive Pixel Classifier DPCM

An alternative method of adapting the DPCM to changing statistics (nonstationary) of the input image is to use different quantizers to quantize the differential pixel according to its class or type [14, 16]. Even though the quantizers are fixed, they have different number of bits of quantization as well as different input ranges and output levels. The crucial element in this ADPCM is the classifier. The current pixel is classified into one of C classes and the corresponding differential pixel is quantized using the quantizer belonging to that class. There are several possible classification rules to choose. One such rule is to use the variance of the neighborhood of pixels with the current pixel at its center. Another classification rule is to use the local neighborhood contrast. In any case, the chosen measure (variance, for instance) is divided into intervals and the pixel class is decided on the basis of the interval in which the current pixel measure falls. Then the corresponding quantizer is chosen to quantize the differential pixel. The decoder must know which quantizer was used for each pixel so that the image could be reconstructed. This requires additional or side information to be conveyed to the decoder (see Figure 6.14).

Example 6.4 Design a 1D ADPCM using pixel classifier with three classes and three uniform quantizers, one for each classifier. Choose the neighborhood pixel variance as the metric to classify a pixel. Implement the resulting DPCM and encode a real intensity image.

Solution Figure 6.14 shows the ADPCM, where $C = 3$. The three uniform quantizers have 3, 7, and 6 bits of quantization, respectively. The classifier classifies a pixel as belonging to class 1 if the pixel neighborhood variance is in the interval $(0, 9]$, class 2 if its neighborhood variance is in the interval $(9, 1225]$, and class 3 otherwise. The percentage of each class for the cameraman image is found to be 50.84%,

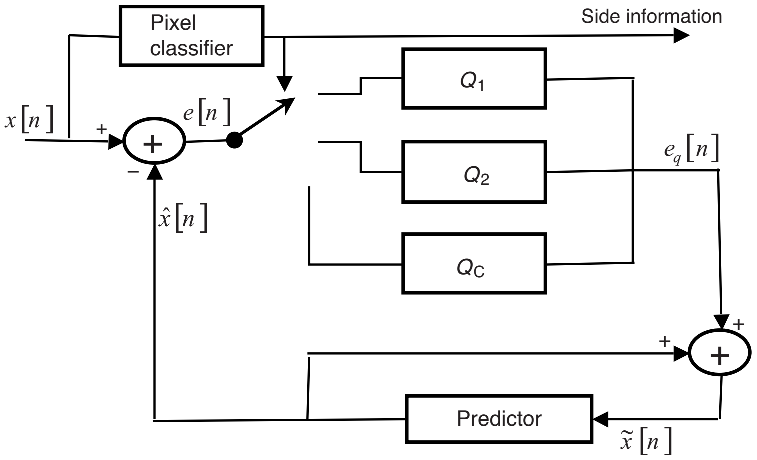


Figure 6.14 Block diagram of a classified ADPCM using 1D linear prediction.

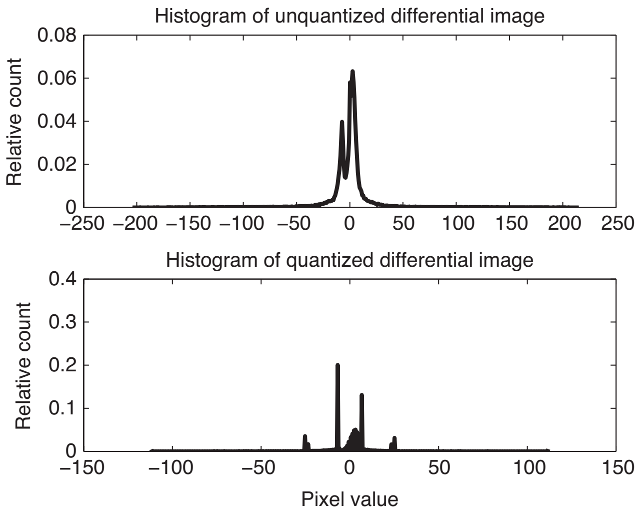


Figure 6.15 Histograms of the differential images of Example 6.4. The top plot is the histogram of the unquantized differential image, and the bottom plot is that of the quantized differential image.

35.87%, and 13.29%, respectively. The resulting SNR is 20.13 dB at approximately 5 bpp. The neighborhood size for the calculation of the variance is 5 × 5 pixels. The histograms of the unquantized and quantized differential image are shown in the top and bottom plots of Figure 6.15. The reconstructed image is shown in Figure 6.16. The MATLAB code for this example is listed below.

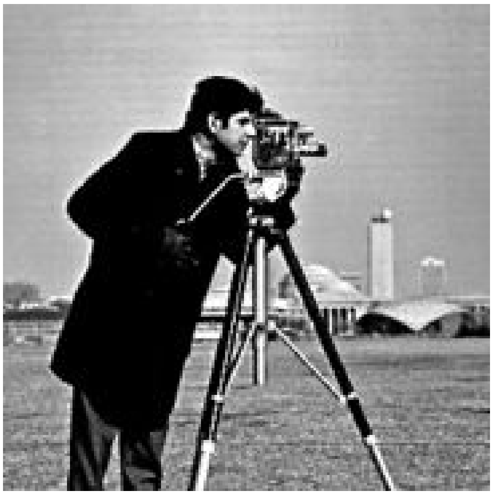


Figure 6.16 Reconstructed image of Example 6.4 with an SNR of 20.13 dB. There are three classes and three uniform quantizers with 3, 7, and 6 bits of quantization.

```
% Example6_4.m
% Designs an ADPCM with pixel classification.
% The function ADPCM_Classify implements the
% 1D DPCM with neighborhood variance as the metric
% to classify a pixel. It uses 3 classes along with
% three uniform quantizers with B(1), B(2), and B(3)
% number of bits of quantization.

clear, %close all
A = imread('cameraman.tif');
B = [3 7 6]; % # bits of quantization for the three quantizers
% Call ADPCM_Classify function to implement the ADPCM
ADPCM_Classify(A,B);

function ADPCM_Classify(A,B)
% ADPCM_Classify(A,B)
% Adaptive 1D DPCM using pixel classifier
% Classification is based on the variance of a
% neighborhood of pixels centered on the current pixel.
% Uses 3 uniform quantizers with different bits
% of quantization.
% Input:
% A = intensity image
% B = 3-vector whose elements are the # bits of each quantizer

[Height,Width,Depth] = size(A);
if Depth > 1
    A1 = double(A(:,:,1));
else
    A1 = double(A);
end
[alfa,E] = LinearPredict_1D(A);
%
% Design 3 uniform quantizers
dMin(1) = 0;
dMax(1) = 0.3*std2(E);
dMin(2) = dMax(1);
dMax(2) = 1.05*std2(E);
dMin(3) = dMax(2);
dMax(3) = 5*std2(E);
%
L(1) = 2^B(1); L(2) = 2^B(2); L(3) = 2^B(3);
q = zeros(3);
q2 = zeros(3);
rL1(1:L(1)) = 0; % reconstruction levels
rL2(1:L(2)) = 0;
rL3(1:L(3)) = 0;
%
q(1) = 2*dMax(1)/L(1);
```

```
q2(1) = q(1)/2;
dR1 = linspace(dMin(1),dMax(1),L(1)+1); % decision intervals
%
for k = 1:L(1)
    rL1(k) = dR1(k)+q2(1);
end
%
q(2) = 2*dMax(2)/L(2);
q2(2) = q(2)/2;
dR2 = linspace(dMin(2),dMax(2),L(2)+1);
%
for k = 1:L(2)
    rL2(k) = dR2(k)+q2(2);
end
%
q(3) = 2*dMax(3)/L(3);
q2(3) = q(3)/2;
dR3 = linspace(dMin(3),dMax(3),L(3)+1);
%
for k = 1:L(3)
    rL3(k) = dR3(k)+q2(3);
end
%
Mu = mean2(A1);% mean value of the image
% Implement the ADPCM
pe = zeros(Height,Width);% array to store differential image
y = zeros(Height,Width);% array to store reconstructed image
peq = zeros(Height,Width);% array to store quantized
      differential image
A1 = A1 - Mu;% mean removed input image
SigVar = (std2(A1))^2; % input signal variance
padRows = 3; padCols = 3;
% extend the input image with padRows and padCols all around
A1 = padarray(A1,[padRows padCols],'symmetric','both');
Pcount = zeros(1,3); % percent of each class
for r = padRows:Height+padRows-1
    y(r-padRows+1,1) = A1(r-padRows+1,1) + Mu; % first pixel
                         unquantized
    x1 = A1(r-padRows+1,1); % previous pixel
    for c = padCols:Width+padCols-1
        xhat = alfa*x1; % predicted pixel
        pe(r,c) = A1(r,c) - xhat; % differential pixel
        % local variance for classification
        BlkVar = std2(A1(r-padRows+1:r+padRows-1,c-padCols+1:
                        c+padCols-1))^2;
        s = 1;
        if pe(r,c) < 0
            s = -1;
        end
        e = abs(pe(r,c));
        if BlkVar < 9 % Class I
            Pcount(1) = Pcount(1) + 1;
            for k = 1:L(1)
                if e>dR1(k) && e<=dR1(k+1)
                    peq(r,c) = s*rL1(k);
                elseif e<= dR1(1)
                        peq(r,c) = s*rL1(1);
                elseif e > dR1(L(1)+1)
                    peq(r,c) = s*rL1(L(1));
                end
            end
        elseif BlkVar >= 9 && BlkVar < 1225 % Class II
            Pcount(2) = Pcount(2) + 1;
            for k = 1:L(2)
                if e>dR2(k) && e<=dR2(k+1)
                    peq(r,c) = s*rL2(k);
                elseif e<= dR2(1)
                        peq(r,c) = s*rL2(1);
                elseif e > dR2(L(2)+1)
                    peq(r,c) = s*rL2(L(2));
                end
            end
        else % Class III
            Pcount(3) = Pcount(3) + 1;
            for k = 1:L(3)
                if e>dR3(k) && e<=dR3(k+1)
                    peq(r,c) = s*rL3(k);
                elseif e<= dR3(1)
                        peq(r,c) = s*rL3(1);
                elseif e > dR3(L(3)+1)
                    peq(r,c) =s* rL3(L(3));
                end
            end
        end
        x1 = peq(r,c) + xhat; % previously reconstructed pixel
        y(r-padRows+1,c-padCols+1) = x1 + Mu;% mean added
                                     reconstructed pixel
    end
end
Pcount = Pcount/sum(Pcount);
% display the reconstructed image & calculate SNR
figure,imshow(y,[]), title('Reconstructed image')
SNR = 10*log10(SigVar/((std2(A1(padRows:Height+padRows-1,
                        padCols:Width+padCols-1) - y))^2));
sprintf('SNR = %4.2f\n',SNR)
sprintf('Class 1 = %5.2f%%\tClass 2 = %5.2f%%\tClass 3 =
%5.2f%%\n',Pcount*100)
%
```

```
% display histogram of quantized & unquantized
% differential images
[H,B] = hist(pe(:),512);
H = H/sum(H);
[H1,B1] = hist(peq(:),512);
H1 = H1/sum(H1);
figure,subplot(2,1,1),plot(B,H,'k','LineWidth',2)
title('Histogram of unquantized differential image')
ylabel('relative count')
subplot(2,1,2),plot(B1,H1,'k','LineWidth',2)
title('Histogram of quantized differential image')
xlabel('pixel value'),ylabel('relative count')
```

6.4 SUMMARY

DPCM is a relatively low complexity encoding scheme for compressing still images and video. In a line-by-line DPCM, a current pixel is estimated from the previously reconstructed pixel in an optimal fashion and the difference between the actual and predicted pixels is quantized and transmitted. The predictor forms a closed loop at the encoder and decoder. The optimality criterion typically used is the minimum MSE between the actual and predicted image pixels. We described a *p*th-order linear predictor and showed how the optimal coefficients and the resulting minimum MSE can be calculated using the orthogonality principle in an elegant manner. The differential image has the typical Laplacian PDF and can be quantized using either the uniform or Lloyd–Max quantizer optimized to the PDF of the differential image. The performance of a 1D DPCM was illustrated with an example giving the various performance measures such as the prediction gain, SNR, and so on. For an actual image (cameraman), the SNR due to DPCM encoding is found to be 24.02 dB at 5 bpp with a uniform quantizer and 24.83 dB at the same bit rate with a nonuniform quantizer.

Since there is correlation among the pixels in both spatial dimensions, a 2D linear predictor performs slightly better than the 1D predictor. A causal 2D predictor model was used to derive the optimal predictor coefficients. Again, an example is used to demonstrate the implementation of a 2D DPCM. At 4 bpp, the 2D DPCM yields an SNR of 23.87 dB with a uniform quantizer. As a comparison, the 1D DPCM yields only 21.28 dB at 4 bpp. So, there is a gain of 2.59 dB for the 2D DPCM with only a slight increase in complexity.

A DPCM (1D or 2D) typically uses fixed quantizers to quantize the differential pixels. Since real images are not stationary, fixing the quantizer design will be suboptimal because the statistics of its input do not remain constant. However, if the quantizer could be adapted to the changing statistics of its input, then one can expect a much better performance from the DPCM. We described two ADPCMs, one, called the gain-controlled ADPCM, where the quantizer input is normalized by the standard deviation of the current pixel and the other, called the classified ADPCM, where a quantizer from a set of fixed quantizers is chosen to quantize a current

Table 6.2 SNR and complexity of the DPCMs

DPCM Type	Bits/Pixel	SNR (dB)	Complexity (MFLOPS)
1D nonadaptive	5	24.02	1.97
2D nonadaptive	4	23.87	7.86
1D gain-controlled ADPCM	3	26.80	9.83
2D gain-controlled ADPCM	3	29.52	15.73
1D classified ADPCM	5	20.13	51.12

differential pixel on the basis of a pixel classification procedure. For the gain-controlled ADPCM, the factor that normalizes the quantizer input is computed recursively using a simple procedure (see equation 6.50). In the case of the classified ADPCM, the classifier decides the class of the current pixel on the basis of the variance of a neighborhood of pixels with the current pixel at its center. Two examples are used to illustrate the principle of the ADPCMs. It is found that the gain-controlled ADPCM performs the best yielding an SNR of 26.80 dB using 1D prediction and 29.52 dB using 2D prediction, both at 3 bpp. However, the classified ADPCM yields only 20.13 dB at about 5 bpp with good visual quality. Its performance could be improved if a better classifier metric such as the local neighborhood contrast is used and/or 2D predictor with nonuniform quantizers is used. The performance and complexity of the different versions of the DPCM discussed thus far are compared and listed in Table 6.2. The complexity is in terms of *millions of floating point operations per second* (MFLOPS). Only floating point multiplications are considered because floating point addition/subtraction is relatively much faster. The best DPCM is found to be the gain-controlled ADPCM, which yields the highest SNR at the lowest bit rate of 3 bpp with a complexity of around 10–16 MFLOPS.

The next two chapters deal with image coding in the transform domain. We will discuss how transform coding can achieve fractional bit rates, which the DPCM cannot achieve. Transform coders are very important because they are used in major image and video compression standards. These coders also incorporate the human visual perception model to achieve a high compression with visually lossless quality.

REFERENCES

1. N. L. Gerr and S. Cambanis, "Analysis of adaptive differential PCM of a stationary Gauss–Markov input," *IEEE Trans. Inf. Theory*, IT-33, 350–359, 1987.
2. J. B. O'Neal, Jr., "Differential pulse-code modulation (DPCM) with entropy coding," *IEEE Trans. Inf. Theory*, IT-21 (2), 169–174, 1976.
3. V. R. Algazi and J. T. DeWitte, "Theoretical performance of entropy coded DPCM," *IEEE Trans. Commun.*, COM-30 (5), 1088–1095, 1982.
4. A. N. Netravali, "On quantizers for DPCM coding of picture signals," *IEEE Trans. Inf. Theory*, IT-23, 360–270, 1977.

5. T. Kailath, "An innovations approach to least-squares estimation: Part 1. Linear filtering in additive white noise," *IEEE Trans. Autom. Control*, AC-1333, 646–655, 1968.
6. N. S. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice Hall, Englewood Cliffs, NJ, 1984.
7. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
8. J. W. Modestino and V. Bhaskaran, "Robust two-dimensional tree encoding of images," *IEEE Trans. Commun.*, COM-29 (12), 1786–1798, 1981.
9. G. Musmann, "Predictive image coding," in *Adv. Electron. Electron Phys.*, Suppl. 12, 73–112, Academic Press, New York, 1979.
10. K. S. Thyagarajan and H. Sanchez, "Image sequence coding using VDPCM and motion compensation," *IEEE ICASSP*, 1989.
11. K. S. Thyagarajan and H. Sanchez, "Encoding of videoconference signals using VDPCM," *Signal Process., Image Commun.*, 2(1), 81–94, 1990.
12. A. K. Jain and S. H. Wang, "Stochastic image models and hybrid coding," Final Report, NOSC contract N00953–77-C-003MJE, Dept. of EE, SUNY Buffalo, New York, 1977.
13. L. H. Zetterberg, S. Ericsson, and C. Couturier, "DPCM picture coding with two-dimensional control of adaptive quantization," *IEEE Trans. Commun.*, COM-32 (4), 457–462, 1984.
14. W. Zschunke, "DPCM picture coding with adaptive prediction," *IEEE Trans. Commun.*, 1295–1301, 1977.
15. G. Bostlemann, "A high-quality DPCM system for video telephone signals using 8 Mb/s," *Nach. tech. Z.*, 115–117, 1974.
16. H. Yamamoto, Y. Hatori, and H. Murakami, "30 Mb/s codec for the NTSC color TV signal using an interfiled-intrafield adaptive predictor," *IEEE Trans. Commun.*, COM-29, 1859–1867, 1981.

PROBLEMS

6.1. A suboptimal 1D linear predictor is described by $\hat{x}[n] = x[n-1]$. Show that the resulting variance of the prediction error $e[n] = x[n] - \hat{x}[n]$ equals $2\sigma_x^2(1-\rho_1)$, where σ_x^2 is the variance of the input signal and $\rho_1 = R_x(1)/R_x(0)$ is the correlation coefficient.

6.2. If the input sequence $\{x[n]\}$ has a mean value of μ, then show that the optimal first-order linear predictor is described by $\hat{x}[n] = \rho_1 x[n-1] + (1-\rho_1)\mu$.

6.3. If the first-order linear predictor coefficient is 1, then show that the resulting SNR is less than only 1 dB if $\rho_1 \succ 0.6$.

6.4. Let the first-order predictor be described by $\hat{x}[n] = \alpha x[n-2]$. Determine the value of α that minimizes the MSE between the actual and predicted samples. Assume that the correlation coefficient varies as $\rho_k = \rho^{|k|}$. If $\rho = 0.95$, show that the prediction gain of this predictor is lower by about 3 dB than that of the first-order predictor which uses the previous sample, that is, $\hat{x}[n] = \alpha x[n-1]$.

6.5. If the predictor is described by a simple differencing, that is, $\hat{x}[n] = x[n-1]$, then show that the first correlation coefficient

$$\rho_e(1) = \frac{E\{e[n]\,e[n-1]\}}{E\left\{e^2[n]\right\}}$$

of the prediction error is given by $\rho_e(1) = 2\rho_1 - \rho_2 - 1/[2(1-\rho_1)]$.

6.6. Consider an optimal first-order linear predictor for which $\hat{x}[n] = \rho_1 x[n-1]$. Show that the first correlation coefficient of the prediction error signal of this optimal predictor is $\rho_e(1) = \rho_1(\rho_1^2 - \rho_2)/(1-\rho_1^2)$. If $\rho_1 = 0.75$ and $\rho_2 = 0.5$, what is the value of $\rho_e(1)$? Explain the implication of a small value for $\rho_e(1)$.

6.7. Design a classifier ADPCM using a 2D causal linear predictor and apply it to encode a real image.

6.8. Implement a classifier ADPCM using the local contrast as the adaptation rule. Local contrast may be defined as the ratio of the standard deviation to the mean of the pixels in a local neighborhood. Test the DPCM using a real intensity image. Choose a neighborhood size of 9×9 pixels.

6.9. Develop a method to implement a gain-controlled 1D ADPCM using optimal predictors on subimages rather than a single optimal predictor for the whole image. Do you expect any improvement in performance over a single predictor gain-controlled ADPCM?

7 IMAGE COMPRESSION IN THE TRANSFORM DOMAIN

7.1 INTRODUCTION

Predictive coding is based on quantizing each individual differential pixel followed by entropy coding. Because predictive coding is based on coding each pixel at a time, it is not possible to achieve fractional bit rates. On the other hand, transform coding is based on quantizing a block of transform coefficients followed by entropy coding of the quantized coefficients [1–4]. As was described earlier, the idea behind applying a unitary transform to a block of pixels is to decorrelate the pixels and compact the energy in the pixels into as few coefficients as possible with the result that only a few *significant* coefficients need to be coded, hence the compression. Unlike predictive coding, a different number of bits of quantization in an *optimal fashion* can be used for the transform coefficients to realize as low a bit rate as possible. Therefore, it is feasible to achieve fractional bit rates on an average in the transform domain. That this is so will be proven in what follows.

7.2 BASIC IDEA BEHIND TRANSFORM CODING

The basic principle behind transform coding can be described as follows. On the encoding side, an image to be compressed is first divided into $N \times N$ nonoverlapping blocks in the usual raster scanning order, that is, from left-to-right, top-to-bottom. Then, a particular unitary transform of size $N \times N$ is applied to each pixel block. Next, the transform coefficients in each block are quantized individually to different

Still Image and Video Compression with MATLAB By K. S. Thyagarajan.
Copyright © 2011 John Wiley & Sons, Inc.

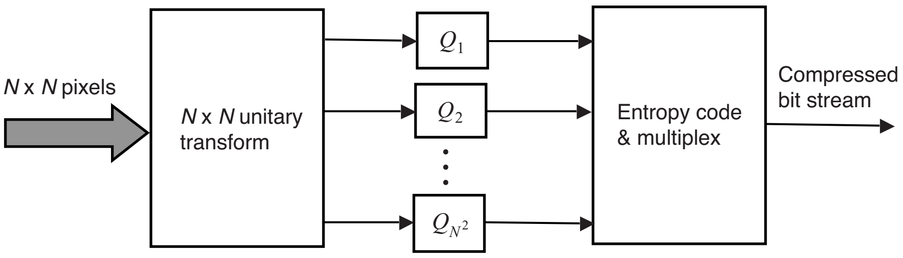


Figure 7.1 Block diagram of a transform coder.

number of bits depending on the *significance* of the coefficients. The block of quantized coefficients is then scanned to serialize for transmission or storage. As there may be a large run of zeros in each quantized coefficient block, *run-length coding* (RLC) of different run-lengths are found and are entropy coded for transmission or storage. On the decoding side, the operations are reversed. First, the compressed bits are entropy decoded to separate the individual transform coefficients. Next, the coefficients are dequantized, arranged in a rectangular block followed by inverse unitary transform. Finally, the decompressed blocks are arranged to form the image. Figure 7.1 is a block diagram description of a transform coder.

From the forgoing brief description of a transform-domain coder, we observe the following features. First, a suitable transform and its size must be determined. Second, the optimal number of bits of quantization to use for the individual transform coefficients must be found. Third, for the quantizer bits determined from the bit allocation rule, one has to design the optimal quantizers. Finally, a suitable entropy coder must be chosen and the codewords must be generated. In the following, we will discuss the individual items.

7.2.1 Choice of a Unitary Transform

The best transform one can use is the Karhunen–Loève transform (KLT) [5–7], which completely decorrelates the pixels in the transform domain. Therefore, the autocorrelation matrix of the KL-transformed image is a diagonal matrix. However, the KLT matrix is image dependent and so is not quite suitable for real-time compression nor is suitable for standardization. The discrete cosine transform (DCT) has been shown to be very nearly equal to the KLT in its energy compaction capability [8] and is also independent of an image in question, that is, its kernel matrix is fixed for a given size. Therefore, DCT has been the choice of compression vehicle in image and video compression standards such as the Joint Photographic Experts Group (JPEG) and Moving Picture Experts Group (MPEG). Hence, our discussion will be limited to the DCT-based compression in this chapter.

Typical transform size is 8×8 pixels. A larger size transform say 16×16 is not as efficient as say 8×8 in its energy compaction ability. Table 7.1 shows the percentage of DCT coefficients containing greater than 99% of total energy for transform sizes of 4×4, 8×8, and 16×16 for the cameraman image. One might anticipate a

Table 7.1 Percent coefficients containing greater than 99% total energy

DCT Size	% Coefficients Containing $> 99\%$ of Total Energy
4×4	62.5
8×8	65.63
16×16	68.75

much smaller percentage of coefficients for 16×16 DCT containing 99% or greater total energy, but we see that 68.75% of coefficients are needed to compact the total energy. Thus, 16×16 DCT is not that much efficient in its energy compaction ability. Moreover, the computational complexity increases with the size of the transform. DCT is a fast algorithm meaning that an N-point DCT can be computed in $\frac{N}{2}\log_2 N$ operations; an operation here refers to multiplication. Thus, an 8×8 block DCT implemented in a row–column fashion requires 144 operations, while a 16×16 DCT needs 1024 operations, almost an order of magnitude increase in computations from that for 8×8 DCT. Based on energy compaction ability and computational efficiency, we will limit our discussion to 8×8 DCT-based coding methods.

7.2.2 Optimal Bit Allocation

The numerical values of the transform coefficients are a function of the activities of the image in that block. Flat pixel blocks have very few AC coefficients with significant values, while very active blocks have a large number of AC coefficients with significant values. Therefore, it is meaningful to design the quantizers with different number of bits of quantization. The average bit rate in bits per pixel (bpp) is usually fixed for a given encoder. Therefore, this opens up the question of how many bits to assign to each quantizer for the transform coefficients to meet the bit budget. This question is answered by the optimal bit allocation rule or algorithm [9–12], as described below.

Problem Statement

Given the unitary transform matrix $\mathbf{A}$ of size $N \times N$ and an overall average bit rate of R bpp, determine the individual quantizer bits R_i, $1 \le i \le N^2$ such that the mean square error (MSE) between the input and reconstructed images is a minimum. The average bit rate of the coder is related to the individual quantizer bits by $R = \frac{1}{M}\sum_{i=1}^{M} R_i$, $M = N^2$.

We know that a unitary transform conserves energy. Therefore, it implies that the energy in a block of pixels equals the sum of the energies in the transform coefficients. If the transform coefficients are quantized, then the MSE due to reconstruction equals the sum of the errors due to quantizing the coefficients. That is, each transform coefficient introduces an MSE equal to the MSE due to quantization of that coefficient. Hence, the overall MSE D between the input and reconstructed images is given by

$$D = \frac{1}{M}\sum_{i=1}^{M}\sigma_{q_i}^2 \tag{7.1}$$

where the variance of the quantization error of the ith quantizer is related to its number of bits, as given by

$$\sigma_{q_i}^2 = f_i\sigma_i^2 2^{-2R_i} \tag{7.2}$$

In equation (7.2), f_i is the ith quantizer constant specific to the probability density function (PDF) of its input and σ_i^2 is the variance of the ith quantizer input, which is a transform coefficient. Since the minimization of the MSE is subject to the average bit rate constraint, we use the Lagrange multiplier method to find the minimum MSE. More specifically, we want

$$\frac{\partial}{\partial R_k}\left\{\frac{1}{M}\sum_{i=1}^{M} f_i\sigma_i^2 2^{-2R_i} - \lambda\left(R - \frac{1}{M}\sum_{i=1}^{M}R_i\right)\right\} = 0 \tag{7.3}$$

where the Lagrange multiplier λ is a constant. Using the fact that $\partial\left(f_k\sigma_k^2 2^{-2R_k}\right)/\partial R_k = -(2\ln 2)\, f_k 2^{-2R_k}$, equation (7.3), after some algebraic manipulation, results in

$$R_k = \lambda' + \log_2\sigma_k \tag{7.4}$$

In equation (7.4),

$$\lambda' = \tfrac{1}{2}\log_2(2f_k\ln 2) - \tfrac{1}{2}\log_2\lambda \tag{7.5}$$

If we sum the terms in equation (7.4) over M, we get

$$\sum_{k=1}^{M}R_k = M\lambda' + \sum_{k=1}^{M}\log_2\sigma_k \tag{7.6}$$

Therefore, from equation (7.6) we find that

$$\lambda' = \frac{1}{M}\sum_{k=1}^{M}R_k - \frac{1}{M}\sum_{k=1}^{M}\log_2\sigma_k = R - \frac{1}{M}\sum_{k=1}^{M}\log_2\sigma_k \tag{7.7}$$

Table 7.2 Variances of 8 × 8 DCT coefficients of cameraman image

202000	13310	4530	1820	920	430	340	240
6830	2660	1410	970	740	560	280	210
2170	1090	660	510	380	240	210	140
1110	410	310	260	200	150	120	100
680	220	160	200	150	110	70	50
330	130	90	90	100	90	50	30
230	90	70	60	50	50	40	30
120	70	50	40	30	30	30	30

Using equation (7.7) in (7.4), we determine the optimal bits for the kth quantizer as

$$R_k = R + \log_2 \left\{ \frac{\sigma_k}{\left(\prod_{i=1}^{M} \sigma_i\right)^{1/M}} \right\} = R + \frac{1}{2}\log_2 \left\{ \frac{\sigma_k^2}{\left(\prod_{i=1}^{M} \sigma_i^2\right)^{1/M}} \right\} \tag{7.8}$$

Thus, the number of bits for the kth quantizer is equal to the average bit rate plus a quantity that depends on the ratio of its input variance to the geometric mean of the variances of all the coefficients. In general, the quantizer bits are not integers nor are they all positive. We have to round them to the nearest integer greater than or equal to the quantity given in equation (7.8). This will naturally result in an average rate equal to or greater than the bit budget R. The following example will elucidate the optimal bit allocation rule for a real image.

Example 7.1 Let us read a real intensity image and calculate the quantizer bits in an optimal fashion in the minimum MSE sense for the DCT transform matrix of size 8×8 and a bit budget of 1 bpp.

Let us choose the cameraman image for this example. The variances of the 8×8 DCT coefficients are listed in Table 7.2. The geometric mean of the coefficient variances is found to be 235.157. Then the quantizer optimal bit allocation looks like the entries in Table 7.3. The average actual bit rate works out to be 1.05 bpp, which is slightly greater than the bit budget. As pointed out, this slight increase is because the actual optimal bits are not integers and the arithmetic rounding causes the actual bit rate to be greater than the bit budget.

A point of observation is that this bit allocation used in Example 7.1 may not be optimal for another image because it is based on a single image. Therefore, one can use several images of similar contents and dynamic range and calculate the DCT coefficient variances, or can use a suitable analytical model for the coefficient variances and use it to calculate the quantizer bits. For a stationary random field, the $N \times N$ covariance matrix $\mathbf{R_Y}$ of the coefficients of the unitary transform of N-vectors

Table 7.3 8 × 8 DCT quantizer optimal bits for cameraman image

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

is expressed in terms of the covariance $\mathbf{R_X}$ of the input vectors as

$$\mathbf{R_Y} = \mathbf{A R_X A^{*T}} \tag{7.9}$$

The coefficient variances of the unitary transform matrix are then the diagonal elements of $\mathbf{R_Y}$:

$$\sigma_Y^2(j) = [\mathbf{R_Y}]_{j,j} \tag{7.10}$$

For an image, if the covariance is separable, its covariance can be expressed as the outer product of the covariance matrices along each dimension. Then the variances of the coefficients of the two-dimensional (2D) unitary transform can be expressed as

$$\sigma_Y^2(k,l) = \sigma_u^2(k)\,\sigma_v^2(l) = \left[\mathbf{A R_1 A^{*T}}\right]_{k,k}\left[\mathbf{A R_2 A^{*T}}\right]_{l,l} \tag{7.11}$$

A separable covariance function of a random field [13] is of the form

$$r(m,n) = r_1(m)\,r_2(n) = \sigma_x^2 \rho_1^{|m|} \rho_2^{|n|} \tag{7.12}$$

A more realistic, nonseparable, circularly symmetric covariance function of images may be written as

$$r(m,n) = \sigma_x^2 \exp\left(-\sqrt{a\,m^2 + b\,n^2}\right) \tag{7.13}$$

In equation (7.13), $a = -\ln\rho_1$, $b = -\ln\rho_2$, ρ_1 and ρ_2 are the first correlation coefficients along the respective spatial dimensions. Using equation (7.12) or (7.13) in (7.9), we can determine the variances of the coefficients of the DCT and then calculate the optimal quantizer bits. Alternatively, we can calculate $\mathbf{R_1}$ and $\mathbf{R_2}$ from the actual image and use equation (7.11) to determine the coefficient variances.

```
%   Example7_1.m
%   Computes transform quantizer bits optimally using
%   the rule: R(k) = R +0.5*log2(var of kth coeff./geometric variance)

clear
A = imread('cameraman.tif');  % read an image
R = 1.0; % Bit budget in bpp
N = 8; % Transform matrix size
% call function AllocateBits to determine quantizer optimal bits
Qbits = AllocateBits(A,R,N);
%
sprintf('Optimal quantizer bits\n')
disp(Qbits)
sprintf('Desired avg. rate = %3.2f bpp\tActual avg. rate
                           = %3.2f bpp\n',R,sum(Qbits(:))/(N*N))
```

```
function Qbits = AllocateBits(A,R,N)
% Qbits = AllocateBits(A,R)
% Computes the quantizer bits optimally using the rule:
% R(k) = R +0.5*log2(var of kth coeff./geometric variance)
% where R in bpp is the bit budget.

[Height,Width,Depth] = size(A); % Height, Width & Depth of the image
if Depth > 1
    A1 = double(A(:,:,1));
else
    A1 = double(A);
end
% N is the size of the unitary transform matrix
CoefSigma = zeros(N,N); % Std. dev. of the transform coefficients
Qbits = zeros(N,N); % Array containing the quantizer bits
fun = @dct2;
T = blkproc(A1,[N N],fun);   % do N x N 2D DCT
%   compute the DCT coefficient variance
for k = 1:N
    for l = 1:N
        CoefSigma(k,l) = std2(T(k:N:Height,l:N:Width));
    end
end
%
P1 = CoefSigma .^ (1/(N^2));
GeoMeanOfSigma = prod(P1(:)); % geometric mean of the coef. variances
for k = 1:N
    for l = 1:N
        b = R + log2(CoefSigma(k,l)/GeoMeanOfSigma);
        if b <= 0
            Qbits(k,l) = 0;
        else
            Qbits(k,l) = round(b);
        end
    end
```

Table 7.4 Optimal bit assignment for cameraman image via equation (7.11)

7	5	4	4	3	3	2	2
5	3	2	2	1	1	1	0
4	2	2	1	1	0	0	0
4	2	1	1	0	0	0	0
4	2	1	0	0	0	0	0
3	1	1	0	0	0	0	0
3	1	0	0	0	0	0	0
3	1	0	0	0	0	0	0

For the same cameraman image with a bit budget of 1 bpp, use of equation (7.11) results in the optimal bit assignment as shown in Table 7.4 and an actual average rate of 1.3 bpp.

Integer Bit Assignment The quantizer bit assignment rules described above do not guarantee either positive or integer value for the number of bits. Therefore, one can use an iterative procedure for the assignment of positive integer number of bits of quantization, as follows.

Procedure for Integer Bit Allocation Given an $N \times N$ unitary transform, variances of the transform coefficients, and an average bit budget of R bpp, where $R_t = MR$, $M = N^2$ is the total number of bits for each of the $N \times N$ block of quantizers,

1. Set step j to zero and all quantizer bits to zero, $R_i^{(j)} = 0; 1 \leq i \leq M$.
2. Sort the coefficient variances and denote the maximum variance by σ_m^2.
3. Set $R_m^{(j)} \leftarrow R_m^{(j-1)} + 1$ and $\sigma_m^2 \leftarrow \sigma_m^2/2$.
4. Set $R_t \leftarrow R_t - 1$. If $R_t = 0$, stop. Otherwise, set $j \leftarrow j + 1$ and go to step 2.

Using the integer bit assignment procedure [14], the optimal quantizer bits for the cameraman image for an average bit budget of 1 bpp are shown in Table 7.5. The

Table 7.5 Quantizer bits for cameraman image using integer bit assignment

10	6	5	3	2	1	1	0
5	4	3	2	2	2	1	0
3	2	2	1	1	0	0	0
3	1	1	0	0	0	0	0
2	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

actual bit rate comes out exactly as 1 bpp. The MATLAB code for integer bit assignment procedure is listed below. The function *AssignIntgrBits* has three arguments: an intensity image, the average bit rate in bpp, and the size of the unitary transform (assumed square). It returns the quantizer bits in the array *Qbits* of the same size as the transform.

```
function Qbits = AssignIntgrBits(x,R,N)
%   Assigns quantizer bits optimally using
%   recursive integer bit allocation rule.


M = N*N; % total number of quantizers
Rtotal = M*R;   % total bits for the N x N block
[Rows,Cols] = size(x); % Rows & Columns of input image
fun = @dct2;
T = blkproc(x,[N N],fun);   % do N x N 2D DCT
%   compute the DCT coefficient variances
VarOfCoef = zeros(N,N);
for k = 1:N
    for l = 1:N
        V = std2(T(k:N:Rows,l:N:Cols));
        VarOfCoef(k,l) = V * V;
    end
end
%   assign integer bits recursively
Qbits = zeros(N,N);
Var = VarOfCoef;
while Rtotal > 0
    Max = -9999;
    for k = 1:N
        for l = 1:N
            if Max < Var(k,l)
                Max = Var(k,l);
                Indx = [k l];
            end
        end
    end
    Qbits(Indx(1),Indx(2)) = Qbits(Indx(1),Indx(2)) + 1;
    Var(Indx(1),Indx(2)) = Var(Indx(1),Indx(2))/2;
    Rtotal = Rtotal - 1;
end
%
sprintf('Optimal quantizer bits\n')
disp(Qbits)
sprintf('Desired avg. rate = %3.2f bpp\tActual avg. rate
                           = %3.2f bpp\n',R,sum(Qbits(:))/(N*N))
```

7.2.3 Design of the Quantizers

The next step in the transform coder is to design the optimal quantizers with individual bits obtained from the optimal bit allocation rule. The optimal quantizer is the

Table 7.6 Quantization step sizes of uniform quantizers using optimal bit assignment

12	18	29	35	30	46	32	22
32	22	23	22	28	24	53	38
20	23	23	17	39	29	27	26
32	27	32	26	35	25	21	31
34	19	27	25	27	47	22	34
19	14	37	23	34	47	35	19
38	30	33	27	18	31	18	18
18	36	26	25	14	19	16	23

Lloyd–Max quantizer, which is optimized to its input PDF and which has nonuniform quantization steps. Since the decision intervals and output levels for given input PDF and quantizer bits are available in tabular form in many textbooks for the Lloyd–Max quantizers, one can use them in the transform coder, or can design such quantizers for a given image to be compressed. A third option is to use simply uniform quantizers with respective number of bits obtained from the bit allocation rule. The uniform quantizer being simple is the preferred quantizer in the popular compression standards, such as JPEG and MPEG. At reasonably high bit rates, the performance of a uniform quantizer is nearly the same as that of a nonuniform quantizer. It is also efficient to implement both in software and hardware. Moreover, one has the ability to control the amount of compression achieved by simply scaling the quantization steps without altering the quantization steps themselves. This may not be feasible with nonuniform quantizers because scaling the quantization steps and corresponding output levels may not necessarily be optimal.

In continuation of Example 7.1, the cameraman image is quantized and dequantized using the optimal bits of Table 7.3 at an average bit budget of 1 bpp. The quantization step sizes for the uniform quantizers are shown in Table 7.6. The signal-to-noise ration (SNR) for the optimal case is found to be 17.24 dB. The reconstructed image is shown in Figure 7.2a. The reconstructed images using the optimal bit assignment via equation (7.11) and integer bit assignment rule are shown, respectively, in Figures 7.2b,c for comparison. The quantization matrix for the optimal case is designed purely on the basis of minimizing the MSE. Incorporation of the human visual perception model is a key factor to achieving visually lossless compression at moderate bit rates.

7.2.4 Entropy Coding of the Transform Coefficients

The last part in the design of the transform coder is the entropy coder. As was pointed out earlier, entropy coding is necessary to code the numerical values of the quantized transform coefficients for transmission or storage. That is, the entropy coder converts the quantized transform coefficients or some other feature of the coefficients into symbols for storage or transmission. Since these values have certain probabilities of occurrence, entropy coding, which is based on symbol probabilities, also

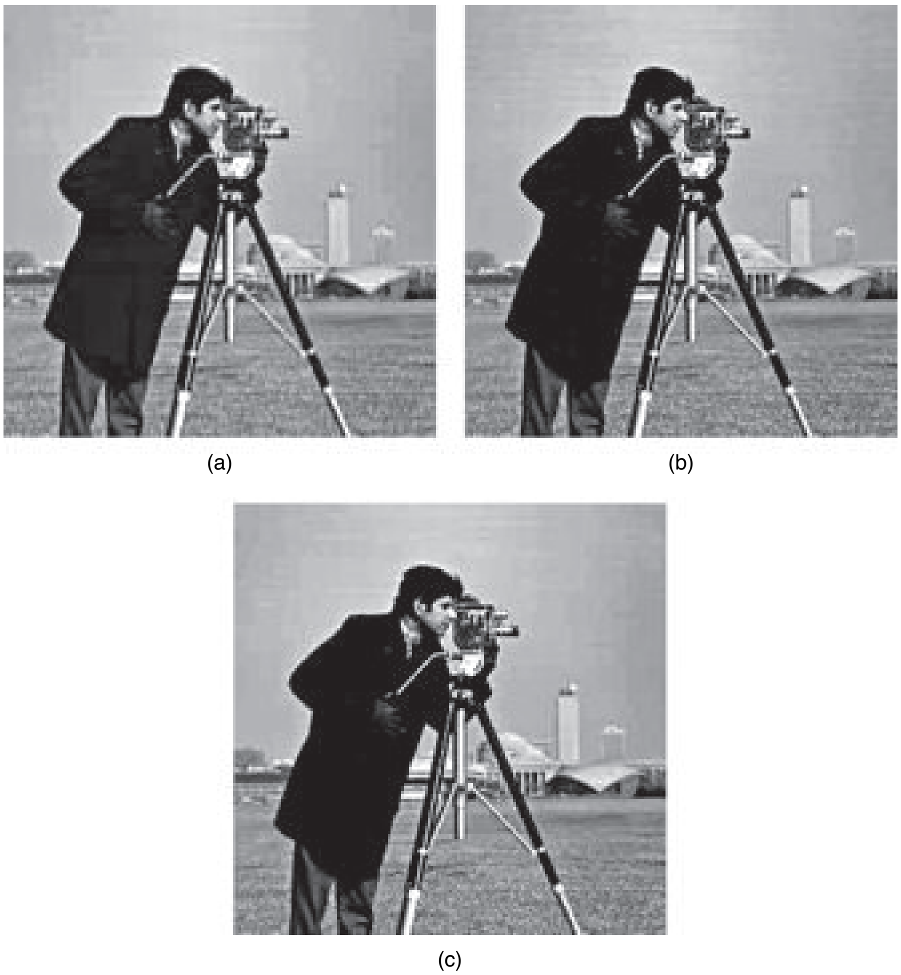


Figure 7.2 Quantized/dequantized cameraman image using the quantization matrix of (a) optimal bit assignment with a bit budget of 1 bpp, (b) optimal bit assignment via (7.11), and (c) optimal integer bit assignment rule.

serves the purpose of yielding additional compression. We have already described several entropy coding methods in detail in Chapter 5. One can use any one of those methods to generate the codewords for the various DCT coefficients (symbols) and use them for compression [15, 16]. But, in the context of transform coding, there are some characteristics that are specific to transform coding, which can be exploited in entropy coding. We will describe these characteristics in this section.

Run-Length Coding Many DCT coefficients become zero after quantization. Instead of coding each zero coefficient separately, it is more efficient to code the

Table 7.7 Ordering coefficients using zigzag scanning in Figure 5.8

1	2	6	7	15	16	28	29
3	5	8	14	17	27	30	43
4	9	13	18	26	31	42	44
10	12	19	25	32	41	45	54
11	20	24	33	40	46	53	55
21	23	34	39	47	52	56	61
22	35	38	48	51	57	60	62
36	37	49	50	58	59	63	64

Table 7.8 Quantized 8 × 8 DCT coefficients

55	−43	7	6	−3	4	0	−4
0	1	1	0	0	1	0	0
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0
0	1	1	1	1	0	0	1
0	0	0	0	1	1	1	0

zero runs. Further it is much more efficient to code the run-length and coefficient amplitude as a pair since there are only a finite number of combinations of run-length/amplitude pairs in a quantized $N \times N$ DCT block. Here, the amplitude refers to the value of that DCT coefficient which breaks the zero run. Generally, the high-frequency coefficients are zero after quantization. In order to get large run-lengths, one can use a zigzag scanning pattern instead of raster scanning. A zigzag scan used in JPEG and MPEG was shown in Figure 5.8, and the coefficient ordering is shown in Table 7.7 for visualization. It is more convenient to explain the process of generating the run-length/amplitude pairs by an example. Consider the quantized 8×8 DCT coefficients shown in Table 7.8, where the first entry is the DC value and the rest is AC. Table 7.9 shows the DCT coefficients after zigzag scanning. We then find the run-length/amplitude pairs as shown in Table 7.10.

Table 7.9 Zigzag scanned 8 × 8 DCT coefficients of Table 7.8

55	−43	0	0	1	7	6	1	0	0	0	0	0	0	−3	4
0	1	0	1	1	0	1	1	1	1	1	0	−4	0	1	1
0	1	1	0	0	1	0	0	1	1	0	1	1	0	0	1
0	0	1	0	0	1	0	0	0	1	1	0	0	1	1	0

Table 7.10 Run-length/amplitude pairs of entries in Table 7.9

RL/Amp	RL/Amp	RL/Amp	RL/Amp
0/−43	0/1	1/1	2/1
2/1	1/1	0/1	3/1
0/7	0/1	2/1	0/1
0/6	0/1	2/1	2/1
0/1	0/1	0/1	0/1
6/−3	0/1	1/1	EOB
0/4	1/−4	0/1	
1/1	1/1	2/1	
1/1	0/1	2/1	

7.3 CODING GAIN OF A TRANSFORM CODER

The MSE due to quantizing the *i*th transform coefficient is given by equation (7.2), and we repeat it here for easy reference:

$$\sigma_{q_i}^2 = f_i \sigma_i^2 2^{-2R_i} \tag{7.14}$$

In equation (7.14), σ_i^2 is the variance of the input coefficient to the *i*th quantizer, R_i is the number of bits of quantization, and f_i is the quantizer constant. From the optimal bit allocation rule, the number of bits for the *i*th coefficient quantizer is given by equation (7.8). Therefore, the total average distortion for all the M quantizers can be found by summing equation (7.14) and is expressed as

$$D = \frac{1}{M} \sum_{i=1}^{M} f_i \sigma_i^2 2^{-2\left(R + \frac{1}{2}\log_2(\sigma_i^2)/\left(\prod_{k=1}^{M}\sigma_k^2\right)^{1/M}\right)} \tag{7.15}$$

Taking the quantizer constants to be equal and noting that

$$2^{-\log_2(\sigma_k^2)/\left(\prod_{j=1}^{M}\sigma_j^2\right)^{1/M}} = \frac{\left(\prod_{j=1}^{M}\sigma_j^2\right)^{1/M}}{\sigma_k^2} \tag{7.16}$$

we find that the total average distortion for the M quantizers reduces to

$$D = f 2^{-2R} \left(\prod_{i=1}^{M} \sigma_i^2\right)^{1/M} \tag{7.17}$$

where R is the bit budget for the transform coder. For M pulse code modulation (PCM) quantizers with each having R bits of quantization, the overall MSE due to quantization is

$$D_{PCM} = \frac{f2^{-2R}}{M}\sum_{i=1}^{M}\sigma_i^2 \tag{7.18}$$

The *coding gain* of the transform coder over the PCM coder for the same bit rate R is defined as the ratio of the average distortion of the PCM coder to that of the transform coder [11] and is found using equations (7.17) and (7.18) as

$$G_{\mathrm{TC}} = \frac{D_{\mathrm{PCM}}}{D_{\mathrm{TC}}} = \frac{\frac{1}{M}\sum_{i=1}^{M}\sigma_i^2}{\left(\prod_{j=1}^{M}\sigma_j^2\right)^{1/M}} \tag{7.19}$$

The numerator on the right-hand side of equation (7.19) is the arithmetic mean of the variances of the coefficients and the denominator is the geometric mean of the variances. Thus, the gain of the transform coder over the PCM equals the ratio of the arithmetic mean of the variances to the geometric mean of the variances.

Example 7.2 We shall compute the coding gain of the 2D DCT of sizes 4×4, 8×8, 16×16, and 32×32 pixels for a set of intensity images. The collage of the images used is shown in Figure 7.3. Some of the images are actually RGB images, and we convert them to luminance images and then compute the coding gain. Table 7.11 lists the coding gain for the various transform sizes, and Figure 7.4 is a plot of coding gain versus DCT size. From the table, we notice that the difference in coding gain between 32×32 and 8×8 blocks is only about 1.5 dB with the exception of the last two images. We also observe that the gain is higher for images with smaller details than that for images with larger details. The MATLAB code for calculating the coding gain is listed below.

```
% Example7_2.m
% Calculate the coding gain of a DCT coder
% for transform sizes of 4 x 4, 8 x 8, 16 x 16 and
% 32 x 32 pixels.
% The function "TCgain" calculates the coding gain in dB

clear
fileName = 'airplane.ras';
A = imread(fileName);
[Height,Width,Depth] = size(A);
%
if Depth == 1
    A1 = double(A);
```

```
else
    A = double(rgb2ycbcr(A));
    A1 = A(:,:,1);
end
figure,imshow(A1,[]), title(fileName)
N = 4;
for j = 1:4
    G = TCgain(A1,N);
    sprintf('Coding Gain for %dx%d DCT = %4.2f\n',N,N,G)
    N = 2*N;
end


function G = TCgain(x,N)
% G = TCgain(x)
%   computes the coding gain of the 2D DCT coder
% Input:
%   x = intensity image
%   N = size (assumed square) of the transform block
% Output:
%   G = coding gain over PCM in dB


[Height,Width] = size(x);
fun = @dct2;
Y = blkproc(x,[N N],fun);   % do N x N 2D DCT
%   compute the DCT coefficient variance
CoefVar = zeros(N,N);
for i = 1:N
    for j = 1:N
        t = std2(Y(i:N:Height,j:N:Width));
        CoefVar(i,j) = t * t;
    end
end

%
MeanVar = mean(CoefVar(:));
P1 = CoefVar .^ (1/(N^2));
GeoMeanVar = prod(P1(:)); % geometric mean of the coef. variances
G = 10*log10(MeanVar/GeoMeanVar); % coding gain
```

7.4 JPEG COMPRESSION

JPEG is an acronym for *Joint Photographic Experts Group* and defines a set of still picture grayscale and color image compression algorithms and data transport format [17–20]. There are two compression algorithms in the standard, one is based on 2D DCT and the other is based on spatial prediction methods. The DCT-based algorithm is intended for compression quality ranging from *very good* to *visually indistinguishable*. For lossless and nearly lossless coding, predictive coding based on DPCM is used.

Figure 7.3 Collage of images used in Example 7.2 for calculating the coding gain of the DCT coder.

Table 7.11 Coding gain of transform coder for various images

	Coding Gain (dB)			
Image	4×4	8×8	16×16	32×32
Cameraman	10.97	12.18	12.82	13.35
Autumn	17.03	18.87	19.68	20.49
Peppers	18.42	20.50	21.45	21.96
Aerial	7.93	9.00	9.52	9.81
Airplane	17.55	19.35	20.15	20.58
Birds	16.10	18.02	18.94	19.46
Lighthouse	9.73	10.98	11.60	11.92
Masuda	23.14	26.06	27.45	28.18
Yacht	16.45	19.40	20.94	21.74

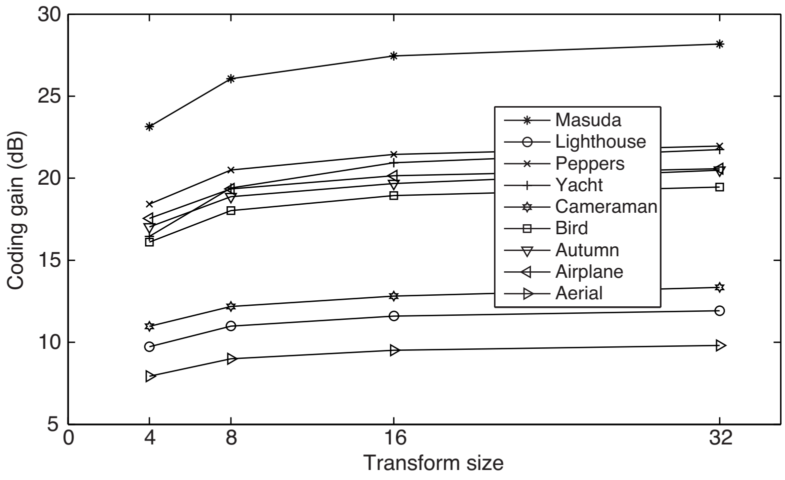


Figure 7.4 A plot of transform coding gain versus DCT block size using the cameraman image.

JPEG defines four options or modes of operation, namely, sequential, progressive, hierarchical, and lossless coding. As the name implies, in the sequential mode, image blocks are fully coded sequentially block by block in a raster scan order. Similarly, the blocks are fully decoded sequentially in the same raster scan order. In the progressive mode, the entire image is first coded at a quality level that is lower than the final quality intended. In the second time around, the quality is improved by providing additional information in an incremental fashion. Similarly at each step incremental information is coded until the final intended quality level is reached. After decoding the initial image, one obtains a lower quality image. By decoding each additional bit stream the quality is improved progressively. In the hierarchical mode, a pyramidal coding scheme is used. Instead of coding the image at full resolution, first a lower resolution image is encoded that requires much less number of bits. It is then upsampled and interpolated to the full resolution and the difference between the original full resolution image and the interpolated image is encoded and transmitted or stored. Thus, there are two bit streams, one corresponding to the lower resolution image and the other belonging to the difference image. Finally, the lossless coding uses predictive coding instead of the DCT because DCT involves rounding operations that will introduce some errors.

JPEG allows two entropy coders—Huffman and arithmetic coders [21–23]. The *baseline* system guarantees a reasonable level of functions in all decoders that use the DCT algorithms. It allows a restricted version of the sequential DCT algorithm for images with 8 bits per sample and Huffman coding. For color images, YCbCr components are used with 4:2:0 chroma sampling format [24]. The baseline coding and decoding procedures are shown in block diagrams in Figures 7.5a,b, respectively. The coding procedure consists of (1) level shifting, (2) 2D DCT, (3) quantization,

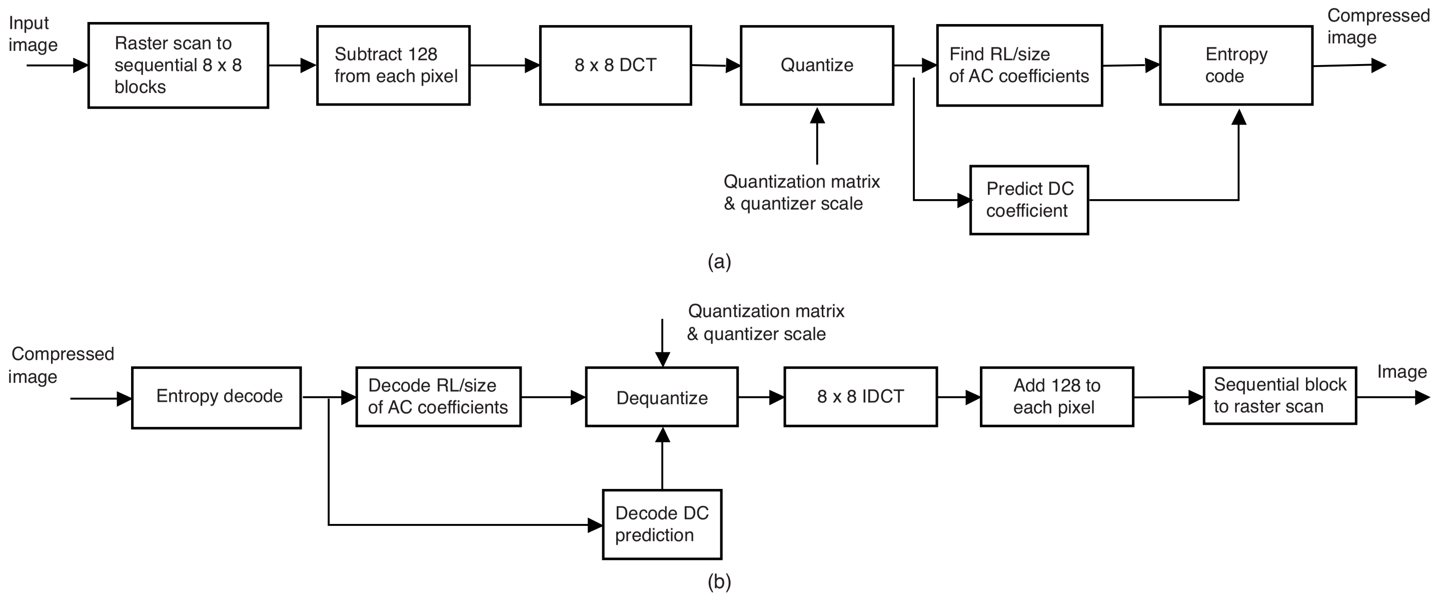

Input image
Raster scan to sequential 8 x 8 blocks
Subtract 128 from each pixel
8 x 8 DCT
Quantize
Quantization matrix & quantizer scale
Find RL/size of AC coefficients
Predict DC coefficient
Entropy code
Compressed image
(a)
Compressed image
Entropy decode
Decode RL/size of AC coefficients
Decode DC prediction
Quantization matrix & quantizer scale
Dequantize
8 x 8 IDCT
Add 128 to each pixel
Sequential block to raster scan
Image
(b)

Figure 7.5 Block diagram of JPEG baseline coder: (a) encoder and (b) decoder.

Table 7.12 Default JPEG luma quantization step sizes of uniform quantizers

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
24	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(4) zigzag scanning the quantized DCT coefficients, and (5) entropy coding. These steps are briefly described as follows.

7.4.1 Level Shifting

The input data is unsigned. Because the standard video interfaces use offset binary representation, the input data is *level shifted* by subtracting 2^{B-1} before applying the DCT, where B in bits is the precision of the input data. After inverse DCT, 2^{B-1} must be added to restore the decompressed data to the unsigned representation.

7.4.2 Quantization

Each 8×8 block of DCT coefficients is quantized using uniform quantizers whose quantization step sizes are determined based on human visual perception model. Table 7.12 is the default JPEG luma quantization matrix. The corresponding chroma (Cb and Cr) quantization matrix is listed in Table 7.13. The compression ratio can be controlled by multiplying the quantization matrices by a *quantizer scale*. The JPEG rule for quantizing the DCT coefficients is described by the following equation.

$$Y_q(k,l) = \left\lfloor \frac{Y(k,l)}{Q(k,l)} + 0.5 \right\rfloor, \ 1 \le k, l \le 8 \tag{7.20}$$

where, $Y(k,l)$ is the unquantized 2D DCT coefficient, $Y_q(k,l)$ is the corresponding uniformly quantized coefficient, $Q(k,l)$ is the quantization step size for the (k, l)th

Table 7.13 Default JPEG chroma quantization step sizes of uniform quantizers

17	18	24	47	66	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

coefficient, and $\lfloor x \rfloor$ is the flooring operation. The dequantization of the quantized DCT coefficients is given by

$$\hat{Y}(k,l) = Y_q(k,l) \times Q(k,l) \tag{7.21}$$

7.4.3 Zigzag Scanning

Due to efficient energy compaction property of the DCT, many coefficients, especially the higher frequency coefficients become zero after quantization. Therefore, zigzag scanning the $N \times N$ DCT array is used to maximize the zero run-lengths. The zigzag scanning pattern used in JPEG is shown in Figure 5.8. After zigzag scanning the DCT coefficients, the DC and AC coefficients are entropy coded.

7.4.4 Entropy Coding

The quantized DCT coefficients are integers and must be entropy coded for transmission or storage. The baseline system uses Huffman coding. Because the DC coefficients of neighboring blocks are correlated, further compression is achieved by coding the DC differences. The quantized AC coefficients are Huffman coded using run-length/amplitude pairs. These are described in the following.

JPEG Coding of DC Coefficients The DC coefficient in each 8×8 block of DCT is a measure of the average value of that block of pixels—actually it is 8 times the average. Since there is considerable correlation in the DC values between adjacent blocks, we can gain additional compression if we code the DC differential values than if we code the DC values individually. This amounts to simple integer prediction of the current block $DC(j)$ from the previous block $DC(j-1)$ as describe by

$$e_{dc}(j) = DC(j) - DC(j-1) \tag{7.22}$$

In JPEG baseline, the DC differential is divided into 16 size categories with each size category containing exactly 2^{size} number entries. From the definition of 2D DCT, we see that the maximum DC value of an 8×8 block of pixels is $8 \times (255\text{–}128) = 1016$ for 8-bit images. Since lossless coding is one mode of JPEG, the maximum value of the DC coefficient difference could be 2032, which is size category 11. For 12-bit images, the DC differential will have a range between −32,767 and +32,767, which is size category 15—the maximum size allowed. Table 7.14 shows the 16 sizes and the corresponding amplitude range for the DC differentials.

To code a DC differential, we first determine its size category from

$$\text{size} = \lceil \log_2(|\text{Val}| + 1) + 0.5 \rceil \tag{7.23}$$

where Val is the DC differential value. The code for the DC differential Val is then the code for its size followed by the binary code for the amplitude. If the differential

Table 7.14 Size category and amplitude range of DC differentials

Size	Amplitude Range
0	0
1	−1,1
2	−3 −2, 2 3
3	−7 −6. . ...−4, 4. . ...6 7
4	−15 −14. . ...−8, 8. . ...14 15
5	−31 −30. . ...−16, 16. . ...30 31
6	−63 −62 . . ...−32, 32 . . ...62 63
7	−127 −126 . . ...−64, 64 . . ...126 127
8	−255 −254 . . ...−128, 128 . . ...254 255
9	−511 −510. . ..−256, 256. . ..510 511
10	−1023 −1022 . . . −512, 512 . . .1022 1023
11	−2047 −2046. . .−1024, 1024. . .2046 2047
12	−4095 −4094. . .−2048, 2048. . .4094 4095
13	−8191 −8190. . .−4096, 4096. . .8190 8191
14	−16383 −16382. . .−8192, 8192. . .16382 16383
15	−32767 −32766. . .−16384, 16384. . .32766 32767

value is negative, then the code for the amplitude is the 2s complement of Val-1. For example, if the DC differential value is +5, then its size category is 3. The code for the size category is **011**. The 3-bit binary code for the amplitude of +5 is **101**. Therefore, concatenating the code for the size with the code for the amplitude will give the code **011101**. If the DC differential is −5, then the size category is still 3. But the 3-bit code for the amplitude is the 2s complement of −6, which is **010**. The code for −5 is, therefore, **011010**.

The default Huffman codes for the luma and chroma DC coefficients are listed in Tables 7.15 and 7.16, respectively.

JPEG Coding of AC Coefficients The quantized AC coefficients are coded slightly differently in the JPEG standard than the DC coefficient because of the possibilities of run-lengths and nonzero amplitudes. The run-length/amplitude or level pair is coded using variable-length codes (VLC). Similar to the DC differential range, Table 7.17 lists the AC amplitude range category and values. Table 7.18 lists the run/size and corresponding Huffman codes for the AC coefficients of luma. A similar listing of run/size and Huffman codes for the chroma is shown in Table 7.19. However, only codes for a subset of the run-length/amplitude pairs are used and any pair not in the allowed range is coded using an *escape* code followed by a 6-bit code for the run-length and an 8-bit code for the amplitude. In many quantized DCT blocks, there may be a large run of zeros up to the end of the block. In such cases, an end-of-block (EOB) code of 4 bits for the luma and 2 bits for the chroma is used to signal the end of the block. Having 2 or 4 bits for the EOB is an indication that the EOB occurs most often.

Table 7.15 JPEG default luma Huffman table for DC coefficient

Category	Code Length	Code
0	3	010
1	3	011
2	3	100
3	2	00
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Table 7.16 JPEG default chroma Huffman table for DC coefficient

Category	Code Length	Code
0	2	00
1	2	01
2	3	100
3	3	101
4	3	110
5	4	1110
6	5	11110
7	6	111110
8	7	1111110
9	9	111111110
10	9	111111110
11	9	111111110

Table 7.17 JPEG size category and amplitude range of AC coefficients

Size	Amplitude Range
1	−1,1
2	−3 −2, 2 3
3	−7 −6. . ...−4, 4. . ...6 7
4	−15 −14. . ...−8, 8. . ...14 15
5	−31 −30. . ...−16, 16. . ...30 31
6	−63 −62 . . ...−32, 32 . . ...62 63
7	−127 −126 . . ...−64, 64 . . ...126 127
8	−255 −254 . . ...−128, 128 . . ...254 255
9	−511 −510. . ..−256, 256. . ..510 511
10	−1023 −1022 . . . −512, 512 . . .1022 1023
11	−2047 −2046. . .−1024,1024. . .2046 2047
12	−4095 −4094. . .−2048,2048. . .4094 4095
13	−8191 −8190. . .−4096,4096. . .8190 8191
14	−16383 −16382. . .−8192,8192. . .16382 16383
15	unused

Table 7.18 JPEG VLCs for luma AC coefficients

RL/Size	Code Length	VLC	RL/Size	Code Length	VLC
0/0	**4 (EOB)**	**1010**	4/1	6	111011
0/1	2	00	4/2	10	1111111000
0/2	2	01	4/3	16	1111111110010111
0/3	3	100	4/4	16	1111111110011000
0/4	4	1011	4/5	16	1111111110011001
0/5	5	11010	4/6	16	1111111110011010
0/6	6	111000	4/7	16	1111111110011011
0/7	7	1111000	4/8	16	1111111110011100
0/8	10	1111110110	4/9	16	1111111110011101
0/9	16	1111111110000010	4/A	16	1111111110011110
0/A	16	1111111110000011	5/1	7	1111010
1/1	4	1100	5/2	10	1111111001
1/2	6	111001	5/3	16	1111111110011111
1/3	7	1111001	5/4	16	1111111110100000
1/4	9	111110110	5/5	16	1111111110100001
1/5	11	11111110110	5/6	16	1111111110100010
1/6	16	1111111110000100	5/7	16	1111111110100011
1/7	16	1111111110000101	5/8	16	1111111110100100
1/8	16	1111111110000110	5/9	16	1111111110100101
1/9	16	1111111110000111	5/A	16	1111111110100110
1/A	16	1111111110001000	6/1	7	1111011
2/1	5	11011	6/2	11	11111111000
2/2	8	11111000	6/3	16	1111111110100111
2/3	10	1111110111	6/4	16	1111111110101000
2/4	16	1111111110001001	6/5	16	1111111110101001
2/5	16	1111111110001010	6/6	16	1111111110101010
2/6	16	1111111110001011	6/7	16	1111111110101011
2/7	16	1111111110001100	6/8	16	1111111110101100
2/8	16	1111111110001101	6/9	16	1111111110101101
2/9	16	1111111110001110	6/A	16	1111111110101110
2/A	16	1111111110001111	7/1	8	11111001
3/1	6	111010	7/2	11	11111111001
3/2	9	111110111	7/3	16	1111111110101111
3/3	11	11111110111	7/4	16	1111111110110000
3/4	16	1111111110010000	7/5	16	1111111110110001
3/5	16	1111111110010001	7/6	16	1111111110110010
3/6	16	1111111110010010	7/7	16	1111111110110011
3/7	16	1111111110010011	7/8	16	1111111110110100
3/8	16	1111111110010100	7/9	16	1111111110110101
3/9	16	1111111110010101	7/A	16	1111111110110110
3/A	16	1111111110010110	8/1	8	11111010

(Continued)

Table 7.18 ***(Continued)***

RL/Size	Code Length	VLC	RL/Size	Code Length	VLC
8/2	15	111111111000000	C/3	16	1111111111011011
8/3	16	1111111110110111	C/4	16	1111111111011100
8/4	16	1111111110111000	C/5	16	1111111111011101
8/5	16	1111111110111001	C/6	16	1111111111011110
8/6	16	1111111110111010	C/7	16	1111111111011111
8/7	16	1111111110111011	C/8	16	1111111111000000
8/8	16	1111111110111100	C/9	16	1111111111100000
8/9	16	1111111110111101	C/A	16	1111111111100001
8/A	16	1111111110111110	D/1	11	11111111010
9/1	9	111111000	D/2	16	1111111111100011
9/2	16	1111111110111111	D/3	16	1111111111100100
9/3	16	1111111111000000	D/4	16	1111111111100101
9/4	16	1111111111000001	D/5	16	1111111111100110
9/5	16	1111111111000010	D/6	16	1111111111100111
9/6	16	1111111111000011	D/7	16	1111111111101000
9/7	16	1111111111000100	D/8	16	1111111111101001
9/8	16	1111111111000101	D/9	16	1111111111101010
9/9	16	1111111111000110	D/A	16	1111111111101011
9/A	16	1111111111000111	E/1	12	111111110110
A/1	9	111111001	E/2	16	1111111111101100
A/2	16	1111111111001000	E/3	16	1111111111101101
A/3	16	1111111111001001	E/4	16	1111111111101110
A/4	16	1111111111001010	E/5	16	1111111111101111
A/5	16	1111111111001011	E/6	16	1111111111110000
A/6	16	1111111111001100	E/7	16	1111111111110001
A/7	16	1111111111001101	E/8	16	1111111111110010
A/8	16	1111111111001110	E/9	16	1111111111110011
A/9	16	1111111111001111	E/A	16	1111111111110100
A/A	16	1111111111010000	**F/0**	**12**	**111111110111**
B/1	9	111111010	F/1	16	1111111111110101
B/2	16	1111111111010001	F/2	16	1111111111110110
B/3	16	1111111111010010	F/3	16	1111111111110111
B/4	16	1111111111010011	F/4	16	1111111111111000
B/5	16	1111111111010100	F/5	16	1111111111111001
B/6	16	1111111111010101	F/6	16	1111111111111010
B/7	16	1111111111010110	F/7	16	1111111111111011
B/8	16	1111111111010111	F/8	16	1111111111111100
B/9	16	1111111111011000	F/9	16	1111111111111101
B/A	16	1111111111011001	F/A	16	1111111111111110
C/1	10	1111111010			
C/2	16	1111111111011010			

Table 7.19 JPEG VLCs for chroma AC coefficients

RL/Size	Code Length	VLC	RL/Size	Code Length	VLC
0/0	**2 (EOB)**	**00**	4/1	5	11011
0/1	2	01	4/2	8	11110111
0/2	3	100	4/3	11	11111111000
0/3	4	1010	4/4	16	1111111110011011
0/4	5	11000	4/5	16	1111111110011100
0/5	7	1110110	4/6	16	1111111110011101
0/6	9	111111000	4/7	16	1111111110011110
0/7	16	1111111110000110	4/8	16	1111111110011111
0/8	16	1111111110000111	4/9	16	1111111110100000
0/9	16	1111111110001001	4/A	16	1111111110100001
0/A	16	1111111110001001	5/1	6	111001
1/1	4	1011	5/2	9	111111010
1/2	6	111000	5/3	14	11111111100000
1/3	8	11110110	5/4	16	1111111110100010
1/4	10	1111110110	5/5	16	1111111110100011
1/5	15	111111111000010	5/6	16	1111111110100100
1/6	16	1111111110001010	5/7	16	1111111110100101
1/7	16	1111111110001011	5/8	16	1111111110100110
1/8	16	1111111110001100	5/9	16	1111111110100111
1/9	16	1111111110001101	5/A	16	1111111110101000
1/A	16	1111111110001110	6/1	7	1111001
2/1	5	11001	6/2	10	1111111000
2/2	7	1110111	6/3	16	1111111110101001
2/3	10	1111110111	6/4	16	1111111110101010
2/4	11	11111110110	6/5	16	1111111110101011
2/5	16	1111111110001111	6/6	16	1111111110101100
2/6	16	1111111110010000	6/7	16	1111111110101101
2/7	16	1111111110010001	6/8	16	1111111110101110
2/8	16	1111111110010010	6/9	16	1111111110101111
2/9	16	1111111110010011	6/A	16	1111111110110000
2/A	16	1111111110010100	7/1	6	111010
3/1	5	11010	7/2	10	1111111001
3/2	7	1111000	7/3	16	1111111110110001
3/3	9	111111001	7/4	16	1111111110110010
3/4	11	11111110111	7/5	16	1111111110110011
3/5	16	1111111110010101	7/6	16	1111111110110100
3/6	16	1111111110010110	7/7	16	1111111110110101
3/7	16	1111111110010111	7/8	16	1111111110110110
3/8	16	1111111110011000	7/9	16	1111111110110111
3/9	16	1111111110011001	7/A	16	1111111110111000
3/A	16	1111111110011010	8/1	7	1111010

(Continued)

Table 7.19 ***(Continued)***

RL/Size	Code Length	VLC	RL/Size	Code Length	VLC
8/2	12	111111110100	C/3	16	1111111111011011
8/3	16	1111111110111001	C/4	16	1111111111011100
8/4	16	1111111110111010	C/5	16	1111111111011101
8/5	16	1111111110111011	C/6	16	1111111111011110
8/6	16	1111111110111100	C/7	16	1111111111011111
8/7	16	1111111110111101	C/8	16	1111111111000000
8/8	16	1111111110111110	C/9	16	1111111111100001
8/9	16	1111111110111111	C/A	16	1111111111100010
8/A	16	1111111111000000	D/1	10	1111111010
9/1	8	11111000	D/2	16	1111111111100011
9/2	12	111111110101	D/3	16	1111111111100100
9/3	16	1111111111000001	D/4	16	1111111111100101
9/4	16	1111111111000010	D/5	16	1111111111100110
9/5	16	1111111111000011	D/6	16	1111111111100111
9/6	16	1111111111000100	D/7	16	1111111111101000
9/7	16	1111111111000101	D/8	16	1111111111101001
9/8	16	1111111111000110	D/9	16	1111111111101010
9/9	16	1111111111000111	D/A	16	1111111111101011
9/A	16	1111111111001000	E/1	12	111111110111
A/1	8	11111001	E/2	16	1111111111101100
A/2	16	1111111111001001	E/3	16	1111111111101101
A/3	16	1111111111001010	E/4	16	1111111111101110
A/4	16	1111111111001011	E/5	16	1111111111101111
A/5	16	1111111111001100	E/6	16	1111111111110000
A/6	16	1111111111001101	E/7	16	1111111111110001
A/7	16	1111111111001110	E/8	16	1111111111110010
A/8	16	1111111111001111	E/9	16	1111111111110011
A/9	16	1111111111010000	E/A	16	1111111111110100
A/A	16	1111111111010001	**F/0**	**11**	**11111111001**
B/1	8	11111010	F/1	16	1111111111110101
B/2	12	111111110110	F/2	16	1111111111110110
B/3	16	1111111111010010	F/3	16	1111111111110111
B/4	16	1111111111010011	F/4	16	1111111111111000
B/5	16	1111111111010100	F/5	16	1111111111111001
B/6	16	1111111111010101	F/6	16	1111111111111010
B/7	16	1111111111010110	F/7	16	1111111111111011
B/8	16	1111111111010111	F/8	16	1111111111111100
B/9	16	1111111111011000	F/9	16	1111111111111101
B/A	16	1111111111011001	F/A	16	1111111111111110
C/1	8	11111011			
C/2	16	1111111111011010			

182	182	185	185	185	188	186	188
180	183	182	186	189	191	191	192
183	186	185	194	180	132	85	84
181	187	191	150	44	16	16	17
183	190	155	27	13	12	12	13
185	181	37	14	14	15	15	16
195	96	16	14	14	15	14	14
159	24	13	13	12	13	14	14

(a)

54	54	57	57	57	60	58	60
52	55	54	58	61	63	63	64
55	58	57	66	52	4	–43	–44
53	59	63	22	–84	–112	–112	–111
55	62	27	–101	–115	–116	–116	–115
57	53	–91	–114	–114	–113	–113	–112
67	–32	–112	–114	–114	–113	–114	–114
31	–104	–115	–115	–116	–115	–114	–114

(b)

Figure 7.6 Actual 8 × 8 pixels from the cameraman image: (a) before level shifting and (b) after subtracting 128 from the pixel values.

Example 7.3 Let us explain the process of entropy coding the quantized luma DCT coefficients by an example using actual pixels and DCT. The 8 × 8 pixels before and after level shifting by 128 are shown in Figures 7.6a,b, respectively. The rounded DCT and the quantized DCT of the pixel block are shown in Figures 7.7a,b, respectively. Quantization is done using the default JPEG luma quantization matrix with scale factor of 1. Figure 7.8 shows the quantized DCT coefficients after zigzag scanning using the pattern of Figure 5.8. Now we can entropy code the coefficients as follows:

DC Coefficient: The difference DC coefficient is entropy coded. The quantized DC coefficient of the preceding block is found to be 26. Therefore, the difference between the current and previous DC values is $-11 - 26 = -37$. This value falls in the DC difference category of 6 (Table 7.14) whose code is **1110** form Table 7.15. The code for -37 can be obtained by adding 63 (highest value in category 6) to -37, which is 26, and the 6-bit binary code for 26 is **011010**, or it can be found by first subtracting 1 from -37 and then finding the 2s complement of -38. Therefore, the code for the DC difference of -37 is obtained by concatenating the code for -37 to the code for category 6. Thus, the DC difference has the code **1110011010**. If the DC coefficient being coded belongs to the first block in a scan line, then it is coded by itself.

AC Coefficients: The first AC coefficient is 27, which is actually a zero run followed by 27. The size category of 27 is 5. Therefore, we look for the entry

–183	302	105	31	14	1	2	–4
456	–109	–130	–72	–26	–12	–1	–2
66	–186	5	70	51	25	10	4
–9	1	83	14	–40	–36	–22	–5
–53	36	1	–49	–19	26	27	17
–19	44	–19	–6	23	16	–7	–16
–4	6	–24	17	9	–18	–12	–8
9	2	–10	16	–5	–13	17	20

(a)

–11	27	10	2	1	0	0	0
38	–9	–9	–4	–1	0	0	0
5	–14	0	3	1	0	0	0
–1	0	4	0	–1	0	0	0
–3	2	0	–1	0	0	0	0
–1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b)

Figure 7.7 8 × 8 DCT of the pixels in Figure 7.6: (a) rounded DCT of the level-shifted pixels and (b) quantized DCT using JPEG default luma quantization matrix.

corresponding to $\frac{0}{5}$ in Table 7.18, which is **11010**. The 5-bit binary code for 27 is **11011**. By concatenating the code for the nonzero coefficient value to the category code, we obtain **1101011011**. In a similar fashion, the code for 38 is **111000100110**. Let us look at finding the code for −9. Its size category is 4 and the code for $\frac{0}{4}$ is **1011**. The binary code for −9 is equivalent to the 4-bit binary code for $15 - 9 = 6$, which is **0110**. Therefore, the code for −9 is **10110110**. The size category for the run-length/amplitude of $\frac{2}{-4}$ is 3. Therefore, the code for $\frac{2}{3}$ from Table 7.18 is **1111110111**. The code for −4 is **101**. Therefore, the code for $\frac{2}{-4}$ is **1111110111101**. Continuing to the end, we find the EOB whose code from Table 7.18 corresponding to the entry $\frac{0}{0}$ is **1010**. Thus, the bit stream for the 8 × 8 block is **11100110101101011011** . . . **1010**. The total number of bits for the block is 145, which is about 2.27 bpp.

7.4.5 Huffman Decoding the Encoded Bit Stream

Huffman decoding is done a bit at a time. Consider the bit stream generated in Example 7.3. First, the DC differential value is decoded. Since no codeword is less than

–11	27	38	5	–9	10	2	–9	–14	–1	–3	2/–4
1	1/–1	3	4	2	–1	1/1	2/1	5/–1	–1	EOB	

Figure 7.8 Quantized DCT coefficients after zigzag scanning using the scan pattern of Figure 5.8.

2-bit long, let us parse the first two bits—11. There is no codeword with a code of 11. So, add a third bit, which is 110. Still, no codeword is found in the Huffman table (Table 7.15). Adding the fourth bit, the codeword is 1110 and corresponds to the code for size category 6. Since size category 6 has 6 additional bits, the next 6 bits are 011010. Because the most significant bit (MSB) is a 0, the corresponding DC differential value is negative and is equal to the one's complement of 011010, which is -37. We must add the DC value of the previous block to obtain the DC value of the current block. The previous block DC value is 26 (from Example 7.3), and therefore, the DC value of the current block is $-37 + 26 = -11$.

Next, we decode the AC coefficients. We find no codeword corresponding to the bits 11. Adding the third and fourth bits also results in no valid code. Adding the fifth bit gives us the code 11010, which corresponds to the RL/Size of $\frac{0}{5}$. Because the size category is 5, we have to decode the next 5 bits—11011. Because the MSB is 1, the AC amplitude is positive and is equal to 27. Thus far, the decoded values are -11 and 27.

Following the decoding procedure for the AC coefficient, the next six bits—111000— correspond to the RL/Size of $\frac{0}{6}$. Because the size is 6, we need to decode the next six bits, namely, 100110, which is 38. Thus, the first three decoded coefficients are -11, 27, and 38. We continue further until 64 coefficients have been decoded. We then move to decode the next block and so on.

7.5 COMPRESSION OF COLOR IMAGES

A color image is acquired in three primary colors, namely, red, green, and blue. Similarly, a color image is displayed in the three primary colors. The three components, namely, the red, green, and blue have high correlations. Because of the high correlation among the primary color components, it is not possible to process each component individually without introducing false coloring. Instead, we must first express the RGB components in three other orthogonal components and then process each component in the transformed space individually. One such component space is the *Y Cb Cr* space, where *Y* is the *luma* and *Cb* and *Cr* are the *color differences* or *chroma*. This is the color space adopted in the JPEG standard.

The human visual perception is most sensitive to the luminance and not as sensitive to colors. Therefore, it is advantageous from the compression standpoint to use lower resolutions for the chroma components. As discussed in Chapter 2, a lower resolution *Cb* or *Cr* component is obtained from its full resolution by first lowpass filtering and then subsampling. Two sampling formats allowed in JPEG are 4:2:2 and 4:2:0. In 4:2:2, chroma is filtered and subsampled by a factor of 2 in the horizontal dimension, while in 4:2:0 format, the chroma is filtered and subsampled by a factor of 2 in both dimensions. The coding procedure for chroma is the same as that for the luma except that appropriate quantization and Huffman tables are used. After decompression *Cb* and *Cr* components are upsampled to the original size and the three components are inverse transformed to obtain the RGB components for display.

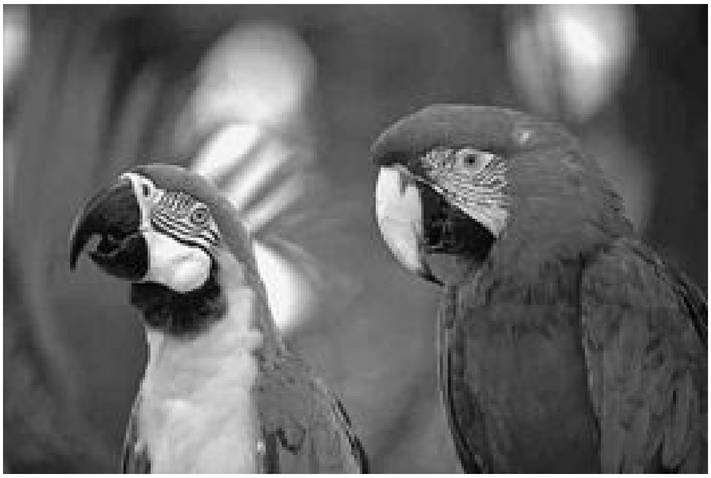


Figure 7.9 Bird image quantized/dequantized using baseline JPEG compression. The sampling format used is 4:2:0. The encoding rate is 0.38 bpp with SNRs of 21.11 dB, 18.38 dB, and 18.43 dB for the *Y*, *Cb*, and *Cr* components, respectively.

Example 7.4 In this example, we implement the baseline JPEG compression algorithm to compress/decompress a color image. The input image is assumed to have a pixel depth of 8 bpp. The algorithm uses the JPEG default quantization matrices and Huffman tables and calculates only the exact bit rate in bpp. It does not generate the compressed bit stream nor does it generate any headers. The MATLAB code allows both sampling formats, namely, 4:2:0 and 4:2:2, and uses the YCbCr color space for compression. Figure 7.9 shows the quantized/dequantized bird image at 0.38 bpp with an SNR of 21.11 dB for the *Y* component and 18.38 dB and 18.43 dB for the *Cb* and *Cr* components, respectively. The sampling format used is 4:2:0. It was found that the SNR and quality were not different for the 4:2:2 sampling format. The image quality is fairly good at this low bit rate. The MATLAB code is listed below.

```
% Example7_4.m
% JPEG baseline image compression using DCT.
% Calculates only the bit rate and does not
% generate the bitstream.
% Accepts both B&W and RGB color images.
% 4:2:0 and 4:2:2 sampling formats are allowed.
% Amount of compression can be adjusted by changing
% Qscale value.

clear
A = imread('birds.ras');  % read an image
[Height,Width,Depth] = size(A);
N = 8; % Transform matrix size
% Limit Height & Width to multiples of 8
if mod(Height,N)~= 0
    Height = floor(Height/N)*N;
```

```
end
if mod(Width,N)~= 0
    Width = floor(Width/N)*N;
end
A1 = A(1:Height,1:Width,:);
clear A
A = A1;
SamplingFormat = '4:2:0';
if Depth == 1
    y = double(A);
else
    A = double(rgb2ycbcr(A));
    y = A(:,:,1);
    switch SamplingFormat
        case '4:2:0'
            Cb = imresize(A(:,:,2),[Height/2 Width/2],'cubic');
            Cr = imresize(A(:,:,3),[Height/2 Width/2],'cubic');
        case '4:2:2'
            Cb = imresize(A(:,:,2),[Height Width/2],'cubic');
            Cr = imresize(A(:,:,3),[Height Width/2],'cubic');
    end
end
jpgQstepsY = [16 11 10 16 24 40 51 61;...
    12 12 14 19 26 58 60 55;...
    14 13 16 24 40 57 69 56;...
    14 17 22 29 51 87 80 62;...
    18 22 37 56 68 109 103 77;...
    24 35 55 64 81 104 113 92;...
    49 64 78 87 103 121 120 101;...
    72 92 95 98 112 100 103 99];
QstepsY = jpgQstepsY;
Qscale = 1.5;
Yy = zeros(N,N);
xqY = zeros(Height,Width);
acBitsY = 0;
dcBitsY = 0;
if Depth > 1
    jpgQstepsC = [17 18 24 47 66 99 99 99;...
    18 21 26 66 99 99 99 99;...
    24 26 56 99 99 99 99 99;...
    47 66 99 99 99 99 99 99;...
    99 99 99 99 99 99 99 99;...
    99 99 99 99 99 99 99 99;...
    99 99 99 99 99 99 99 99;...
    99 99 99 99 99 99 99 99];
    QstepsC = jpgQstepsC;
    YCb = zeros(N,N);
    YCr = zeros(N,N);
    switch SamplingFormat
        case '4:2:0'
            xqCb = zeros(Height/2,Width/2);
            xqCr = zeros(Height/2,Width/2);
        case '4:2:2'
            xqCb = zeros(Height,Width/2);
            xqCr = zeros(Height,Width/2);
    end
    acBitsCb = 0;
    dcBitsCb = 0;
    acBitsCr = 0;
    dcBitsCr = 0;
end
% Compute the bits for the Y component
for m = 1:N:Height
    for n = 1:N:Width
        t = y(m:m+N-1,n:n+N-1) - 128;
        Yy = dct2(t); % N x N 2D DCT of input image
        % quantize the DCT coefficients
        temp = floor(Yy./(Qscale*QstepsY) + 0.5);
        % Calculate bits for the DC difference
        if n == 1
            DC = temp(1,1);
            dcBitsY = dcBitsY + jpgDCbits(DC,'Y');
        else
            DC = temp(1,1) - DC;
            dcBitsY = dcBitsY + jpgDCbits(DC,'Y');
            DC = temp(1,1);
        end
        % Calculate the bits for the AC coefficients
        ACblkBits = jpgACbits(temp,'Y');
        acBitsY = acBitsY + ACblkBits;
        % dequantize & IDCT the DCT coefficients
        xqY(m:m+N-1,n:n+N-1)= idct2(temp .* (Qscale*QstepsY))+ 128;
    end
end
% If the input image is a color image,
% calculate the bits for the chroma components
if Depth > 1
    if strcmpi(SamplingFormat,'4:2:0')
        EndRow = Height/2;
    else
        EndRow = Height;
    end
    for m = 1:N:EndRow
        for n = 1:N:Width/2
            t1 = Cb(m:m+N-1,n:n+N-1) - 128;
            t2 = Cr(m:m+N-1,n:n+N-1) - 128;
            Ycb = dct2(t1); % N x N 2D DCT of Cb image
            Ycr = dct2(t2);
            temp1 = floor(Ycb./(Qscale*QstepsC) + 0.5);
            temp2 = floor(Ycr./(Qscale*QstepsC) + 0.5);
            if n == 1
                DC1 = temp1(1,1);
                DC2 = temp2(1,1);
                dcBitsCb = dcBitsCb + jpgDCbits(DC1,'C');
                dcBitsCr = dcBitsCr + jpgDCbits(DC2,'C');
            else
                DC1 = temp1(1,1) - DC1;
                DC2 = temp2(1,1) - DC2;
                dcBitsCb = dcBitsCb + jpgDCbits(DC1,'C');
                dcBitsCr = dcBitsCr + jpgDCbits(DC2,'C');
                DC1 = temp1(1,1);
                DC2 = temp2(1,1);
            end
            ACblkBits1 = jpgACbits(temp1,'C');
            ACblkBits2 = jpgACbits(temp2,'C');
            acBitsCb = acBitsCb + ACblkBits1;
            acBitsCr = acBitsCr + ACblkBits2;
            % dequantize and IDCT the coefficients
            xqCb(m:m+N-1,n:n+N-1)= idct2(temp1 .* (Qscale*QstepsC))+ 128;
            xqCr(m:m+N-1,n:n+N-1)= idct2(temp2 .* (Qscale*QstepsC))+ 128;
        end
    end
end
%
mse = std2(y-xqY);
snr = 20*log10(std2(y)/mse);
sprintf('SNR = %4.2f\n',snr)
if Depth == 1
    TotalBits = acBitsY + dcBitsY;
    figure,imshow(xqY,[])
    title(['JPG compressed ' '@ ' num2str(TotalBits/(Height*Width)) ' bpp'])
else
    TotalBits = acBitsY + dcBitsY + dcBitsCb +...
        acBitsCb + dcBitsCr + acBitsCr;
    c1 = imresize(xqCb,[Height Width],'cubic');
    c2 = imresize(xqCr,[Height Width],'cubic');
    mseb = std2(A(:,:,2)-c1);
    snrb = 20*log10(std2(A(:,:,2))/mseb);
    msec = std2(A(:,:,3)-c2);
    snrc = 20*log10(std2(A(:,:,3))/msec);
    sprintf('SNR(Cb) = %4.2fdB\tSNR(Cr) = %4.2fdB\n',snrb,snrc)
    xq(:,:,1) = xqY;
    xq(:,:,2) = c1;
    xq(:,:,3) = c2;
    figure,imshow(ycbcr2rgb(uint8(round(xq))))
    title(['JPG compressed ' '@ ' num2str(TotalBits/(Height*Width)) ' bpp'])
end
sprintf('Bit rate = %4.2f bpp\n',TotalBits/(Height*Width))
```

```
function Bits = jpgDCbits(dc,C)
% Bits = jpgDCbits(dc,C)
% Computes the exact number of bits to entropy code
% the DC differential coefficient using the JPEG default
% Huffman tables.
% Input:
% dc = predicted DC differential value
% C = 'Y'  for luma or 'C' for chroma
% Output:
% # of bits for the DC differential
%
% CodeLengthY contains the lengths of codes for the
% luma DC differential for size categories fro 0 to 11 inclusive.
% Code length equals the length of the Huffman code for
% the size category plus the binary code for the value.
% CodeLengthC is the corresponding table for the chroma.
```

```
CodeLengthY = int16([3 4 5 5 7 8 10 12 14 16 18 20]);
CodeLengthC = int16([2 3 5 6 7 9 11 13 15 18 19 20]);
switch C
    case 'Y'
        CodeLength = CodeLengthY;
    case 'C'
        CodeLength = CodeLengthC;
end
if dc == 0
    Bits = double(CodeLength(1));
else
    Bits = double(CodeLength(round(log2(abs(dc))+0.5)+1));
end
```

```
function Bits = jpgACbits(x,C)
% y = jpgACbits(x,C)
% Computes the exact number of bits to Huffman code
% an N x N quantized DCT block
% using JPEG Huffman table for AC coefficients.
% DCT coefficients are zigzag scanned and
% the RL/Amplitude pairs are found. Then,
% using the JPEG RL/Amp table, number of bits to
% compress the block is determined.
% Input:
%   x = N x N quantized DCT block
%   C = 'Y' for luma or 'C' for chroma
% Outputs:
%   Bits = total bits to encode the block
%
% RLCy is the JPEG run-length/size-category Huffman table
% for luma AC coefficients &
% RLCc is the corresponding Huffman table for the chroma component
% Each entry in the table is the code length for the RL/size
% category which equals the RL/size Huffman code plus the
% binary code of the actual non-zero coefficient amplitude.
% RLC{1}(1) is the EOB code corresponding to 0/0
% RLC{16}(1) is the Huffman code for RL/size = 16/0
%
RLCy = cell(1,15);
RLCy{1}=int16([4 3 4 6 8 10 12 14 18 25 26]);
RLCy{2} = int16([5 8 10 13 16 22 23 24 25 26]);
RLCy{3} = int16([6 10 13 20 21 22 23 24 25 26]);
RLCy{4} = int16([7 11 14 20 21 22 23 24 25 26]);
RLCy{5} = int16([7 12 19 20 21 22 23 24 25 26]);
RLCy{6} = int16([8 12 19 20 21 22 23 24 25 26]);
RLCy{7} = int16([8 13 19 20 21 22 23 24 25 26]);
RLCy{8} = int16([9 13 19 20 21 22 23 24 25 26]);
RLCy{9} = int16([9 17 19 20 21 22 23 24 25 26]);
RLCy{10} = int16([10 18 19 20 21 22 23 24 25 26]);
RLCy{11} = int16([10 18 19 20 21 22 23 24 25 26]);
RLCy{12} = int16([10 18 19 20 21 22 23 24 25 26]);
RLCy{13} = int16([11 18 19 20 21 22 23 24 25 26]);
RLCy{14} = int16([12 18 19 20 21 22 23 24 25 26]);
```

```
RLCy{15} = int16([13 18 19 20 21 22 23 24 25 26]);
RLCy{16} = int16([12 17 18 19 20 21 22 23 24 25 26]);
%
RLCc = cell(1,15);
RLCc{1}=int16([2 3 5 7 9 12 15 23 24 25 26]);
RLCc{2} = int16([5 8 11 14 20 22 23 24 25 26]);
RLCc{3} = int16([6 9 13 15 21 22 23 24 25 26]);
RLCc{4} = int16([6 9 12 15 21 22 23 24 25 26]);
RLCc{5} = int16([6 10 14 20 21 22 23 24 25 26]);
RLCc{6} = int16([7 11 17 20 21 22 23 24 25 26]);
RLCc{7} = int16([8 12 19 20 21 22 23 24 25 26]);
RLCc{8} = int16([7 12 19 20 21 22 23 24 25 26]);
RLCc{9} = int16([8 14 19 20 21 22 23 24 25 26]);
RLCc{10} = int16([9 14 19 20 21 22 23 24 25 26]);
RLCc{11} = int16([9 14 19 20 21 22 23 24 25 26]);
RLCc{12} = int16([9 14 19 20 21 22 23 24 25 26]);
RLCc{13} = int16([9 18 19 20 21 22 23 24 25 26]);
RLCc{14} = int16([11 18 19 20 21 22 23 24 25 26]);
RLCc{15} = int16([13 18 19 20 21 22 23 24 25 26]);
RLCc{16} = int16([11 17 18 19 20 21 22 23 24 25 26]);
%
switch C
    case 'Y'
        RLC = RLCy;
    case 'C'
        RLC = RLCc;
end
x1 = ZigZag(x); % zigzag scan the 8 x 8 DCT blocks
%
k = 2; Count = 0; Bits = double(0);
while k <= 64
    if x1(k) == 0
        Count = Count + 1;
        if k == 64
            Bits = Bits + double(RLC{1}(1));
            break;
        end
    else
        if Count == 0
            RL = Count;
            Level = round(log2(abs(x1(k)))+0.5);
            Bits = Bits + double(RLC{RL+1}(Level+1));
        elseif Count >= 1 && Count <=15
            RL = Count;
            Level = round(log2(abs(x1(k)))+0.5);
            Bits = Bits + double(RLC{RL+1}(Level));
            Count = 0;
        else
            Bits = Bits + double(RLC{16}(1));
            Count = Count - 16;
        end
    end
    k = k + 1;
end
```

```
function y = ZigZag(x)
% y = ZigZag(x)
% returns the zigzag scan addresses of an 8 x 8 array

y(1) = x(1,1);y(2) = x(1,2);y(3) = x(2,1);y(4) = x(3,1);
y(5) = x(2,2);y(6) = x(1,3);y(7) = x(1,4);y(8) = x(2,3);
y(9) = x(3,2);y(10) = x(4,1);y(11) = x(5,1);y(12) = x(4,2);
y(13) = x(3,3);y(14) = x(2,4);y(15) = x(1,5);y(16) = x(1,6);
y(17) = x(2,5);y(18) = x(3,4);y(19) = x(4,3);y(20) = x(5,2);
y(21) = x(6,1);y(22) = x(7,1);y(23) = x(6,2);y(24) = x(5,3);
y(25) = x(4,4);y(26) = x(3,5);y(27) = x(2,6);y(28) = x(1,7);
y(29) = x(1,8);y(30) = x(2,7);y(31) = x(3,6);y(32) = x(4,5);
y(33) = x(5,4);y(34) = x(6,3);y(35) = x(7,2);y(36) = x(8,1);
y(37) = x(8,2);y(38) = x(7,3);y(39) = x(6,4);y(40) = x(5,5);
y(41) = x(4,6);y(42) = x(3,7);y(43) = x(2,8);y(44) = x(3,8);
y(45) = x(4,7);y(46) = x(5,6);y(47) = x(6,5);y(48) = x(7,4);
y(49) = x(8,3);y(50) = x(8,4);y(51) = x(7,5);y(52) = x(6,6);
y(53) = x(5,7);y(54) = x(4,8);y(55) = x(5,8);y(56) = x(6,7);
y(57) = x(7,6);y(58) = x(8,5);y(59) = x(8,6);y(60) = x(7,7);
y(61) = x(6,8);y(62) = x(7,8);y(63) = x(8,7);y(64) = x(8,8);
```

7.6 BLOCKING ARTIFACT

Transform coding is also called *block coding* because it uses an $N \times N$ block of pixels at a time. The DC term of the transform is a measure of the mean of the block being transformed. For the DCT, the DC coefficient is actually N times the block mean. When the transform coefficients are quantized and dequantized, the mean values of the blocks are not reconstructed to the same original values. This imperfection becomes noticeable at low bit rates and appears as distinct blocks known as *blocking artifact*. Blocking artifact is common to all block coding methods including *vector quantization*. Figure 7.10 shows the Masuda image compressed using JPEG luminance quantization matrix with a scale factor of 2. Blocking artifacts are clearly noticeable, especially in flat areas. The intensity profile of the Masuda image along a certain row is shown in Figure 7.11 for scale factors of 1 and 5. Blockiness is recognized in areas where the intensity remains fairly constant (flat regions) by the flat profile (intensity does not follow that corresponding to scale factor 1).

7.6.1 Removal of Blocking Artifact

Several procedures have been proposed in the literature to remove or reduce the blocking artifacts [25–30]. In one such method, the *deblocking* procedure involves the steps of (a) adjusting the quantized AC DCT coefficients to minimize the MSE, (b) performing the inverse DCT, (c) classifying the image into low and high detailed areas, and (d) smoothing the classified regions appropriately. A brief description of *deblocking* procedure is as follows.

AC Coefficient Adjustment JPEG coding uses uniform quantization. This forces the DCT coefficient after dequantization to fall in the middle of the

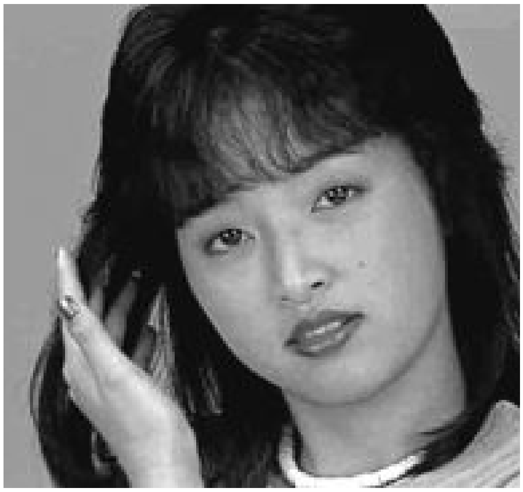


Figure 7.10 Masuda image quantized/dequantized using default JPEG luma quantization matrix with a quantizer scale of 2 to illustrate blocking artifact.

quantization bin. Because the AC coefficients have a Laplacian distribution for their amplitudes, the uniform quantizer is not PDF optimized, and therefore, the dequantization into the center of the bin is suboptimal. Instead, the reconstruction must correspond to the *centroid* of the quantization bin to achieve the minimum MSE (see Figure 7.12). One way to accomplish this is to correct or *adjust* the quantized AC

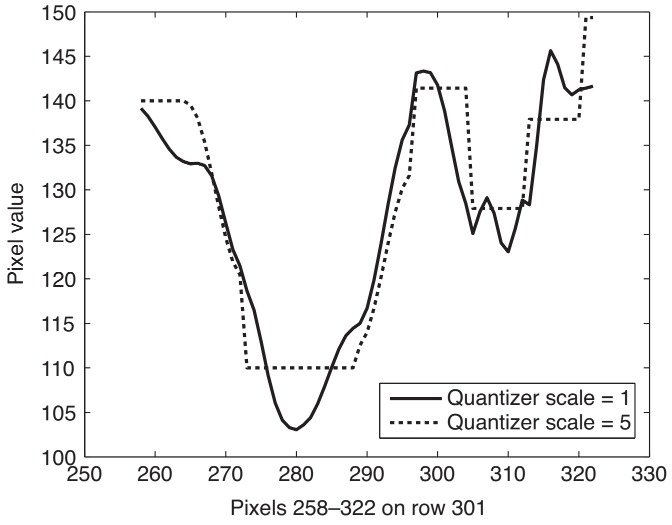


Figure 7.11 Intensity profile of Masuda image along a particular scan line for quantizer scales of 1 and 5.

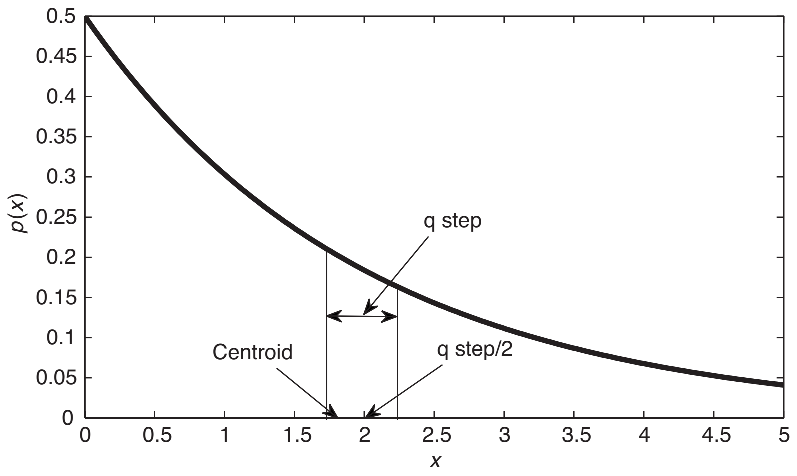


Figure 7.12 A typical Laplacian PDF of the absolute of an AC DCT coefficient showing the midpoint and centroid of a quantization bin.

coefficients using the following equations [31]:

$$\hat{X}_q(k,l) = \begin{cases} X_q(k,l) - 0.5 + \mu - \exp\left(-\frac{1}{\mu}\right) \times \left(1 - \exp\left(-\frac{1}{\mu}\right)\right), \text{if} \quad X_q(k,l) \succ 0 \\ X_q(k,l) + 0.5 - \mu + \exp\left(-\frac{1}{\mu}\right) \times \left(1 - \exp\left(-\frac{1}{\mu}\right)\right), \text{if} \quad X_q(k,l) \prec 0 \\ X_q(k,l), \text{if } X_q(k,l) = 0 \end{cases} \tag{7.24}$$

In equation (7.24), $X_q(k,l)$ is the dequantized AC DCT coefficient, $\hat{X}_q(k,l)$ is the adjusted AC DCT coefficient, and μ is the mean of the absolute of the AC coefficients.

In another approach [28], the AC coefficients are adjusted according to the following rule:

$$\hat{X}_q(k,l) = X_q(k,l) - \text{sign}\left(X_q(k,l)\right)\Delta(k,l) \tag{7.25}$$

where,

$$\Delta(k,l) = \frac{1}{2}\coth\left(\frac{\alpha(k,l)\,Q(k,l)}{2}\right) - \frac{1}{\alpha(k,l)\,Q(k,l)} \tag{7.26}$$

In equation (7.26), $Q(k,l)$ is the JPEG luma quantization matrix and $\alpha(k,l)$ is the parameter of the Laplacian distribution of the (k,l)th AC coefficient. Recall that a Laplacian PDF is characterized by a single parameter $\alpha = \sqrt{2}/\sigma$. If the parameter

$\alpha(k, l)$ of the Laplacian distribution is not known a priori, one can estimate the mean of the absolute of the AC coefficient from the quantized DCT coefficients and taking the inverse of the mean.

In either case, once the DCT coefficients are adjusted for minimum MSE, we next perform the inverse DCT to reconstruct the image. The blockiness still exists and has to be smoothed.

7.6.2 Block Classification

Blockiness is visible more in flat areas than in busy areas. Therefore, one has to smooth flat areas differently from busy areas. So, the next step is to classify the block boundaries into low and high blockiness. For this, we can use a measure of blockiness based on the block boundary variance. If A and B correspond to the original and reconstructed images, respectively, then the variance of the boundaries of the kth $N \times N$ block of pixels is defined as

$$\sigma_k^2 = \frac{1}{4N} \sum_{(i,j) \in \text{ block boundaries}} (A_k(i, j) - B_k(i, j))^2 \tag{7.27}$$

Blockiness is visible when the block boundary variance is high. A plot of block boundary variance versus quantization scale is shown in Figure 7.13. From the figure, we recognize the onset of blocking artifact around a scale factor of 1.5 (a slight change of slope).

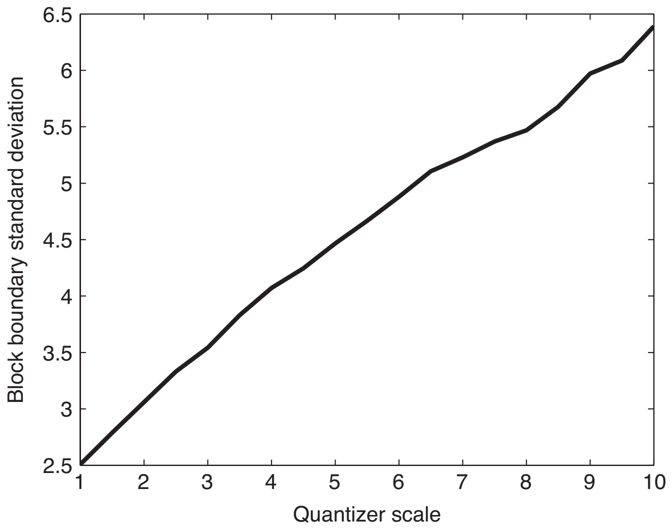


Figure 7.13 Plot of block boundary variance versus quantizer scale for Masuda image.

An alternative approach to block classification is to use the gradient measure. The gradients of an image $f[m, n]$ along the x- and y-directions are defined as

$$G_x(m, n) = \frac{f[m+1, n] - f[m-1, n]}{2} \tag{7.28a}$$

$$G_y(m, n) = \frac{f[m, n+1] - f[m, n-1]}{2} \tag{7.28b}$$

A measure of blockiness is expressed in terms of what is called the *windowed second moment matrix* and is defined as

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{12} & w_{22} \end{bmatrix} \tag{7.29}$$

where

$$w_{11} = \sum_{(i,j)\in(M\times M)} g(i, j)\, G_x^2(i, j) \tag{7.30}$$

$$w_{22} = \sum_{(i,j)\in(M\times M)} g(i, j)\, G_y^2(i, j) \tag{7.31}$$

$$w_{12} = \sum_{(i,j)\in(M\times M)} g(i, j)\, G_x(i, j)\, G_y(i, j) \tag{7.32}$$

The smoothing function $g(i, j) = e^{-(i^2+j^2)/\tau^2}$ is a Gaussian lowpass function. Because the gradient (gradient in equation (7.28) is the discrete version of the derivative) amplifies the noise, it is necessary to smooth the gradient for reliable block classification. Based on the second moment statistics in (7.29), an $M \times M$ block is high in detail if either w_{11} or w_{22} exceeds a threshold T. Otherwise, the block in question is considered low in detail. The two different blocks are treated differently for the removal of blockiness.

7.6.3 Smoothing Blockiness

Once a block (or region) is classified for blockiness, smoothing can be performed on that block (or region). According to the first approach, blockiness is reduced by replacing the adjacent block edge pixels p_1 and p_2 by

$$\hat{p}_1 = ap_1 + (1-a)\, p_2 \tag{7.33a}$$

$$\hat{p}_2 = ap_2 + (1-a)\, p_1 \tag{7.33b}$$

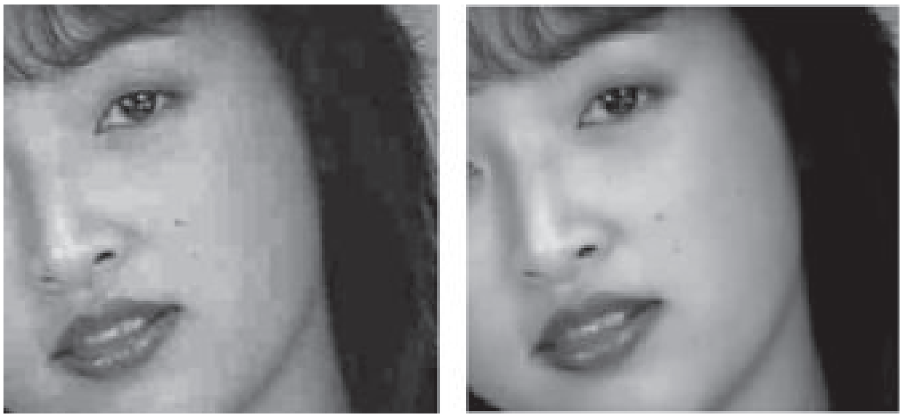


Figure 7.14 Example of deblocking: (a) Masuda image before smoothing—zoomed and (b) same image after deblocking—zoomed.

if $\sigma_k^2 \succ \sigma^2$, where σ_k^2 is the current block boundary variance and σ^2 is the desired block boundary variance for no blockiness. The constant a in equation (7.33) is chosen as

$$a = 0.5 + 0.5\frac{\sigma}{\sigma_k} \tag{7.34}$$

Example 7.5 We shall illustrate the removal of blockiness because of JPEG compression using the Masuda image. We will use equation (7.24) to adjust the AC coefficients. Note that we estimate the parameter of the Laplacian distribution based on the image in question. Then, the smoothing operation is done via equations (7.33a) and (7.33b). For this image, the desired block boundary variance is fixed at 1.25. The quantization scale factor is set at 3. Figure 7.14a shows the zoomed Masuda image before smoothing and Figure 7.14b shows the same image after smoothing. The blockiness is reduced to a great extent by the smoothing operations. The MATLAB code for this example is listed below.

```
% Example7_5.m
% Implements an algorithm to remove blockiness
% due to JPEG compression. First the quantized DCT
% coefficients are adjusted for minimum MSE based on
% the Laplacian PDF of the AC coefficients. Second,
% the reconstructed pixels are classified based on
% block boundary variance and the block edge pixels
% exceeding a threshold are smoothed

clear
%A = imread('birds.ras');  % read an image
A = imread('masuda2.ras');
```

```
[Height,Width,Depth] = size(A);
%
N = 8; % Transform matrix size
% Limit the Height & Width of the input image to multiples of N
if mod(Height,N)~= 0
    Height = floor(Height/N)*N;
end
if mod(Width,N)~= 0
    Width = floor(Width/N)*N;
end
A1 = A(1:Height,1:Width,:);
clear A
A = A1;
% If the input image is an RGB image, convert it to YCbCr image
if Depth == 1
    A1 = double(A);
else
    A = double(rgb2ycbcr(A));
    A1 = A(:,:,1);
end
% Call deBlocking function with a scale factor of 3
xq = deBlockingArt(A1,3);
figure,imshow(xq,[])
function xq = deBlockingArt(A,Qscale)
% xq = deBlockingArt(A,Qscale)
% deblocking reduces blocking artifact due to JPEG compression
% Input:
%   A = Luma component of the input image, which will be compressed,
%       decompressed and smoothed.
%   Qscale = quantizer scale to be used in the DCT coefficient
%            quantization.
% Output:
%   xq = smoothed image

N = 8;
[Height,Width] = size(A);
% Default JPEG luma quantization matrix
Qsteps = [16 11 10 16 24 40 51 61;...
    12 12 14 19 26 58 60 55;...
    14 13 16 24 40 57 69 56;...
    14 17 22 29 51 87 80 62;...
    18 22 37 56 68 109 103 77;...
    24 35 55 64 81 104 113 92;...
    49 64 78 87 103 121 120 101;...
    72 92 95 98 112 100 103 99];
%
Y = blkproc(double(A)-128,[N N],@dct2); % N x N 2D DCT of input image
Yq = zeros(Height,Width); % array to store quantized DCT coefficients
dctMean = zeros(N,N); % array for the mean of DCT coefficients
dctVar = zeros(N,N);  % array for the variance of DCT coefficients
dctAdjustType = 1; % 1 or 2
% Quantize, dequantize, adjust coefficients, and smooth blocks
switch dctAdjustType
```

```
case 1
    % Quantize and calculate the mean of the absolute of the
    % quantized DCT coefficients of the whole image
    for m = 1:N:Height
        for n = 1:N:Width
            temp = floor(Y(m:m+N-1,n:n+N-1)./(Qscale*Qsteps)...
                + 0.5);
            dctMean = dctMean + abs(temp);
            Yq(m:m+N-1,n:n+N-1) = temp;
        end
    end
    dctMean = dctMean/((Height/N)*(Width/N));
    % Now adjust AC DCT coefficient amplitude only and
    % dequantize the DCT coefficients
    dctMean(1,1) = 0;
    mu = sum(dctMean(:))/63;
    for m = 1:N:Height
        for n = 1:N:Width
            temp = Yq(m:m+N-1,n:n+N-1);
            for k = 1:N
                for l = 1:N
                    if k == 1 && l == 1
                    else
                        if temp(k,l)>0
                            temp(k,l) = temp(k,l) - 0.5
                                 + mu - exp(-1/mu)*(1-exp(-1/mu));
                        elseif temp(k,l) < 0
                            temp(k,l) = temp(k,l) + 0.5 - mu
                                 + exp(-1/mu)*(1-exp(-1/mu));
                        end
                    end
                end
            end
            Yq(m:m+N-1,n:n+N-1) = temp.*(Qscale*Qsteps);
        end
    end
    xq = blkproc(Yq,[N N],@idct2)+128; % 2D IDCT
    % perform smoothing operation
    dr = 0; dc = 0;
    for m = 1:N:Height
        for n = 1:N:Width
            temp = xq(m:m+N-1,n:n+N-1);
            dc = dc + sum(sum(A(m,n:n+N-1) - temp(1,:)));
            dc = dc + sum(sum(A(m+N-1,n:n+N-1) - temp(N,:)));
            dc = dc + sum(sum(A(m:m+N-1,n) - temp(:,1)));
            dc = dc + sum(sum(A(m:m+N-1,n+N-1) - temp(:,N)));
            dc = dc/(4*N);
            dr = dr + sum(sum((A(m,n:n+N-1) - temp(1,:)).^2));
            dr = dr + sum(sum((A(m+N-1,n:n+N-1) - temp(N,:)).^2));
            dr = dr + sum(sum((A(m:m+N-1,n) - temp(:,1)).^2));
            dr = dr + sum(sum((A(m:m+N-1,n+N-1) - temp(:,N)).^2));
            dr = dr/(4*N) - dc*dc;
            %T = 0.25/dr;% threshold for birds
```

```
T = 1.25/dr; % threshold for Masuda
if T < 1
    alfa = 0.5*(1+sqrt(T));
    if m == 1
        if n == 1
            i1 = xq(m+N-2,n:n+N-1);
            i2 = xq(m+N,n:n+N-1);
            xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
            xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
            i1 = xq(m:m+N-1,n+N-2);
            i2 = xq(m:m+N-1,n+N);
            xq(m:m+N-1,n+N-2)= alfa*i1 + (1-alfa)*i2;
            xq(m:m+N-1,n+N)= alfa*i2 + (1-alfa)*i1;
        elseif n>1 && n<Width-N
            i1 = xq(m+N-2,n:n+N-1);
            i2 = xq(m+N,n:n+N-1);
            xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
            xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
            i1 = xq(m:m+N-1,n-1);
            i2 = xq(m:m+N-1,n+1);
            xq(m:m+N-1,n-1)= alfa*i1 + (1-alfa)*i2;
            xq(m:m+N-1,n+1)= alfa*i2 + (1-alfa)*i1;
            i1 = xq(m:m+N-1,n+N-2);
            i2 = xq(m:m+N-1,n+N);
            xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
            xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
        else
            i1 = xq(m:m+N-1,n-1);
            i2 = xq(m:m+N-1,n+1);
            xq(m:m+N-1,n-1)= alfa*i1 + (1-alfa)*i2;
            xq(m:m+N-1,n+1)= alfa*i2 + (1-alfa)*i1;
            i1 = xq(m+N-2,n:n+N-1);
            i2 = xq(m+N,n:n+N-1);
            xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
            xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
        end
    elseif m>1 && m<Height-N
        if n == 1
            i1 = xq(m-1,n:n+N-1);
            i2 = xq(m+1,n:n+N-1);
            xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
            xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
            i1 = xq(m+N-2,n:n+N-1);
            i2 = xq(m+N,n:n+N-1);
            xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
            xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
            i1 = xq(m:m+N-1,n+N-2);
            i2 = xq(m:m+N-1,n+N);
            xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
            xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
        elseif n>1 && n<Width-N
            i1 = xq(m-1,n:n+N-1);
            i2 = xq(m+1,n:n+N-1);
```

```
                xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m+N-2,n:n+N-1);
                i2 = xq(m+N,n:n+N-1);
                xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n-1);
                i2 = xq(m:m+N-1,n+1);
                xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n+N-2);
                i2 = xq(m:m+N-1,n+N);
                xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
            else
                i1 = xq(m-1,n:n+N-1);
                i2 = xq(m+1,n:n+N-1);
                xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m+N-2,n:n+N-1);
                i2 = xq(m+N,n:n+N-1);
                xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n-1);
                i2 = xq(m:m+N-1,n+1);
                xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
            end
        else
            if n == 1
                i1 = xq(m-1,n:n+N-1);
                i2 = xq(m+1,n:n+N-1);
                xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n+N-2);
                i2 = xq(m:m+N-1,n+N);
                xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
            elseif n>1 && n<Width-N
                i1 = xq(m-1,n:n+N-1);
                i2 = xq(m+1,n:n+N-1);
                xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n-1);
                i2 = xq(m:m+N-1,n+1);
                xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                i1 = xq(m:m+N-1,n+N-2);
                i2 = xq(m:m+N-1,n+N);
                xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
            else
                i1 = xq(m-1,n:n+N-1);
```

```
                        i2 = xq(m+1,n:n+N-1);
                        xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                        xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                        i1 = xq(m:m+N-1,n-1);
                        i2 = xq(m:m+N-1,n+1);
                        xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                        xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                    end
                end
            end
            dr = 0; dc = 0;
        end
    end
case 2
    % Alternative method of deblocking
    for m = 1:N:Height
        for n = 1:N:Width
            temp = floor(Y(m:m+N-1,n:n+N-1)./(Qscale*Qsteps)...
                + 0.5);
            dctMean = dctMean + temp;
            dctVar = dctVar + temp.*temp;
            Yq(m:m+N-1,n:n+N-1) = temp;
        end
    end
    dctMean = dctMean/((Height/N)*(Width/N));
    dctVar = dctVar/((Height/N)*(Width/N)) - dctMean.^2;
    dctStd = sqrt(dctVar);
    dctStd(1,1) = 0;
    a = zeros(N,N);
    for k = 1:N
        for l = 1:N
            if dctStd(k,l) ~= 0
                a(k,l) = sqrt(2)/dctStd(k,l);
            end
        end
    end
    [Indx,Indy] = find(a ~= 0);
    d = zeros(N,N);
    for i = 1:length(Indx)
        d(Indx(i),Indy(i)) = (Qsteps(Indx(i),Indy(i))/2)*...
            coth(a(Indx(i),Indy(i))*Qsteps(Indx(i),Indy(i))/2)...
            -1/a(Indx(i),Indy(i));
    end
    % Adjust DCT coefficient amplitude
    for m = 1:N:Height
        for n = 1:N:Width
            temp = Yq(m:m+N-1,n:n+N-1);
            Yq(m:m+N-1,n:n+N-1) = temp.*(Qscale*Qsteps)...
                -sign(temp).*d;
        end
    end
    % Smooth the blocks
    xq = blkproc(Yq,[N N],@idct2)+128; % 2D IDCT
```

```
dr = 0; dc = 0;
for m = 1:N:Height
    for n = 1:N:Width
        temp = xq(m:m+N-1,n:n+N-1);
        dc = dc + sum(sum(A(m,n:n+N-1) - temp(1,:)));
        dc = dc + sum(sum(A(m+N-1,n:n+N-1) - temp(N,:)));
        dc = dc + sum(sum(A(m:m+N-1,n) - temp(:,1)));
        dc = dc + sum(sum(A(m:m+N-1,n+N-1) - temp(:,N)));
        dc = dc/(4*N);
        dr = dr + sum(sum((A(m,n:n+N-1) - temp(1,:)).^2));
        dr = dr + sum(sum((A(m+N-1,n:n+N-1) - temp(N,:)).^2));
        dr = dr + sum(sum((A(m:m+N-1,n) - temp(:,1)).^2));
        dr = dr + sum(sum((A(m:m+N-1,n+N-1) - temp(:,N)).^2));
        dr = dr/(4*N) - dc*dc;
        T = 1.25/dr;
        %T = 0.25/dr;
        if T < 1
            alfa = 0.5*(1+sqrt(T));
            if m == 1
                if n == 1
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n+N-2);
                    i2 = xq(m:m+N-1,n+N);
                    xq(m:m+N-1,n+N-2)= alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+N)= alfa*i2 + (1-alfa)*i1;
                elseif n>1 && n<Width-N
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n-1);
                    i2 = xq(m:m+N-1,n+1);
                    xq(m:m+N-1,n-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n+N-2);
                    i2 = xq(m:m+N-1,n+N);
                    xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
                else
                    i1 = xq(m:m+N-1,n-1);
                    i2 = xq(m:m+N-1,n+1);
                    xq(m:m+N-1,n-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                end
            elseif m>1 && m<Height-N
                if n == 1
                    i1 = xq(m-1,n:n+N-1);
                    i2 = xq(m+1,n:n+N-1);
                    xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n+N-2);
                    i2 = xq(m:m+N-1,n+N);
                    xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
                elseif n>1 && n<Width-N
                    i1 = xq(m-1,n:n+N-1);
                    i2 = xq(m+1,n:n+N-1);
                    xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n-1);
                    i2 = xq(m:m+N-1,n+1);
                    xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n+N-2);
                    i2 = xq(m:m+N-1,n+N);
                    xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
                else
                    i1 = xq(m-1,n:n+N-1);
                    i2 = xq(m+1,n:n+N-1);
                    xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m+N-2,n:n+N-1);
                    i2 = xq(m+N,n:n+N-1);
                    xq(m+N-2,n:n+N-1)= alfa*i1 + (1-alfa)*i2;
                    xq(m+N,n:n+N-1)= alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n-1);
                    i2 = xq(m:m+N-1,n+1);
                    xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                end
            else
                if n == 1
                    i1 = xq(m-1,n:n+N-1);
                    i2 = xq(m+1,n:n+N-1);
                    xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                    xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                    i1 = xq(m:m+N-1,n+N-2);
                    i2 = xq(m:m+N-1,n+N);
                    xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                    xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
```

```
                        elseif n>1 && n<Width-N
                            i1 = xq(m-1,n:n+N-1);
                            i2 = xq(m+1,n:n+N-1);
                            xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                            xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                            i1 = xq(m:m+N-1,n-1);
                            i2 = xq(m:m+N-1,n+1);
                            xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                            xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                            i1 = xq(m:m+N-1,n+N-2);
                            i2 = xq(m:m+N-1,n+N);
                            xq(m:m+N-1,n+N-2) = alfa*i1 + (1-alfa)*i2;
                            xq(m:m+N-1,n+N) = alfa*i2 + (1-alfa)*i1;
                        else
                            i1 = xq(m-1,n:n+N-1);
                            i2 = xq(m+1,n:n+N-1);
                            xq(m-1,n:n+N-1) = alfa*i1 + (1-alfa)*i2;
                            xq(m+1,n:n+N-1) = alfa*i2 + (1-alfa)*i1;
                            i1 = xq(m:m+N-1,n-1);
                            i2 = xq(m:m+N-1,n+1);
                            xq(m:m+N-1,n-1) = alfa*i1 + (1-alfa)*i2;
                            xq(m:m+N-1,n+1) = alfa*i2 + (1-alfa)*i1;
                        end
                    end
                end
                dr = 0; dc = 0;
            end
        end
end
```

7.7 VARIABLE BLOCK SIZE DCT CODING

So far we have dealt with transform coders with fixed DCT block size. With fixed block size DCT, we have no independent control over quality and bit rate simultaneously. Human visual system is less sensitive to quantization noise in high-activity areas of an image than in low-activity areas. That is to say that the threshold of visibility of quantization noise is higher in areas with details than in flat areas. That being the case, why not we exploit our own visual system's weakness to achieve much higher compression with good visual quality by hiding as much quantization noise in busy areas as possible? It is indeed possible if we use DCT with varying block sizes and quantize busy areas heavily and flat areas lightly [32].

One possible method is to start with an $N \times N$ block with $N = 2^m$, m being a positive integer. Typically, N is 16 or 32. Using the block variance as the metric, we can decompose the $N \times N$ block into four $(\frac{N}{2}) \times (\frac{N}{2})$ blocks. Each $(\frac{N}{2}) \times (\frac{N}{2})$ block, in turn, may be subdivided further into four $(\frac{N}{4}) \times (\frac{N}{4})$ blocks depending on the variance metric. This process of subdividing may be continued until left with 2×2 subblocks. This is the familiar *quad tree* decomposition. Once the quad tree decomposition is done, DCT can be applied to each subblock and the DCT coefficients quantized using suitable quantization matrices. DCTs of smaller blocks may be quantized rather heavily and of bigger subblocks lightly to achieve higher compression without sacrificing the quality. This is feasible because smaller blocks were obtained on the basis of the variance—smaller blocks have higher variances than bigger blocks. Therefore, quantization noise will be less visible in those smaller subblocks due to the human visual response.

An alternative to variance-based quad tree decomposition, one can use *local contrast* as the metric for block subdivision. It is a known fact that the block variance does not correlate well with human vision. However, human visual system responds to contrast rather than absolute intensity. Therefore, block subdivision based on local contrast is more meaningful and yields results consistent with what we perceive [33, 34]. Even if the average coding rates are the same for quad tree subdivision and for local contrast-based subdivision, the visual quality of the decompressed image in the latter case is superior to that in the former case.

Example 7.6 It will be interesting to implement a variable block size DCT coder and observe the quality improvement in the reconstructed image. We use quad tree decomposition to divide an input image into subblocks of size between 2×2 and 16×16 pixels. The block decomposition is based on the homogeneity of pixels in a block. MATLAB has the built-in function called *qtdecomp*, which divides a square block if the difference between maximum and minimum pixel values exceeds a threshold. Otherwise, the block is not divided further. Once the image is quad decomposed, the subblocks are DCT transformed, DCT coefficients are quantized and dequantized, and finally inverse DCT transformed. For the 8×8 blocks, we will use the default JPEG luma quantization matrix and quantizer scale. For the 2×2 and 4×4 blocks, we will use heavy quantization, and for the 16×16 blocks, light quantization will be used. The DC coefficient of the 2×2 and 4×4 blocks is quantized using a step size of 8. The AC coefficients of 2×2 blocks are quantized using a step size of 34, while those of 4×4 blocks are quantized using 24 as the step size. The DC coefficient of the 16×16 blocks will be quantized with a step size of 4 and all AC coefficients of the 16×16 blocks will be quantized with a step size of 16. This is not the optimum choice. We have chosen these quantization steps only to illustrate the processes involved in the implementation of a variable block size DCT coder. In order for the decoder to be able to reconstruct the image, additional information about the quad tree subdivision must be sent to the decoder. But keep in mind that this side information is an extreme tiny fraction of the overall bits.

Figure 7.15 shows the various subblocks assigned to the Masuda image as a result of the quad tree decomposition with a threshold value of 0.12. Even though the decomposition is not based on the human visual perception model, the block division appears to correlate well with the activities within the different regions. We notice 2×2 and 4×4 blocks in cluttered regions and 16×16 blocks in flat regions (in both bright and dark areas) and 8×8 blocks in areas with moderate details. The percentages of the different blocks are 9.02, 13.12, 21.61, and 56.25, respectively, for the 2×2, 4×4, 8×8, and 16×16 blocks. The SNR due to quad tree DCT coding is 24.84 dB. The reconstructed zoomed image is shown in Figure 7.16. We see very little blockiness in the reconstructed image.

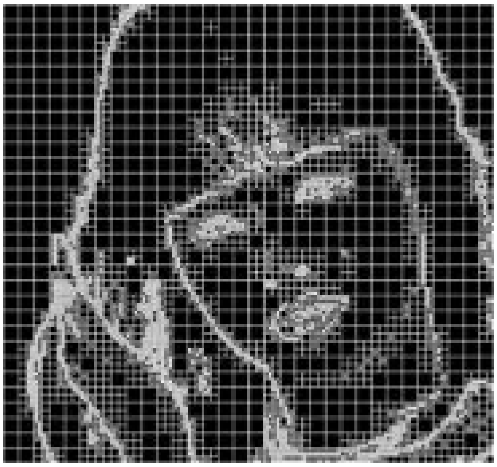


Figure 7.15 Quad tree decomposition of Masuda image showing the various subblocks of sizes 2 × 2, 4 × 4, 8 × 8, and 16 × 16. The percentage of 2 × 2 blocks is 9.02, 4 × 4 blocks is 13.12, 8 × 8 blocks is 21.61, and 16 × 16 blocks is 56.25.

The uniformity of performance of the variable block size DCT coder in terms of percentages of different subblocks is assessed by coding different images of similar content, and the results are shown in Table 7.20. We observe an increase in the number of smaller blocks (2 × 2 and 4 × 4) in images with greater details (especially in aerial, lighthouse, and yacht images). Although the cameraman image has

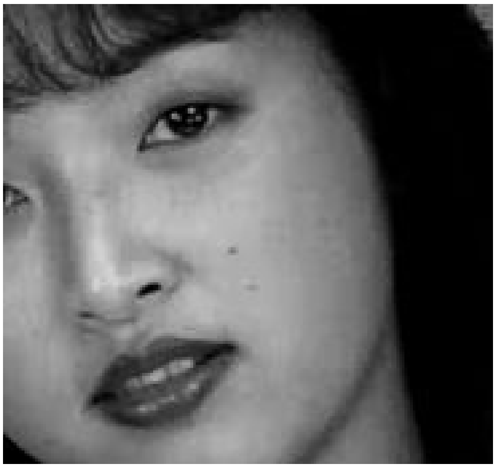


Figure 7.16 Reconstructed Masuda image using quad tree decomposition and variable size DCT with an SNR of 24.84 dB. Quantization matrix for the 8 × 8 blocks is the default JPEG luma quantization matrix. Quantization step size for the DC coefficients of 2 × 2 and 4 × 4 blocks is 8, while for 16 × 16 blocks it is 4. The AC coefficients of 2 × 2 blocks are quantized with step size of 34, while those of 4 × 4 blocks and 16 × 16 blocks are, respectively, 24 and 16.

Table 7.20 Percentage of different block sizes used in the DCT coding

	Percent Blocks				
Image	2 × 2	4 × 4	8 × 8	16 × 16	SNR (dB)
Masuda	9.02	13.12	21.61	56.25	24.84
Birds	8.89	12.04	22.95	56.12	21.92
Cameraman	**34.64**	**14.58**	**10.94**	**39.84**	**22.18**
Peppers	10.97	18.03	22.82	48.18	22.29
Aerial	**80.79**	**14.13**	**2.05**	**3.03**	**17.43**
Airplane	18.95	17.15	12.34	51.56	25.81
Lighthouse	**34.36**	**13.23**	**9.57**	**42.84**	**17.33**
Yacht	**31.30**	**22.38**	**25.49**	**20.83**	**20.73**
Hat woman	14.50	15.76	20.60	49.15	25.58
Barbara	29.27	13.57	13.76	43.40	23.86

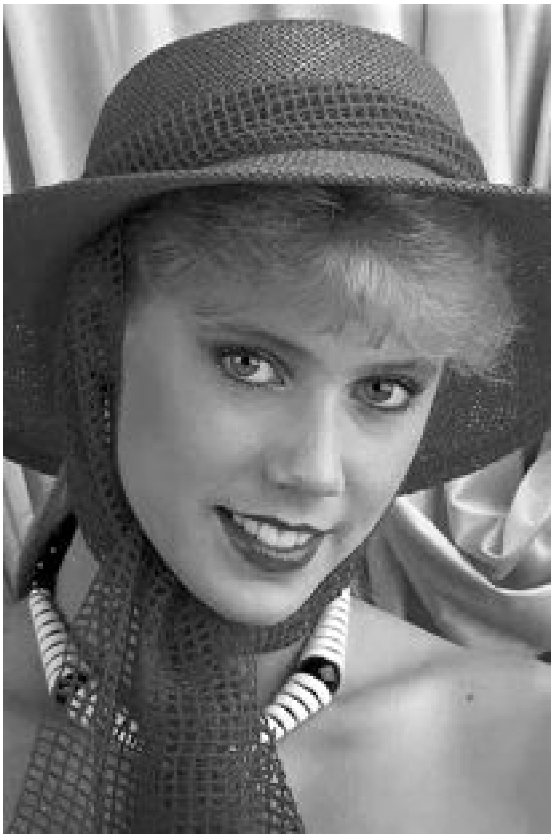


Figure 7.17 Reconstructed image of *Hat Woman* using quad tree decomposition and variable size DCT coding as in Figure 7.16. The SNR is 25.58 dB.

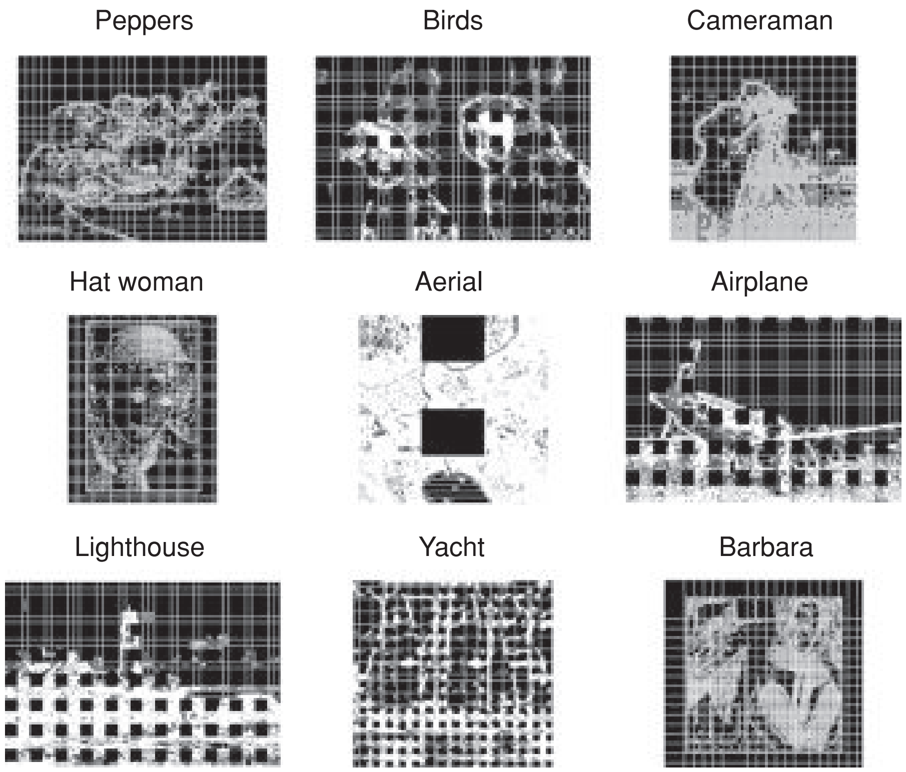


Figure 7.18 Grid structures showing the quad tree decomposition of all the images used in coding via variable size DCT.

fewer meaningful details, we see a high percentage of 2 × 2 blocks mostly in the textured ground area apparently due to the sensitivity of the quad decomposition to textures. Another point of observation is that SNR value is not truly indicative of the visual quality. For instance, the SNR for the reconstructed lighthouse is only 17.33 dB, but it has a very good visual quality. One new entry is the *hat woman*, whose reconstructed image using the variable size DCT coding is shown in Figure 7.17. Figure 7.18 shows the quad tree decomposition grid for all the images listed in Table 7.20. We have demonstrated the effectiveness of the variable size DCT coding by this example.

```
% Example7_6.m
% Uses variable size DCTs to compress an image
% Image is divided into variable size subblocks
% using quad tree decomposition. Each subblock is
% DCT transformed, quantized, dequantized, and
% inverse DCT transformed to reconstruct the image.
% Use of variable size DCT eliminates blocking
% artifact that happens with fixed size DCT.
% Minimum block size is 2 x 2
% maximum block size is 16 x 16


clear
%A = imread('masuda2.ras');
A = imread('birds.ras');
[Height,Width,Depth] = size(A);
% If image height & width are not multiples of 16,
% limit them to multiples of 16 smaller than the
% image dimensions.
if mod(Height,16)~= 0
    Height = floor(Height/16)*16;
end
if mod(Width,16)~= 0
    Width = floor(Width/16)*16;
end
A1 = A(1:Height,1:Width,:);
clear A
A = A1;
% If the input image is an RGB image, convert it to YCbCr image
if Depth == 3
    A = rgb2ycbcr(A);
    A1 = A(:,:,1);
end
clear A
A = A1;
% Invoke the function "varSizeDCTcoder with
% Thresh = 0.12 and Qscale = 2.
y = varSizeDCTcoder(A,0.12,2);


function Aq = varSizeDCTcoder(A,Thresh,Qscale)
% y = varSizeDCTcoder(A,Thresh,Qscale)
% Input:
%   A = input intensity image
%   Thresh = threshold for quad tree splitting
%   Qscale = quantizer scale to adjust the quantization
%   matrix of 8 x 8 block DCT only.
% Output:
%   Aq = reconstructed image after quad tree decomposition,
%   DCT of variable size blocks, quantization, dequantization,
%   and inverse DCT of subblocks.
% The function decompose the image with a minimum
% block size 2 x 2 and maximum block size 16 x 16
% A square block is split if the (maximum - minimum)
% of pixels is greater than the threshold "Thresh"
% "Thresh" is between 0 and 1.
%
[Height,Width] = size(A);
S = qtdecomp(A,Thresh,[2,16]);
% S is a sparse matrix and S(m,n) has the value of the block size
% at the location (m,n).
% Show block division by drawing grids
QuadBlks = repmat(uint8(0),size(S));
for dim = [2 4 8 16]
    numBlks = length(find(S==dim));
    if (numBlks > 0)
        Val = repmat(uint8(1),[dim dim numBlks]);
        Val(2:dim,2:dim,:) = 0;
        QuadBlks = qtsetblk(QuadBlks,S,dim,Val);
    end
end
QuadBlks(end,1:end) =1;
QuadBlks(1:end,end) = 1;
figure,imshow(QuadBlks,[])
%
% Now do the subblock DCT, quantize and dequantize and
% then do the subblock IDCT.
% Default JPEG Luma quantization matrix
Qsteps8 = [16 11 10 16 24 40 51 61;...
    12 12 14 19 26 58 60 55;...
    14 13 16 24 40 57 69 56;...
    14 17 22 29 51 87 80 62;...
    18 22 37 56 68 109 103 77;...
    24 35 55 64 81 104 113 92;...
    49 64 78 87 103 121 120 101;...
    72 92 95 98 112 100 103 99];
% Quantization matrices for 2x2, 4x4,and 16x16 DCTs
Qsteps2 = [8 34; 34 34];
Qsteps4 = [8 24 24 24; 24 24 24 24; 24 24 24 24; 24 24 24 24];
Qsteps16 = [4 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16;...
    16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16];
Aq = uint8(zeros(Height,Width)); % reconstructed image array
BlkPercent = zeros(4,1); % array to store % blocks of different sizes
m = 1;
for dim = [2 4 8 16]
    [x,y] = find(S == dim);
    BlkPercent(m) = length(x)*dim*dim*100/(Height*Width);
    for k = 1:length(x)
        t = dct2(double(A(x(k):x(k)+dim-1,y(k):y(k)+dim-1)));
        switch dim
```

```
            case 2
                t = round(t ./ Qsteps2) .* Qsteps2;
            case 4
                t = round(t ./ Qsteps4) .* Qsteps4;
            case 8
                t = round(t ./ (Qscale*Qsteps8)) .* (Qscale*Qsteps8);
            case 16
                t = round(t ./ Qsteps16) .* Qsteps16;
        end
        Aq(x(k):x(k)+dim-1,y(k):y(k)+dim-1) = uint8(idct2(t));
    end
    m = m + 1;
end
figure,imshow(Aq)
mse = std2(double(A)-double(Aq));
sprintf('2x2 = %5.2f%%\t4x4 = %5.2f%%\t8x8
             = %5.2f%%\t16x16 = %5.2f%%\n',BlkPercent)
sprintf('SNR = %4.2f dB\n', 20*log10(std2(double(A))/mse))
```

7.8 SUMMARY

Transform coding is one of the most powerful image compression vehicles. In general, a unitary transform has the ability to compact the energy in a block of pixels into a few transform coefficients. Among the unitary transforms, the KLT is the optimal one because it completely decorrelates the pixels. As it is image dependent, it is not quite suited for real-time image compression. On the other hand, DCT is nearly as efficient as the KLT in its energy compaction ability, is a real transform, is image independent, and is also a fast algorithm. Therefore, it is the choice for the popular compression standards such as JPEG and MPEG.

In general, the design of a transform coder amounts to (a) determining the number of bits for the coefficient quantizers based on optimal bit allocation rule and (b) entropy coding the quantized coefficients. To determine the optimal quantizer bits, one must have a statistical model for the variances of the DCT coefficients. If such a model is unavailable, then one must use the actual image to be compressed to determine the optimal quantizer bits for a given bit budget. As this is also image dependent, we have a catch 22 situation. That is why the JPEG and MPEG standards use uniform quantizers with a variable scale to control the average bit rate. The quantization steps for the quantizers are determined from a human visual perception standpoint rather than the mean square distortion measure. This may not be optimal in a mathematical sense, but is optimal from a human observer point of view. In fact, the JPEG default quantization matrices have been arrived at by using the *JND* (Just Noticeable Difference) metric and a number of subjects. An example is used to explain the process of bit allocation and quantizer design for an actual image along with MATLAB codes.

In order to convert the quantized DCT coefficients into symbols for storage or transmission, entropy coding is resorted to. The JPEG baseline coding procedure uses Huffman codes for the coefficient-to-symbol conversion. The DC and AC coefficients are entropy coded separately. As the DC coefficients of adjacent blocks are highly correlated, the DC prediction differentials are entropy coded rather than the actual DC coefficients to achieve higher compression. The DC differential values are divided into 16 size categories. For 8-bpp image, there are only 12 size categories 0–11. There are default tables for the DC size categories for luma and chroma components. Thus, to entropy code a luma DC differential we first find its size category, then use the appropriate code from the default table, and then add *size* number of bits to code the actual differential value. The same steps are used to entropy code the chroma DC differential value, except that the default chroma table is used. Again, the JPEG baseline compression technique is explained in Example 7.4 using a color image.

AC coefficients are entropy coded slightly differently. A run-length/level category is found after zigzag scanning the quantized coefficients. JPEG baseline provides Huffman codes for various run-length/level categories for both luma and chroma components. It also provides amplitude size categories for the AC coefficients (15 categories). The appropriate Huffman code for a run-length/level category is first found and the binary code for the actual nonzero amplitude that broke the zero run is appended to find the compound code for a particular run-length/level value.

Baseline JPEG yields very good quality compression at fractional bit rates that depends on the actual image in question. At low bit rates, JPEG introduces annoying artifact known as *blocking* effect. As the human visual response is more sensitive to flat regions of an image, blockiness is easily perceived in such areas. Therefore, one must resort to procedures to remove the blocking artifacts after decompression. Some researchers have proposed *lapped orthogonal transforms*, which are orthogonal transforms with overlapping blocks to eliminate blocking artifacts [35, 36]. Others have proposed methods whereby at the decoder the DCT coefficients are first adjusted to minimize the MSE followed by selective smoothing of the image to eliminate or reduce blocking artifacts. We have shown one such example of deblocking along with MATLAB codes.

Instead of fixed block size DCT, one can use DCT blocks of different sizes to achieve higher compression and better quality. Quad tree decomposition is an efficient method of decomposing an image into square blocks of varying sizes. We demonstrated this by an example whereby we observed that there was hardly any blocking artifact in the reconstructed image.

As pointed out earlier, transforming coding is not the only compression vehicle available. Wavelets have proved to be equally efficacious in realizing a high compression with good quality. We will describe coding of still pictures using 2D discrete wavelet transform in the next chapter. We will also discuss the essentials of JPEG 2000 standard in terms of wavelet-domain compression.

REFERENCES

1. A. Habibi and P. A. Wintz, "Image coding by linear transformation and block quantization," *IEEE Trans. Comm. Tech.*, COM-19 (1), 50–63, 1971.

2. P. A. Wintz, “Transform picture coding,” *Proc. IEEE*, 60 (7), 809–823, 1972.
3. W. K. Pratt, W. H. Chen, and L. R. Welch, “Slant transform image coding,” *IEEE Trans. Comm.*, COM-22 (8), 1075–1093, 1974.
4. K. R. Rao, M. A. Narasimhan, and K. Revuluri, “Image data processing by Hadamard–Haar transform,” *IEEE Trans. Comput.*, C-23 (9), 888–896, 1975.
5. K. Karhunen, “Uber lineare methoden in der wahrscheinlich-kietsrechnung,” *Ann. Acad. Sci. Fenn. A.*, 1.37, 1947. English translation by I. Selin, *On linear methods in probability theory,* Doc. T-131, The RAND Corp., Santa Monica, CA, 1960.
6. M. Loève, “Fonctions Aleatoires de Second Ordre,” in P. Levy, ed., *Processus Stochastiques et Mouvement Brownien*, Hermann, Paris, 1948.
7. H. Hotelling, “Analysis of a complex of statistical variables into principal components,” *J. Educ. Psychol.*, 24, 417–441, 498–520, 1933.
8. N. Ahmed, T. Natarajan, and K. R. Rao, “Discrete cosine transform,” *IEEE Trans. Comp.*, C-23, 90–93, 1974.
9. J.-Y. Huang and P. M. Schulthesis, “Block quantization of correlated Gaussian random variables,” *IEEE Trans. Comm.*, CS-11, 289–296, 1963.
10. A. Segall, “Bit allocation and encoding for vector sources,” *IEEE Trans. Inform. Theory*, IT-22 (2), 162–169, 1976.
11. A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer, New York, 1992.
12. N. S. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice Hall, Englewood Cliffs, NJ, 1984.
13. A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.
14. K. Sayood, *Introduction to Data Compression*, Morgan Kaufman, San Francisco, CA, 1996.
15. K. S. Thyagarajan, “DCT compression using Golomb-Rice coding,” US Patent 6735254.
16. V. R. Raveendran, K. S. Thyagarajan, J. Ratzel, S. A. Morley, and A. C. Irvine, “Apparatus and method for encoding digital image data in a lossless manner,” US Patent 7483581 B2.
17. W.-H. Chen and W. K. Pratt, “Scene adaptive coder,” *IEEE Trans. Comm.*, COM-32, 225–232, March 1984.
18. W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.
19. JPEG, *Digital Compression and Coding of Continuous-Tone Still Images*, Draft ISO 10918, 1991.
20. G. Wallace, “The JPEG still picture compression standard,” *Commun. ACM*, 34 (4), 31–44, 1991.
21. D. A. Huffman, “A method for the construction of minimum redundancy codes,” *Proc. IRE*, 40, 1098–1101, 1951.
22. J. J. Rissanen, “Generalized Kraft inequality and arithmetic coding,” *IBM J. Res. Dev.*, 20, 198–203, 1976.
23. J. J. Rissanen and G. G. Langdon, “Arithmetic coding,” *IBM J. Res. Dev.*, 23 (2), 149–162, 1979.
24. C. Poynton, *Digital Video and HDTV*, Morgan Kaufman, San Francisco, CA, 2003.
25. A. J. Ahumada, Jr., and R. Horng, “De-blocking DCT compressed images,” in B. Rogowitz and J. Allenbach, eds. *Human Vision, Visual Processing, and Digital Display V*, *Proc. SPIE*, 2179, 109–116, 1994.

26. A. J. Ahumada, Jr., and R. Horng, "Smoothing DCT compression artifacts," in J. Morreale, ed., Society for information display international symposium V*, Digest of Technical Papers*, vol. 25, pp. 708–711, 1994.
27. Y. Yang, N. Galatsanos, and A. Katsaggelos, "Iterative projection algorithms for removing the blocking artifacts of block-DCT compressed images," *Proc. IEEE ISCASSP*, V, 405–408, 1993.
28. G. A. Triantafyllidis et al., "Combined frequency and spatial domain algorithm for the removal of blocking artifacts," EURASIP *J. Appl. Signal Process.*, 6, 601–612, 2002
29. S. Wu and A. Gersho, "Enhanced video compression with standardized bit stream syntax," *Proc. IEEE ICASSP*, I, 103–106, 1993.
30. S. Wu and A. Gersho, "Enhancement of transform coding by non-linear interpolation," *Visual Communications and Image Processing '91: Visual Comm.*, vol. 1605, Bellingham, WA, pp. 487–498, 1991.
31. R. Horng and A. J. Ahumada, "A fast DCT block smoothing algorithm," *Visual Communications and Image Processing*, *SPIE*, vol. 2501, paper 5, 1995.
32. C. U. Lee, "Adaptive block-size image compression method and system," US Patent 5452104.
33. K. S. Thyagarajan and M. J. Merritt, "Contrast sensitive variance-based adaptive block-size DCT image compression," US Patent 6529634.
34. K. S. Thyagarajan and S. A. Morley, "Quality-based image compression," US Patent 6600836.
35. H. S. Malvar and D. H. Staelin, "The LOT: transform coding without blocking effects," *IEEE Trans. Acoust. Speech Signal Process.*, 4 (4), 1989.
36. R. L. de Queiroz, T. Q. Nguyen, and K. R. Rao, "GenLOT: generalized linear-phase lapped orthogonal transform," *IEEE Trans. Signal Process.*, 44 (3), 497–507, 1996.

PROBLEMS

7.1. For the transform

$$\mathbf{A} = \begin{bmatrix} \alpha & \sqrt{\mathbf{1}-\alpha^{\mathbf{2}}} \\ \sqrt{\mathbf{1}-\alpha^{\mathbf{2}}} & -\alpha \end{bmatrix}$$

and $\sigma_x^2 = 1$, show that

(a) For adjacent sample correlation ρ, the geometric mean of coefficient variances equals $\sqrt{1-4\alpha^2\rho^2\left(1-\alpha^2\right)}$.

(b) Calculate the geometric mean for $\alpha = 1/\sqrt{2}$.

(c) With $\alpha = 1/\sqrt{2}$, show that the coding gain is $G_{\mathrm{TC}} = 1/\sqrt{1-\rho^2}$.

7.2. For the Haar transform

$$\mathbf{A} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

and an input with variance σ_x^2 and correlation coefficient $\rho = 0.9$, show that

(a) The variances of the transform coefficients are given by $\sigma_0^2 = 1.9\sigma_x^2$ and $\sigma_1^2 = 0.1\sigma_x^2$.

(b) The two transform coefficients are uncorrelated.

7.3. Consider an image with adjacent pixel correlation ρ that is the same in both dimensions. With transform only in one dimension $\mathbf{X} = \mathbf{Ax}$, where $\mathbf{A}$ is as given in Problem 7.2.

(a) Show that the ratio of the arithmetic and geometric means of the coefficient variances equals $1/\sqrt{1-\rho^2}$.

(b) For the transform in both dimensions $\mathbf{X} = \mathbf{AxA^T}$, show that the ratio of the arithmetic and geometric means of the transform coefficients is $1/(1-\rho^2)$ when the autocorrelation function is separable.

(c) Calculate the ratio of the arithmetic to geometric means for $\rho = 0.95$.

(d) If the autocorrelation function is assumed to be isotropic with a correlation between diagonally opposite elements of $\rho^{\sqrt{2}}$, show that a loss in coding gain of 2.2 dB results compared to the isotropic case for $\rho = 0.95$.

7.4. Find the optimal bit assignments for a 2×2 image and transform of Problem 7.3 for an average bit rate of 2 bpp. Assume a quantizer performance factor of 1, $\sigma_x^2 = 1$, and $\rho = 0.95$.

7.5. For an intensity image of your choice, obtain the 8×8 KLT and find the optimal bit allocation for a bit budget of 1 bpp. Compare this result with (a) sine transform and (b) DCT, both of size 8×8. You can use the MATLAB tools to solve the problem.

7.6. Repeat Problem 7.5 to obtain the optimal bits using integer bit assignment rule.

7.7. Develop MATLAB codes to write out the compressed bits to a specified file of a DCT coder using JPEG rule and tables.

7.8. Verify that the compressed bit stream you wrote to a file in Problem 7.7 is correct by reading the compressed file and decoding and displaying the image.

7.9. Use a different metric, say block variance, to decompose an intensity image into variable size square blocks and quantize and dequantize the variable size DCT coefficients and calculate the SNR of the reconstructed image. Increase the quantization level and observe the onset of blocking or other artifacts. Comment on the observed phenomenon.

7.10. Discuss the blocking artifact induced by the Hadamard transform-based quantizer on a real intensity image. Use a transform size of 8×8 pixels.

7.11. Quantize and dequantize an RGB image using the JPEG rule and tables for 4:2:0 and 4:2:2 sampling formats and explain the nature of the blocking artifact. Is it different from the corresponding intensity image? Does color alter the perception? Use a quantizer scale of 2.

8

IMAGE COMPRESSION IN THE WAVELET DOMAIN

8.1 INTRODUCTION

Compression of image data is achieved by removing the correlation among the pixels as in predictive coding or by compacting pixel energy in a few coefficients as in transform coding. This later principle is equally applicable to wavelet domain compression [1–3]. The main difference between transform coding and wavelet domain coding is that wavelet transform is typically applied to the whole image unlike the transform coding where the transform is applied to subblocks. When we mention wavelet domain or wavelet transform, we mean *discrete wavelet transform* (DWT). In two-dimensional (2D) DWT, an image is transformed into four distinct sets of coefficients giving an approximation of the image at a lower resolution, and three details with orientations in the horizontal, vertical, and diagonal directions. This is called one level of wavelet transform. In *octave-band* decomposition the approximation coefficients of the first level are again decomposed into four sets of coefficients to obtain a two-level 2D DWT. This can be iterated further. On the other hand, in a full-tree DWT, each *subband* at each level is split into four subbands to obtain the next higher level DWT. The design of a DWT coder is similar to that of the transform coder. We will discuss the issues involved in the design of a DWT coder in what follows.

8.2 DESIGN OF A DWT CODER

Designing a DWT image coder involves the (1) selection of a suitable 2D discrete wavelet, (2) number of levels of decomposition, (3) assignment of quantizer bits to

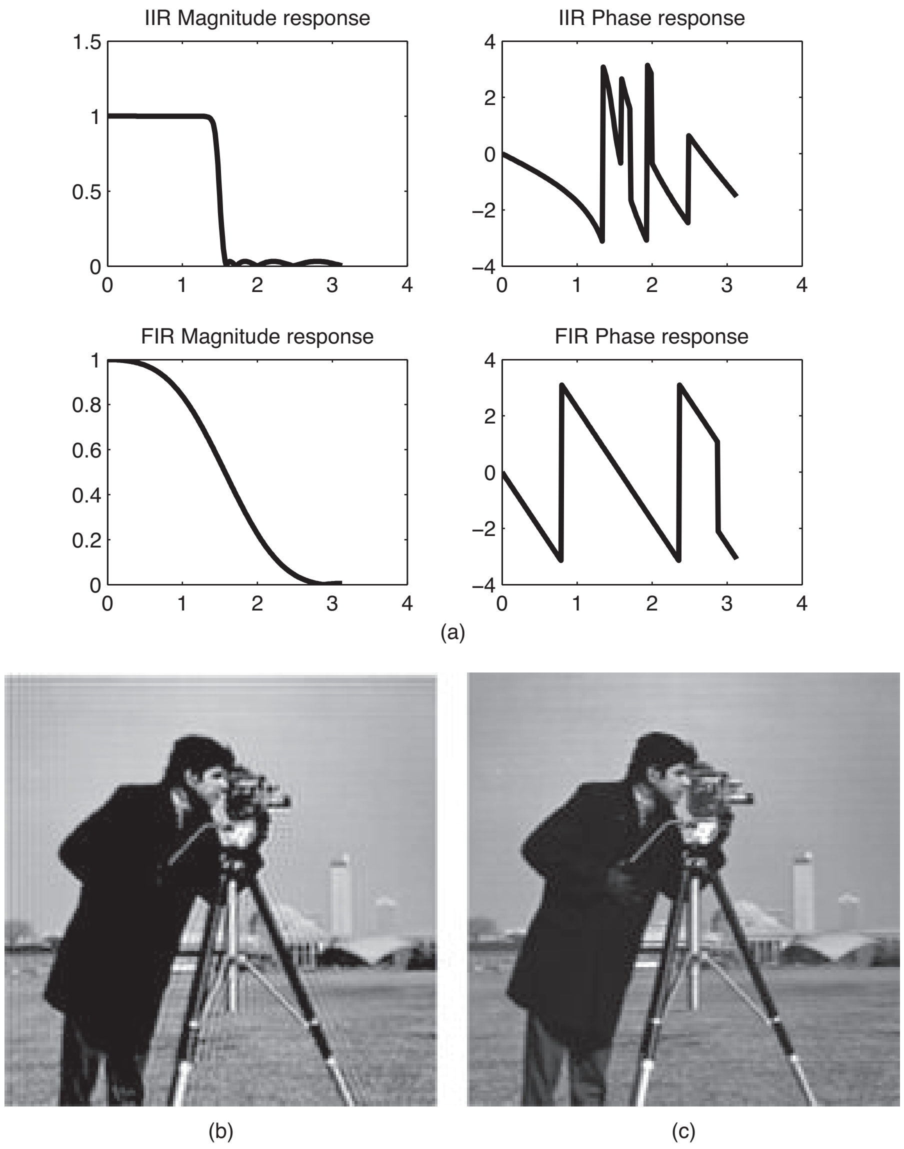


Figure 8.1 Phase distortion due to filtering the cameraman image: (a) magnitude and phase responses: the top plot shows ninth-order IIR filter, and the bottom plot nine-tap FIR filter, (b) ninth-order lowpass Chebychev IIR-filtered image, and (c) nine-tap linear phase FIR-filtered image.

all the coefficients to meet the bit budget in an optimal fashion, (4) design of the quantizers, and (5) entropy coder to convert the coefficients into binary codes for transmission or storage. Each of the design factors will be described in the following.

8.2.1 Choice of a Wavelet

As we learnt in an earlier chapter, the 2D DWT is implemented efficiently via subband coding scheme using FIR filters [4–6]. It is also known that a wavelet can be obtained by iterating a set of suitable lowpass and highpass filters. Therefore, to choose a wavelet is to choose a suitable filter bank. Digital filters have a number of constraints that must be considered when selecting a wavelet and they are explained in the following.

Linear Phase Distortion due to nonlinear phase in an image is annoying, especially around the edges. Figure 8.1b shows the cameraman image filtered with a ninth-order Chebychev lowpass IIR filter. Due to the nonlinear phase of the filter (see Figure 8.1a, top figure), the edges in the image undergo severe distortion. As a comparison, Figure 8.1c is the result of filtering the same image with a linear phase FIR filter of the same order as the IIR filter (see Figure 8.1a, bottom figure). We see no phase distortion in the FIR filtered image. MATLAB code to illustrate phase distortion is listed below. When more than one level of decomposition is used in a DWT, iterated filtering can cause severe distortion if filters with nonlinear phase response are used. Therefore, it is preferable to choose linear phase filters in implementing the DWT. Generally speaking, FIR filters de facto have linear phase characteristics. On the other hand, IIR filters do not generally have linear phase response. Moreover, since linear phase FIR filters have symmetry in their coefficients, they are efficient to implement. One problem though with a four-band subband coding system is that it is not possible to have linear phase FIR filters that are also orthogonal. That is that it is not possible to implement an orthogonal wavelet transform with FIR filters having exact linear phase characteristics. However, an orthogonal wavelet transform can be implemented with IIR filters that satisfy both linear phase and orthogonality [7]. Overall, linear phase FIR filters are preferred to filters with nonlinear phase response.

```
% ShowPhaseDistortion.m
% Illustration of the effect of non-linear
% phase on the filtered image
% IIR filter with non-linear phase and
% FIR filter with linear phase are considered.
% IIR filter is 9th order and FIR filter has 9 taps.


clear
% design IIR Chebychev filter of order N, stopband
% ripple A dB down, and stopband edge frequency Ef
% Ef should be 0 < Ef < 1
A = 30;
```

```
N = 9;
Ef = 0.5;
[b,a] = cheby2(N,A,Ef);
[H,W] = freqz(b,a,256);
subplot(2,2,1),plot(W,abs(H),'k','LineWidth',2)
title('IIR Magnitude response')
subplot(2,2,2),plot(W,angle(H),'k','LineWidth',2)
title('IIR Phase response')
%
% design a lowpass FIR filter with N taps (N-1-st order)
% with passband edge at Ef
b1 = fir1(N-1,Ef);
a1 = zeros(size(b1));
a1(1) = 1;
[H1,W1] = freqz(b1,a1,256);
subplot(2,2,3),plot(W1,abs(H1),'k','LineWidth',2)
title('FIR Magnitude response')
subplot(2,2,4),plot(W,angle(H1),'k','LineWidth',2)
title('FIR Phase response')
%
% read an image and filter it through IIR & FIR filters
f = imread('cameraman.tif');
%f = imread('birds.ras');
[Height,Width,Depth] = size(f);
if Depth == 3
    f1 = rgb2ycbcr(f);
    f = f1(:,:,1);
    clear f1
end
% filter row by row first
g = zeros(Height,Width);
y = zeros(Height,Width);
for m = 1:Height
g(m,:) = filter(b,a,double(f(m,:)));
end
% next filter along columns
for m = 1:Width
y(:,m) = filter(b,a,double(g(:,m)));
end
% Next, filter the original input image through linear
% phase FIR
% filter
% The function "filter2" does the 2D filtering
g1 = filter2(b1,f,'same');
% Display the images
figure,imshow(uint8(round(y))),title('Filter with
    nonlinear phase')
figure, imshow(uint8(round(g1))), title('Filter with
    linear phase')
```

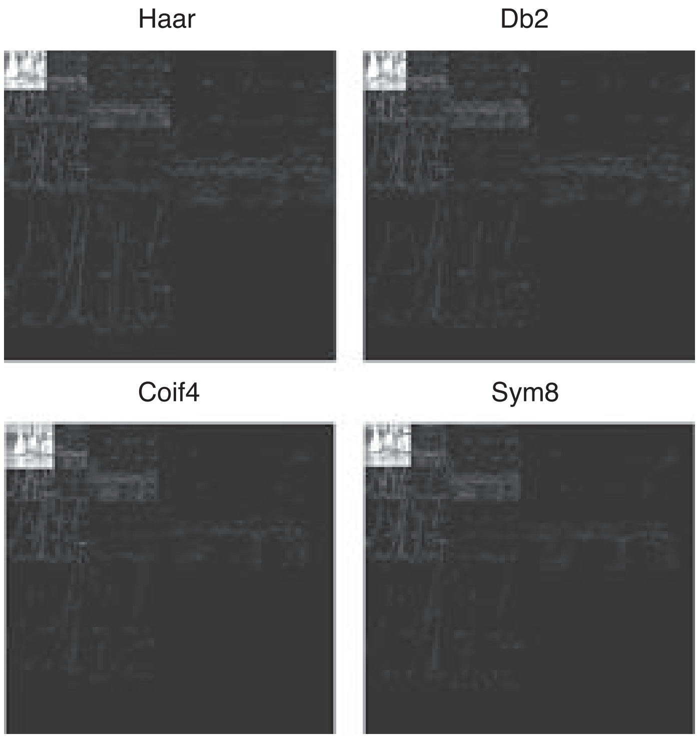


Figure 8.2 Three-level 2D DWT of the yacht image using: (a) Haar wavelet, (b) “db2” wavelet, (c) “coif4” wavelet, and (d) “sym8” wavelet. The images are enhanced using the MATLAB function “wcodemat” so that the detail coefficients are visible.

Orthogonality An important property of an orthogonal transform is that it conserves energy. This also implies that the total distortion due to quantization of the DWT coefficients is equal to the sum of the distortions in each subband. Therefore, it allows us to quantize the individual subbands to different levels to achieve a high compression and good visual quality. So, orthogonal filter banks effect a unitary transform, which, in turn, allow us to assign different bit rates to the different subbands to meet the overall bit budget. Figure 8.2 shows a three-level 2D DWT of the yacht image using the wavelets of Haar, Daubechies 2 (db2), coiflet 4 (coif4), and sym8. The coiflet wavelet compacts the energy in the approximation coefficients the most and Haar the least. Table 8.1 lists the actual percentage of energy contained in the subbands. Figure 8.3 gives a graphical illustration of the energy compaction ability of the four wavelets considered. As expected, the orthogonal wavelets do better in compacting energy than the nonorthogonal wavelet “bior1.5.”

Filter Length Higher order filters are useful because of better out of band rejection characteristics. However, filters with long lengths will cause *ringing* effect around the edges, which may be objectionable. Further, higher order filters are

Table 8.1 Percentage of energy in three-level 2D DWT coefficients

Band	Haar	Db2	Coif4	Sym8	Bior1.5
LL3	99.19	99.35	99.54	99.48	99.12
HL3	0.084	0.055	0.029	0.033	0.036
LH3	0.076	0.044	0.024	0.027	0.030
HH3	0.017	0.012	0.006	0.007	0.004
HL2	0.128	0.117	0.080	0.092	0.145
LH2	0.125	0.090	0.065	0.075	0.113
HH2	0.040	0.037	0.028	0.031	0.041
HL1	0.129	0.106	0.089	0.103	0.213
LH1	0.165	0.133	0.096	0.107	0.177
HH1	0.051	0.055	0.041	0.046	0.118

computationally more intensive than lower order filters. Therefore, it is preferable to use shorter filters.

Regularity A filter is called regular if its iteration tends to a continuous function. From a compression point of view, a regular orthogonal filter bank is preferred because it introduces fewer artifacts especially when more than one level of DWT decomposition is used.

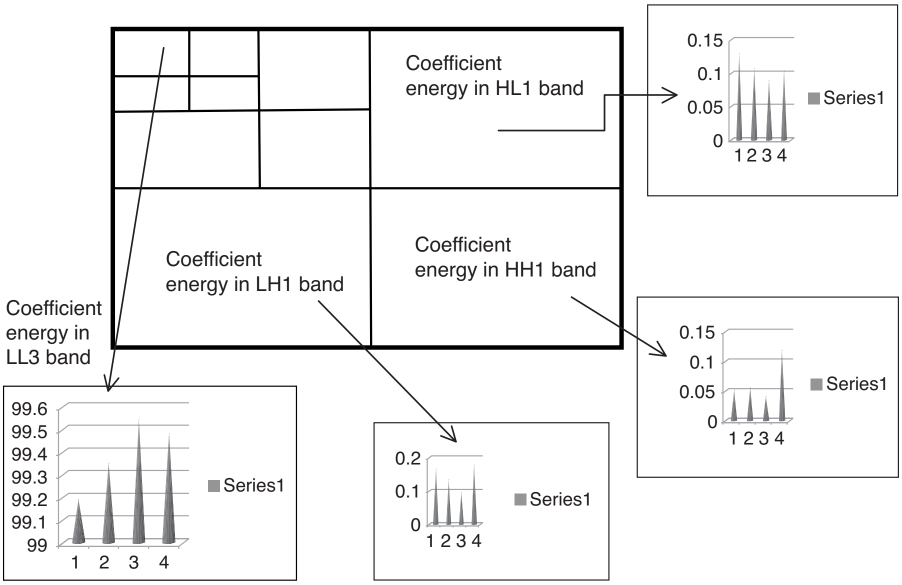


Figure 8.3 Graphical illustration of the energy compaction ability of the 2D DWT: the bands are denoted LL for the approximation coefficients, HL for the detail coefficients with edges oriented horizontally, LH for the detail coefficients with edges oriented vertically, and HH for the detail coefficients with edges oriented diagonally. The number in each band denotes the DWT level. Thus, LL3 corresponds to the approximation coefficients in level 3, and so on. In the bar chart, the data series order is Haar, db2, coif4, and sym8.

In conclusion, it is preferable to use shorter, smoother linear phase filters with moderate regularity for image compression in the wavelet domain.

8.2.2 Number of Levels of DWT

The number of levels of DWT to be used in the coding scheme is flexible and depends on the application. But it must be pointed out that due to iteration of the DWT, the quantization distortion may become objectionable at low bit rates. Of course, the computational load increases with the number of levels though not proportionately because the subband size decreases with the levels. Therefore, one must strike a balance between quality and bit rate to arrive at the appropriate number of levels of DWT to use.

8.2.3 Optimal Bit Allocation

Energy is conserved if the wavelets employed are orthogonal. In such a case, the subbands are orthogonal and we can use the same procedure for allocating bits to the quantizers as the procedure used in discrete cosine transform (DCT) coding. On the other hand, if the wavelets are not orthogonal, we can still use the bit allocation rule but it will only be approximate [7,8]. Thus, for an L-level 2D DWT of an image with a bit budget of R bpp, the number of bits for the quantizer in the ith subband can be obtained from

$$R_i = R + \frac{1}{2}\log_2\left(\frac{\sigma_i^2}{\left(\prod_{k=1}^{3L+1}\sigma_k^2\right)^{1/(3L+1)}}\right), \quad 1 \le i \le 3L+1 \tag{8.1}$$

Note that an L-level 2D DWT of an image has $3L + 1$ subbands. The individual quantizer bits may be negative or noninteger. One can use the same integer bit assignment rule as was used in the transform coder for the calculation of optimal bit assignment with integer number of bits for the quantizers. When using integer bit assignment, one must first calculate the variances of the DWT coefficients of all the subbands and then apply the integer bit assignment rule.

8.2.4 Quantization of the DWT Coefficients

Once the optimal number of bits for the quantizers is determined, the various DWT coefficients must be quantized. As we have seen in transform coding, one can use the Lloyd–Max (nonuniform) quantizers or the uniform quantizers for the quantization of the DWT coefficients.

Quantization of the Approximation Coefficients The lowest band LL (highest level) DWT coefficients are an approximation to the original image at a lower resolution that depends on the DWT level. As a result, the lowest band coefficients have characteristics similar to the original image (Figure 8.4a, top plot is the histogram of the approximation coefficients). In order to achieve higher compression, the lowest band coefficients can be quantized individually either by Lloyd–Max quantizers or uniform quantizers or can be quantized using DCT and DPCM or just by DPCM.

Lloyd–Max Quantizer for Detail Coefficients The detail coefficients of the 2D DWT of an image at all levels have a generalized zero mean Gaussian distribution with a fast decay [9] (see Figure 8.4) as given in equation (8.2):

$$p\left(x\right) = \frac{\beta}{2\alpha\Gamma\left(\frac{1}{\beta}\right)} e^{-\left(\left|x-\mu\right|/\alpha\right)^{\beta}} \tag{8.2}$$

In equation (8.2), the Gamma function $\Gamma\left(z\right)$ is defined as

$$\Gamma\left(z\right) = \int_{0}^{\infty} t^{z-1} e^{-t} dt \tag{8.3}$$

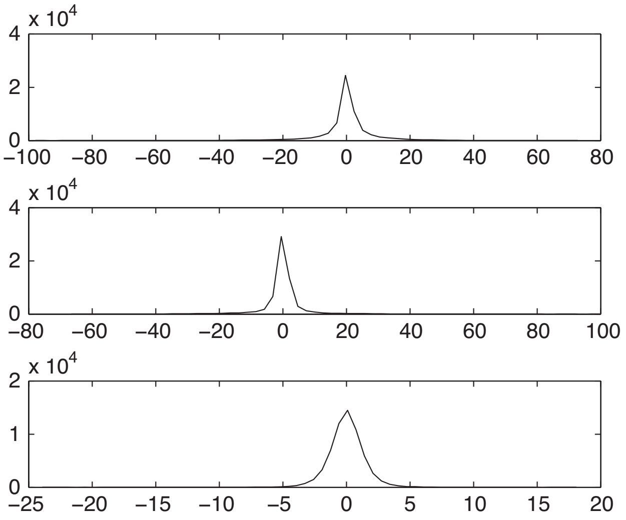
For positive integers,

$$\Gamma\left(n\right) = \left(n-1\right)! \tag{8.4}$$

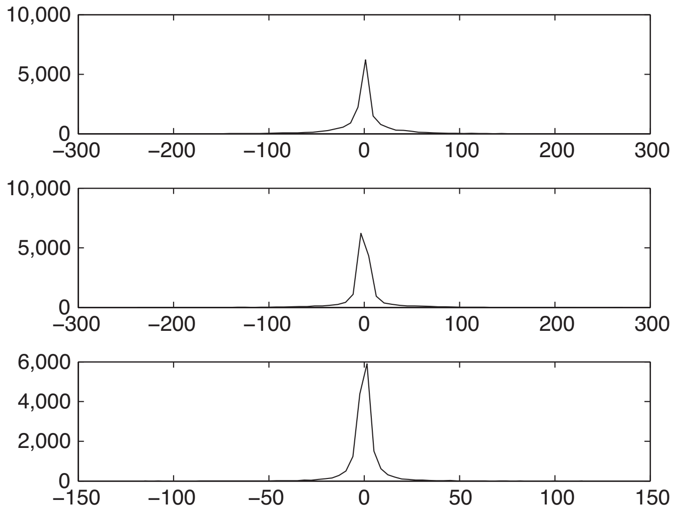
For $\beta = 1$, equation (8.2) corresponds to the Laplacian distribution, and for $\beta = 2$, we get the Normal distribution. Figure 8.5 shows the plotting of equation (8.2) for values of β from 1 to 5 with $\alpha = 2.5$. A comparison of Figures 8.4 and 8.5 clearly shows that the histograms of the detail coefficients do follow the generalized Gaussian function. As the detail coefficients contain only edge information, one can design Lloyd–Max quantizers for the detail DWT coefficients optimized to their probability density functions (PDFs).

Uniform Quantizer for Detail Coefficients Since entropy coding follows quantization, uniform quantizers are nearly optimal and are also easy to implement. Therefore, one can use uniform quantizers to quantize all the DWT coefficients including the approximation coefficients. It has been proven that uniform quantizers with *dead zone* of twice the quantization step size around the origin are efficient form a compression point of view without loss of quality [10]. One reason for this is that the dead zone thresholds out noise in the higher bands.

Example 8.1 Let us illustrate what we have learnt so far about image compression in the wavelet domain by an example. Let us use orthogonal 2D DWT with up to three levels and uniform quantizers for the quantization of the DWT coefficients. We will further use the optimal bit allocation rule (equation 8.1) to calculate the number of bits for the uniform quantizers to meet a bit budget of 1 bpp. For this example,



(a)



(b)

Figure 8.4 Histograms of the 2D DWT coefficients corresponding to the db2 wavelet: (a) level 3, (b) level 2, and (c) level 1. In (a), the top plot is the histogram of the approximation coefficients, the other three are the histograms of the HL, LH, and HH coefficients in level 3. Similarly, in (b) and (c), the order of the plots is HL, LH, and HH from top to bottom. All the histograms follow the generalized Gaussian shape except, of course, the approximation coefficients.

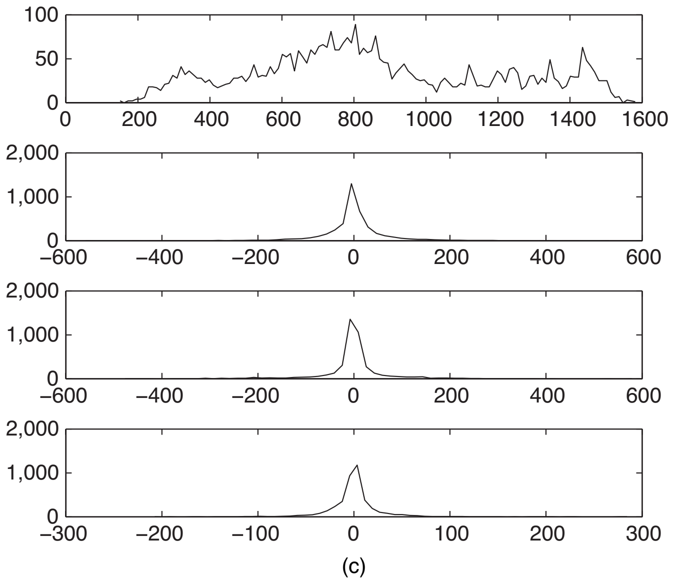


Figure 8.4 (*Continued*)

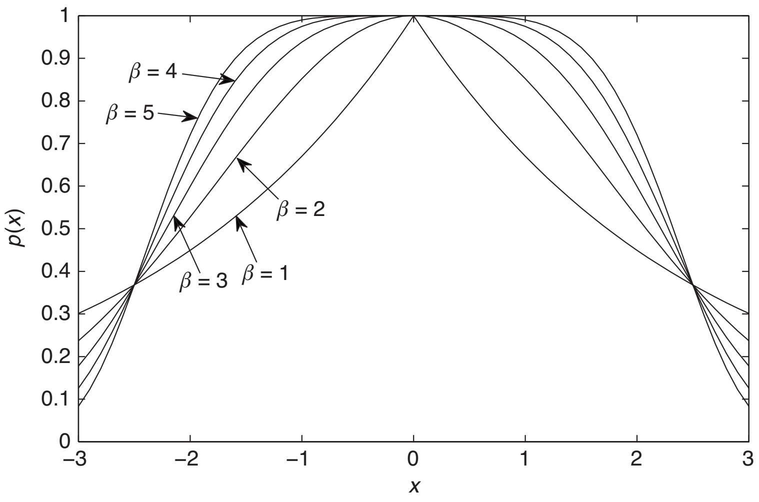


Figure 8.5 Generalized Gaussian function: the plotting shows the generalized Gaussian distribution function of equation (8.2) for values of β between 1 and 5, with $\alpha = 2.5$. For $\beta = 1$, the function corresponds to the Laplacian distribution.

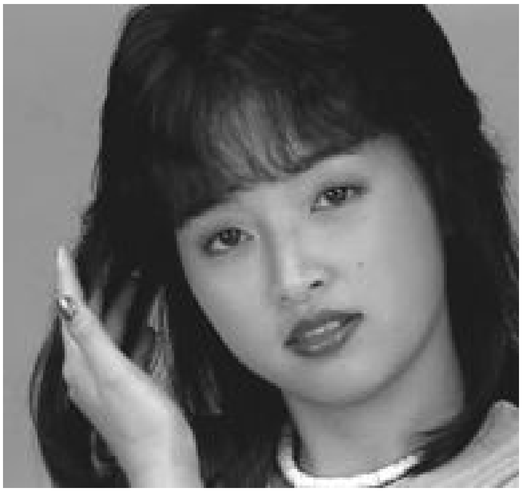


Figure 8.6 Example of still image compression in the wavelet domain: reconstructed Luma component of the Masuda image with a bit budget of 1 bpp and an actual bit rate of 0.5654 bpp. The SNR for the *Y*, *Cb*, and *Cr* components are, respectively, 24.94, 19.06, and 19.85 dB using "db2" wavelet. The chroma sampling format used is 4:2:2.

let the wavelet be Daubechies "db2," which has a length of 4 for its analysis and synthesis filters. The "db2" wavelet is orthogonal and so it implements a unitary transform. For color images, the sampling format to be used is 4:2:2.

Solution First, we level shift the input image components by 128 (input image has 8 bits/component), and then apply the color coordinate transformation to obtain YCbCr components from the input RGB components. The chroma components will then be lowpass filtered and decimated by a factor of 2 in the horizontal dimension only to get 4:2:2 sampling format. Next, we compute the 2D DWT of the luma and decimated chroma components to three levels. After having computed the 2D DWT of the components, we determine the number of bits to the quantizers of the DWT coefficients using the optimal bit assignment rule as per equation (8.1). With quantizer bits known, we next calculate the quantization steps of the uniform quantizers. We then quantize and dequantize the DWT coefficients using the uniform quantizers and perform inverse 2D DWT. The *Cb* and *Cr* components are upsampled to the full resolution, and inverse component transformation (ICT) is carried out to obtain the reconstructed RGB image. Finally, we calculate the signal-to-noise ratio (SNR) in dB and display the original and reconstructed images for comparison.

Figure 8.6 shows the luma component of Masuda image compressed in the wavelet domain. Unlike the DCT-based compression, we do not see any blocking artifact and the visual quality is decent at around 0.6 bpp. Table 8.2 lists the SNR due to wavelet-based quantization/dequantization of the images used in Example 7.3. As a comparison, Table 8.3 lists the SNR for "coif2" orthogonal wavelets, which have

Table 8.2 SNR due to wavelet domain compression: wavelet used is “db2”

Image	bpp	*Y* (dB)	*Cb* (dB)	*Cr* (dB)
Aerial	0.2800	11.23	NA	NA
Airplane	0.5335	21.91	17.78	14.65
Autumn	0.8082	23.00	11.80	10.09
Birds	0.5422	19.36	24.61	25.68
Cameraman	0.3100	16.71	NA	NA
Hat woman	0.5550	22.89	23.04	22.06
Lighthouse	0.5090	12.36	19.77	20.72
Masuda	0.5654	24.94	19.06	19.85
Peppers	0.5697	21.08	20.94	21.03
Yacht	0.6461	18.34	18.35	18.04

Table 8.3 SNR due to wavelet domain compression: wavelet used is “coif2”

Image	bpp	*Y* (dB)	*Cb* (dB)	*Cr* (dB)
Aerial	0.3200	11.74	NA	NA
Airplane	0.6453	21.53	18.24	15.69
Autumn	0.9534	22.80	11.61	10.21
Birds	0.6740	18.76	27.64	26.42
Cameraman	0.4100	16.60	NA	NA
Hat woman	0.6295	23.05	23.19	22.14
Lighthouse	0.5974	12.51	20.84	21.92
Masuda	0.7018	25.20	19.28	21.28
Peppers	0.7118	21.91	22.05	20.89
Yacht	0.7781	18.48	18.83	18.52

FIR filters of length 12. In terms of visual quality, the performance of the “db2” and “coif2” wavelets are approximately the same. However, “coif2” performs slightly better than “db2” in terms of SNR, especially for the chroma components. It is found that level shifting yields a slightly higher SNR than without level shifting!

```
% Example8_1.m
% Image compression in the wavelet domain
% An L-level 2D DWT of an input image is computed
% using "db8" orthogonal wavelet. Optimal quantizer
% bits are computed for a bit budget of R (typically 1) bpp
% and then the quantization steps are computed.
% Uniform quantization is used across all the bands.
% if the input image RGB, it is converted to YCbCr
% components, Cb and Cr components are down sampled to
% 4:2:0 or 4:2:2 format and quantized.
% User may specify the values for R, L, and sampling format.
% Parameters are:
%   SamplingFormat = "4:2:0" or "4:2:2"
%   L = number of levels of DWT
%   R = bit budget in bpp
%   BitAssignRule = "optimal" or "integer"

clear
A = imread('yacht.ras');
L = 3; % number of DWT levels
% make sure that the image size is divisible by NL
[x,y,z] = size(A);
if mod(x,2^L) ~=0
    Height = floor(x/(2^L))*(2^L);
else
    Height = x;
end
if mod(y,2^L) ~=0
    Width = floor(y/(2^L))*(2^L);
else
    Width = y;
end
Depth = z;
clear x y z
%
SamplingFormat = '4:2:2'; % for RGB image
wName = 'db2'; % wavelet name. See MATLAB for available wavelets
%wName = 'coif2';
% bit budget: R1 for Y, R2 for Cb & R3 for Cr
R1 = 1.0; R2 = 0.5; R3 = 0.5;
BitAssignRule = 'optimal';
if Depth == 1
    % input image is B/W
    Y = double(A(1:Height,1:Width))-128;
else
    % Input image is RGB. Convert it to YCbCr
    % using JPEG2000 reversible component transformation
    A1 = double(A(1:Height,1:Width,1:Depth))-128;
    Y = 0.299*A1(:,:,1) + 0.587*A1(:,:,2) + 0.144*A1(:,:,3);
    Cb = -0.16875*A1(:,:,1) - 0.33126*A1(:,:,2) + 0.5*A1(:,:,3);
    Cr = 0.5*A1(:,:,1) - 0.41869*A1(:,:,2) - 0.08131*A1(:,:,3);
    switch SamplingFormat
        % subsample chroma components
        case '4:2:0'
            Cb = imresize(Cb,[Height/2 Width/2],'cubic');
            Cr = imresize(Cr,[Height/2 Width/2],'cubic');
        case '4:2:2'
            Cb = imresize(Cb,[Height Width/2],'cubic');
            Cr = imresize(Cr,[Height Width/2],'cubic');
    end
end
%
[C,S] = wavedec2(Y,L,wName); % L-level 2D DWT of the Luma
```

```
% Compute optimal quantizer bits
switch BitAssignRule
    case 'optimal'
        [Qbits,Qsteps] = DWToptimalBits(C,S,wName,R1);
    case 'integer'
        [Qbits,Qsteps] = AssignIntgrBits2DWT(C,S,wName,R1);
end
% find total bits for the Y component
TotalBitsY = (S(L+1,1)*S(L+1,2))*sum(Qbits(1:3))+...
    (S(L,1)*S(L,2))*sum(Qbits(4:6))+(S(L-1,1)*S(L-1,2))
           *sum(Qbits(7:10));
sprintf('Number of levels of DWT = %d\nquantizer %s bits\n',L,
         BitAssignRule)
disp(Qbits(1:3*L+1)')
%
% Quantize and dequantize the coefficients
% Function "quantizeDWT" quantizes and dequantizes the DWT
% coefficients
Cq = quantizeDWT(C,S,Qsteps);
Yq = waverec2(Cq,S,wName); % do inverse 2D DWT of quantized Y
     coefficients
SNR = 20*log10(std2(Y)/std2(Y-Yq));
% Calculate SNR of Y component
sprintf('SNR(Y) = %4.2fdB\n',SNR)
if Depth == 1
    figure,imshow(Yq+128,[])
    AvgBit = TotalBitsY/(Height*Width);
    sprintf('Desired avg. rate = %3.2f bpp\tActual avg. rate
                               = %3.2f bpp\n',R1,AvgBit)
end
% If the input image is RGB, quantize the Cb & Cr components
if Depth > 1
    [C,S] = wavedec2(Cb,L,wName); % L-level 2D DWT of Cb
    switch BitAssignRule
        case 'optimal'
            [Qbits,Qsteps] = DWToptimalBits(C,S,wName,R2);
        case 'integer'
            [Qbits,Qsteps] = AssignIntgrBits2DWT(C,S,wName,R2);
    end
    % find total bits for the Cb component
    TotalBitsCb = (S(L+1,1)*S(L+1,2))*sum(Qbits(1:3))+...
    (S(L,1)*S(L,2))*sum(Qbits(4:6))+(S(L-1,1)*S(L-1,2))
           *sum(Qbits(7:10));
    sprintf('Cb quantizer bits: ')
    disp(Qbits(1:3*L+1)')
    Cq = quantizeDWT(C,S,Qsteps); % quantize Cb DWT coefficients
    Cbq = waverec2(Cq,S,wName);% do 2D IDWT of quantized Cb
    SNRcb = 20*log10(std2(Cb)/std2(Cb-Cbq));
    %
    [C,S] = wavedec2(Cr,L,wName); % L-level 2D DWT of Cr
    switch BitAssignRule
        case 'optimal'
            [Qbits,Qsteps] = DWToptimalBits(C,S,wName,R3);
        case 'integer'
            [Qbits,Qsteps] = AssignIntgrBits2DWT(C,S,wName,R3);
    end
    % find total bits for the Cr component
    TotalBitsCr = (S(L+1,1)*S(L+1,2))*sum(Qbits(1:3))+...
    (S(L,1)*S(L,2))*sum(Qbits(4:6))+(S(L-1,1)*S(L-1,2))
           *sum(Qbits(7:10));
    sprintf('\nCr quantizer bits: ')
    disp(Qbits(1:3*L+1)')
    % Find the overall average bit rate in bpp.
    AvgBit = (TotalBitsY+TotalBitsCb+TotalBitsCr)/(Height*Width);
    sprintf('Actual avg. rate = %5.4f bpp\n',AvgBit)
    Cq = quantizeDWT(C,S,Qsteps); % quantize Cr DWT coefficients
    Crq = waverec2(Cq,S,wName); % do 2D IDWT of quantized Cr
          coefficients
    SNRcr = 20*log10(std2(Cr)/std2(Cr-Crq));
    % upsample Cb & Cr to full resolution
    Cbq = imresize(Cbq,[Height Width],'cubic');
    Crq = imresize(Crq,[Height Width],'cubic');
    Ahat = zeros(Height,Width,Depth);
    % Do inverse component transformation & add DC level
    Ahat(:,:,1) = Yq + 1.402*Crq + 128;
    Ahat(:,:,2) = Yq - 0.34413*Cbq - 0.71414*Crq + 128;
    Ahat(:,:,3) = Yq + 1.772*Cbq + 128;
    figure,imshow(uint8(round(Ahat)))
    sprintf('Chroma sampling = %s\n',SamplingFormat)
    sprintf('SNR(Cb) = %4.2fdB\tSNR(Cr) = %4.2fdB\n',SNRcb,SNRcr)
end

function [Qbits,Qsteps] = DWToptimalBits(C,S,wName,R)
% [Qbits,Qsteps] = DWToptimalBits(C,S,wName,R)
% Assign quantizer bits for the L-level 2D DWT
% coefficients using the optimal bit allocation rule.
% Quantizer bits are rounded to nearest integers and
% negative bits are assigned zero values.

L = size(S,1) - 2; % number of DWT levels
Coef = cell(3*L+1,1); % cell array to store L-level DWT
       coefficients
% extract the detail and approximation DWT coefficients
j = 1;
for k = 1:L
    [Coef{j},Coef{j+1},Coef{j+2}] = detcoef2('all',C,S,k);
    j = j+3;
end
```

```
Coef{3*L+1} = appcoef2(C,S,wName,L);
% Compute and store the coefficient variances
CoefVar = zeros(3*L+1,1);
for k = 1:L
    k1 = (k-1)*3;
    for j = 1:3
        CoefVar(k1+j) = std2(Coef{k1+j})*std2(Coef{k1+j});
        if CoefVar(k1+j) == 0
            CoefVar(k1+j) = 1;
        end
    end
end
CoefVar(3*L+1) = std2(Coef{3*L+1})*std2(Coef{3*L+1});
% Geometric mean of variances
gm = 1;
p = 1.0/(3*L+1);
for j = 1:L
    j1 = (j-1)*3;
    gm = gm * prod(CoefVar(j1+1:j1+3));
end
gm = (gm*CoefVar(3*L+1))^p;
%
Qbits = zeros(3*L+1,1);
% compute quantizer bits using coefficient variances and
% geometric mean of the variances
for k = 1:3*L+1
    Qbits(k) = round(R + 0.5*log2(CoefVar(k)/gm));
    if Qbits(k) < 0
        Qbits(k) = 0;
    end
end
% Compute the quantization steps
% note that zero quantizer bit is not assigned infinity to Qstep
Qsteps  = zeros(3*L+1,1);
for k = 1:3*L+1
    maxCoef = max(max(Coef{k}));
    D = maxCoef;
    if D ~= 0
        Qsteps(k) = D/(2*2^Qbits(k));
    else
        Qsteps(k) = 1.0e+16;
    end
end


function [Qbits,Qsteps] = AssignIntgrBits2DWT(C,S,wName,R)
%   Assigns DWT coefficient quantizer bits optimally using
%   recursive integer bit allocation rule.
```

```
L = size(S,1) - 2;
Coef = cell(3*L+1,1);
Rtotal = (3*L+1)*R;   % total bits for the N x N block
j = 1;
for k = 1:L
    [Coef{j},Coef{j+1},Coef{j+2}] = detcoef2('all',C,S,k);
    j = j+3;
end
Coef{3*L+1} = appcoef2(C,S,wName,L);
CoefVar = zeros(3*L+1,1);
for k = 1:L
    k1 = (k-1)*3;
    for j = 1:3
        CoefVar(k1+j) = std2(Coef{k1+j})*std2(Coef{k1+j});
        if CoefVar(k1+j) == 0
            CoefVar(k1+j) = 1;
        end
    end
end
CoefVar(3*L+1) = std2(Coef{3*L+1})*std2(Coef{3*L+1});
Qbits = zeros(3*L+1,1);
%
while Rtotal > 0
    Max = -9999;
    for k = 1:3*L+1
        if Max < CoefVar(k)
                Max = CoefVar(k);
                Indx = k;
        end
    end
    Qbits(Indx) = Qbits(Indx) + 1;
    CoefVar(Indx) = CoefVar(Indx)/2;
    Rtotal = Rtotal - 1;
end
Qsteps  = zeros(3*L+1,1);
for k = 1:3*L+1
    maxCoef = max(max(Coef{k}));
    D = maxCoef;
    if D ~= 0
        Qsteps(k) = D/(2*2^Qbits(k));
    else
        Qsteps(k) = 1.0e+16;
    end
end


function y = quantizeDWT(C,S,Qsteps)
% y = quantizeDWT(C,S,Qsteps)
% quantizes uniformly the L-level DWT coefficients
% using the quantization steps calculated
% from the optimal bit assignment rule.
% Input:
%   C = vectors of DWT coefficients obtained from
%   wavedec2.
%   S = book keeping variables, also obtained from wavedec2.
%   Qsteps = quantization steps corresponding to the
%   optimal bits or integer bits  assignment.
%
% Output:
%   y = quantized/dequantized and inverse discrete wavelet
%   transformed image.



L = size(S,1) - 2; % number of DWT levels
% extract the DWT coefficients
appCoefLength = S(1,1)*S(1,2); % length of app coef.
for k = 1:L
    k1 = (k-1)*3+1;
    k2 = k1 +2;
    detCoefLength(k1:k2) = S(k+1,1)*S(k+1,2);
end
temp = C(1:appCoefLength);
% quantize the approximation coefficients
temp = floor(temp/Qsteps(3*L+1)+0.5)*Qsteps(3*L+1);
y(1:appCoefLength) = temp;
startLoc = appCoefLength; % length of app. coef. vector
% quantize the detail coefficients
for k = 1:3*L
    temp = C(startLoc+1:startLoc+detCoefLength(k));
    temp = floor(temp/Qsteps(k)+0.5)*Qsteps(k);
    y(startLoc+1:startLoc+detCoefLength(k)) = temp;
    startLoc = startLoc + detCoefLength(k);
end
```

8.2.5 Entropy Coding of the Quantized DWT Coefficients

The last step in wavelet-based image compression is entropy coding. Because most detail coefficients are zero after quantization, run-length coding can be used to convert the coefficients into symbols for storage or transmission. One can use any of the entropy coding methods, namely, Huffman, arithmetic, or Golomb–Rice, discussed in Chapter 5 for coding the quantized DWT detail coefficients. As pointed out, the approximation coefficients can be compressed using predictive or DCT method followed by Huffman coding.

8.3 ZERO-TREE CODING

"Zero-tree" means a tree of zero coefficients. In octave-band 2D DWT, the LL band at each level is divided into four bands belonging to the next higher level and so on in a quad tree fashion. Thus, a coefficient in each subband in a given level and orientation except the LL band has a corresponding set of 4 coefficients (actually, 2×2) in the next lower level at the same orientation because the 2D DWT coefficients are obtained by iterated filtering and subsampling by 2 in each spatial dimension. Figure 8.7a shows the relationships of the 2D DWT coefficients in a three-level DWT and the quad tree structure is shown in Figure 8.7b. It so happens in DWT that if a coefficient at the highest level in any of HL, LH, or HH bands is either zero or *insignificant*, then all the corresponding coefficients in the lower levels will also most likely be zero or insignificant. This property is exploited in the zero-tree coding of the DWT coefficients to achieve a high compression. Unlike the DCT-based coder where the entropy coding is confined to each 8×8 (or 16×16) block, in wavelet-based coder much larger blocks have to be scanned to sort out the coefficients for entropy coding. As such one has to spend a lot of bits to code the coefficient position information. This is reduced in what is known as the *embedded zero-tree wavelet*

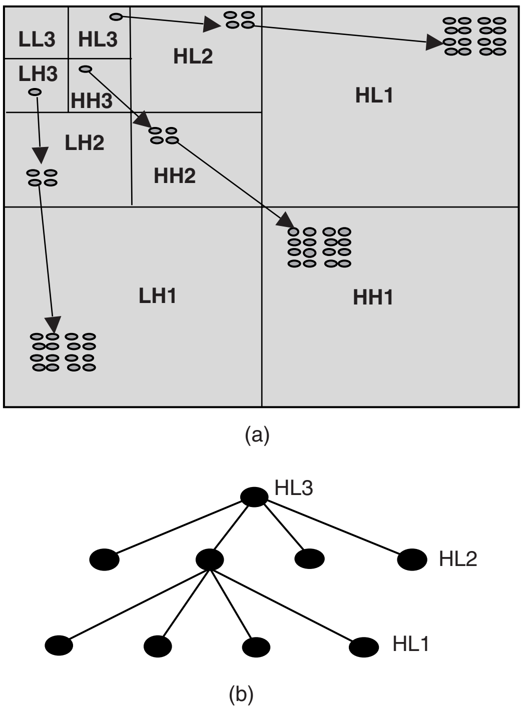


Figure 8.7 Depiction of parents and children of a zero-tree in a three-level 2D DWT: (a) relationship of the DWT coefficients in a three-level 2D DWT and (b) corresponding quad tree structure.

Table 8.4 8 × 8 pixels from the cameraman image after level shifting by 128

58	63	59	60	59	53	27	16
51	57	57	52	58	62	50	26
36	48	46	34	43	49	51	41
29	43	45	31	40	48	51	56
51	50	54	47	47	46	45	46
48	46	49	48	44	42	42	36
26	20	25	27	21	18	15	12
4	6	7	4	4	1	−10	−17

(EZW) coding technique [11, 12]. A scheme similar to EZW is called the set partitioning in hierarchical trees (SPIHT) [13]. EZW is a very cleaver method of coding the coefficient position information in combination with *successive approximation* of the coefficient values. The word "embedded" refers to the fact that the coding (or decoding) can terminate at any desired bit rate because the coarse coefficient values along with refinements are embedded in the coded bit stream.

The HL, LH, and HH coefficients at the highest level are considered the parents in the family tree, the four coefficients in the next lower level corresponding to the parent are the children, and the four coefficients in the next lower level are the grand children, and so on. With this terminology, we can explain how the DWT coefficients are entropy coded. EZW consists of two passes, one for the encoding of the coefficient positional information and the other for coding the approximate coefficient value. These two passes are continued until a required bit budget is met. As in DCT coding, in EZW one has to scan all the DWT coefficients to convert them into symbols for storage or transmission. One scanning method is the familiar raster scanning. But, because there are levels of DWT coefficients, scanning is carried out from the highest level (parents) to the lowest level (distant relatives). The significant coefficient values are coded using *bit plane* coding. That is, we start with the absolute maximum coefficient value to half the nearest integer power of 2 as the threshold, and successively halve the threshold to refine the quantization. Perhaps the EZW coding method is best explained by the following example.

Example 8.2 Consider the 8 × 8 pixels of the cameraman image starting at row 150 and column 180 after level shifting by 128 shown in Table 8.4 and its three-level 2D DWT in Table 8.5, respectively. The wavelet used is "bior1.5". Note that the MATLAB function "dwt2" has the optional parameter "mode," which when set to "per" (periodization), yields each band exactly one quarter the size of the next lower level band. Thus, we have 1 coefficient in each of the level-3 bands, 2 × 2 in the next lower level, and 4 × 4 in the first-level bands.

First, we have to find a threshold value in order to determine the significance of a coefficient. Since the largest absolute coefficient value is 300, the initial threshold is

$$T_0 = 2^{\lfloor \log_2 300 \rfloor} = 256 \tag{8.5}$$

Table 8.5 8 × 8 three-level 2D DWT of the pixels in Table 8.4

300	69	31	−12	7	5	−5	−18
29	−11	71	82	4	1	3	−8
−1	25	−2	38	4	1	2	5
−4	12	1	−13	17	19	17	27
−7	2	1	19	0	−3	5	−7
−14	14	−8	3	0	−2	1	7
2	3	1	−1	−1	2	−1	−3
2	−1	3	4	4	−3	−1	−3

Scanning the coefficients from LL3, we find 300 to be greater than T_0 and is assigned the symbol *POS*. The quantization interval is (256,512] and the midpoint is 384. Therefore, 300 is quantized to 384. Next, the HL3 coefficient value 69 is less than T_0 and its descendents are also found to be insignificant (less than T_0). Therefore, it is assigned the symbol *ZTR* for zero-tree root and quantized to 0. Similarly, we find the LH3 coefficient 20 and its descendents to be insignificant with respect to T_0 and assign the symbol ZTR and quantized to 0. The HH3 coefficient −11 and its descendents are similarly found to be insignificant. Therefore, the HH3 coefficient is assigned the symbol ZTR and quantized to 0 as well. At this point, we have covered all the coefficients and the *dominant list* looks like the one shown in Table 8.6. The symbols in the dominant list can be entropy coded and transmitted or stored. Since EZW only generates a few symbols, it is more efficient to use arithmetic coding. It is also necessary to transmit the initial threshold, number of levels of DWT, and the size of the image as side information so as to enable the decoder to initiate the decoding process.

A second list called the *subordinate list* at this first pass contains the code for the refined quantized value as shown in Table 8.7. For the first subordinate list, the threshold is half that of T_0, that is, $T_1 = T_0/2 = 128$. The quantization intervals are (256,384] and (384,512]. Since the significant coefficient 300 is in the first or lower interval, it is quantized to 320—the midpoint of the interval (256,384] and a “0”

Table 8.6 First dominant pass: $T_0 = 256$

Position	Coef. Value	Symbol	Quantized Value
LL3	300	POS	384
HL3	69	ZTR	0
LH3	20	ZTR	0
HH3	−11	ZTR	0

Table 8.7 First subordinate list: $T_1 = 256$

Coef. Value	Code	Quantized Value
300	0	320

Table 8.8 Reconstructed pixels from the first subordinate pass

168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168
168	168	168	168	168	168	168	168

is sent or stored. Had the coefficient value been in the upper interval, a "1" would be sent, instead. If we were to reconstruct the 8×8 pixels from the coefficients obtained at the end of the first subordinate pass, we will obtain the pixels as shown in Table 8.8.

In the second pass, we have to scan the coefficients that were found to be insignificant in the previous pass. The threshold value is $T_2 = 128$, and the dominant list is shown in Table 8.9. Again, the symbols from the second dominant list will be sent or stored using arithmetic coding. The refinement in the coefficient approximation is contained in the second subordinate list as shown in Table 8.10. The quantization intervals are (128, 192], (192, 256], (256, 320], (320, 384], (384, 448], and (448, 512] and a "0" or "1" bit is assigned to each interval in an alternating fashion, as shown in Figure 8.8. Thus, the coefficient value 300 has the code "010" and the approximated coefficient value is 288. Using this approximate value the corresponding reconstructed pixels are shown in Table 8.11.

Continuing further, we obtain the dominant and subordinate lists in the third pass as shown in Tables 8.12 and 8.13, respectively. The thresholds for the third pass are 64 for the dominant list and 32 for the subordinate list, respectively. The symbols from the dominant list and the interval in which the coefficient value lies in the subordinate list are transmitted or stored at each pass. The reconstructed pixels at the end of the third pass are shown in Table 8.14. The various quantities of interest

Table 8.9 Second dominant pass: $T_2 = 128$

Position	Coef. Value	Symbol	Quantized Value
HL3	69	ZTR	0
LH3	20	ZTR	0
HH3	−11	ZTR	0

Table 8.10 Second subordinate list: $T_3 = 64$

Coef. Value	Code	Quantized Value
300	010	288

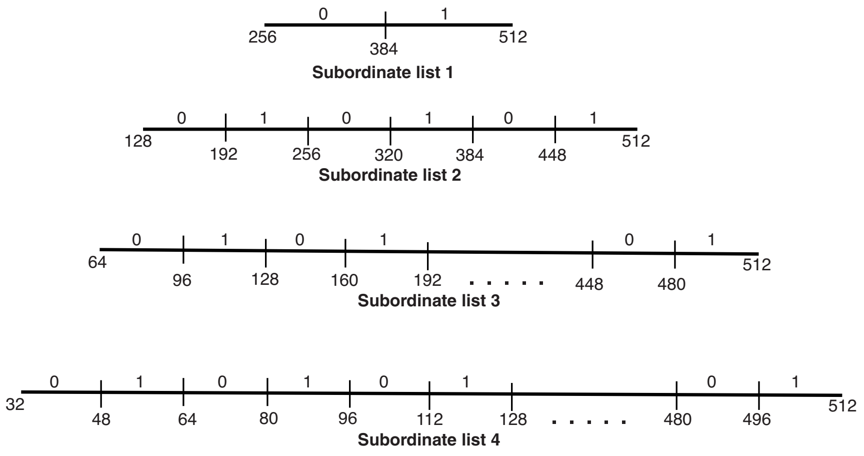


Figure 8.8 Quantization intervals in a successive approximation EZW coding of Example 8.2.

Table 8.11 Reconstructed pixels from the second subordinate pass

164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164
164	164	164	164	164	164	164	164

Table 8.12 Third dominant pass: $T_4 = 64$

Position	Coef. Value	Symbol	Quantized Value
HL3	69	POS	96
LH3	20	ZTR	0
HH3	-11	ZTR	0
HL2	31	ZTR	0
HL2	-12	ZTR	0
HL2	71	POS	96
HL2	82	POS	96

Table 8.13 Third Subordinate list: $T_5 = 32$

Coef. Value	Code	Quantized Value
300	01010101	304
82	0	80
71	0	80
69	0	80

Table 8.14 Reconstructed pixels from the third subordinate pass

176	176	176	176	176	176	176	176
176	176	176	176	176	176	176	176
176	176	176	176	176	176	176	176
176	176	176	176	176	176	176	176
176	176	176	176	176	176	176	176
176	176	176	176	176	176	176	176
136	136	136	136	136	136	136	136
136	136	136	136	136	136	136	136

for the fourth dominant and subordinate passes are shown in Tables 8.15–8.17. This process can be repeated until the bit budget is reached or until the threshold value becomes 1.

8.4 JPEG2000

In Chapter 7, we learnt the Joint Photographic Experts Group (JPEG) algorithm for compressing still images based on DCT. In 2000, ISO standardized another compression algorithm based on DWT and it became the JPEG2000 standard [14–17]. As with JPEG, JPEG2000 is a recommendation that (1) specifies the decoding processes for decompressing compressed image data to reconstruct the image data, (2) specifies a codestream syntax containing information for interpreting the compressed image data, (3) specifies a file format, (4) provides guidance on the encoding processes for

Table 8.15 Fourth dominant pass: $T_4 = 32$

Position	Coef. Value	Symbol	Quantized Value
LH3	20	ZTR	0
HH3	−11	IZ	0
HL2	31	ZTR	0
HL2	−12	ZTR	0
HH2	−2	ZTR	0
HH2	38	POS	48
HH2	1	ZTR	0
HH2	−13	ZTR	0

Table 8.16 Fourth subordinate list: $T_5 = 16$

Coef. Value	Code	Quantized Value
300	01010101010101010	308
82	0101	88
71	010	72
69	010	72
38	0	40

Table 8.17 Reconstructed pixels from the fourth subordinate pass

176	176	176	176	186	186	166	166
176	176	176	176	186	186	166	166
176	176	176	176	166	166	186	186
176	176	176	176	166	166	186	186
176	176	176	176	180	180	180	180
176	176	176	176	180	180	180	180
140	140	140	140	136	136	136	136
140	140	140	140	136	136	136	136

converting source image data to compressed image data, and (5) provides guidance on how to implement these processes in practice. We are concerned only with the basic compression algorithm used in JPEG2000 standard. The standard specifies both lossless and lossy compression algorithms.

JPEG2000 uses 2D DWT as the compression vehicle. The overall encoding procedure is shown in Figure 8.9 and is described as follows.

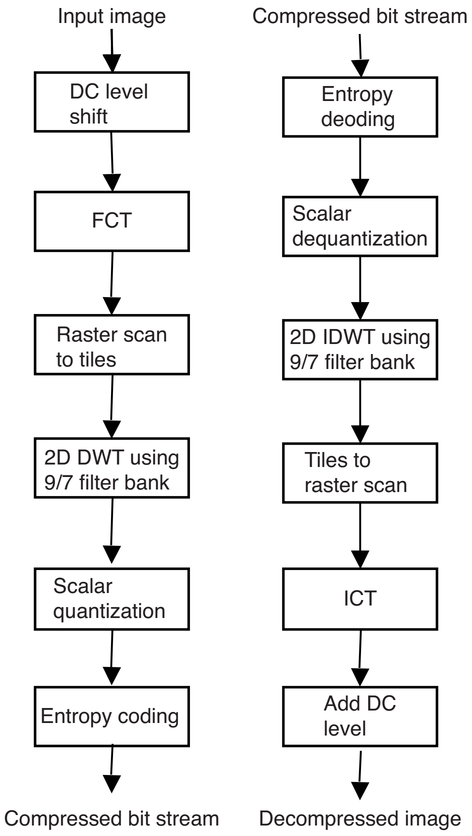


Figure 8.9 Overall encoding procedure adopted by JPEG2000. Left: encoder; right: decoder.

8.4.1 DC Level-Shifting

The input image components are disparity-compensated (DC) level shifted by subtracting a fixed integer as defined by

$$I'_k[m,n] = I_k[m,n] - 2^{\text{size}_k - 1} \tag{8.6}$$

where size_k is the maximum possible value of the kth input component. For instance, if an input component has a pixel depth of 8 bits, then $2^{\text{size}_k-1} = 2^{8-1} = 128$. The DC level is added back to each pixel of each component after the ICT, as given by

$$I_k[m,n] = I'_k[m,n] + 2^{\text{size}_k - 1} \tag{8.7}$$

8.4.2 Forward Component Transformation

The level-shifted image components, for example, red, green, and blue, are forward transformed into equivalent components in another set of coordinates. This is important from the point of view of achieving higher compression. The *R*, *G*, and *B* components typically have a high correlation. Therefore, it is rather inefficient to compress the components in the RGB coordinates independently as this might introduce false colors. However, if they are transformed into a set of equivalent orthogonal coordinates, it is possible to compress the components in the new coordinates independent of each other and it results in higher compression efficiency. For irreversible (lossy) transform, the forward component transformation (FCT) is expressed as

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{bmatrix} \begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} \tag{8.8}$$

For reversible (lossless) transform, the FCT is described by the following equation:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} \dfrac{I_0 + 2I_1 + I_2}{4} \\ I_2 - I_1 \\ I_0 - I_1 \end{bmatrix} \tag{8.9}$$

In equations (8.8) and (8.9), I_0, I_1, and I_2 are the input components and Y_0, Y_1, and Y_2 are the coordinate transformed components. Note that if the input components in equation (8.8) are *R*, *G*, and *B*, respectively, then the transformed output components approximately correspond to *Y*, *Cb*, and *Cr*, respectively.

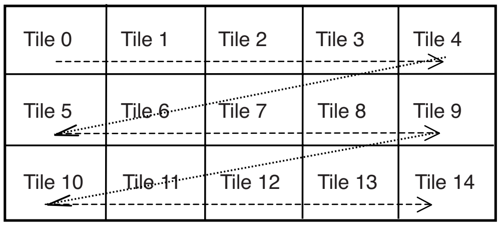


Figure 8.10 Division of image components into tiles: tiles in a sequential raster scanning order.

8.4.3 Inverse Component Transformation

The ICTs for the irreversible and reversible cases are expressed by equations (8.10) and (8.11), respectively:

$$\begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.34413 & -0.71414 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} \tag{8.10}$$

$$\begin{bmatrix} I_1 \\ I_0 \\ I_2 \end{bmatrix} = \begin{bmatrix} Y_0 - \left\lfloor \dfrac{Y_1 + Y_2}{4} \right\rfloor \\ Y_2 + I_1 \\ Y_1 + I_1 \end{bmatrix} \tag{8.11}$$

8.4.4 Division of Components into Tiles

The components are divided into *tiles*, which are the basic blocks similar to the *macroblocks* in JPEG. Tiles are rectangular subblocks of an image component and are numbered sequentially in a raster scanning order as shown in Figure 8.10. Note that all tiles in a component are of the same size. As in JPEG, the chroma components are subsampled to 4:2:2 or 4:2:0 sampling format before tiling.

8.4.5 Tile-Component Division into DWT

After tiling, each tile of each component is decomposed into an L-level 2D DWT using Daubechies's $\frac{9}{7}$ FIR filter bank for irreversible transform. The impulse responses of the analysis and synthesis filters of Daubechies's $\frac{9}{7}$ FIR filter bank are listed in Tables 8.18 and 8.19, respectively. The corresponding frequency

Table 8.18 Daubechies $\frac{9}{7}$ analysis lowpass and highpass filters

n	LPF	HPF
0	0.6029490182	1.1150870525
±1	0.2668641184	−0.5912717631
±2	−0.0782232665	−0.0575435263
±3	−0.0168641184	0.0912717631
±4	0.0267487574	

Table 8.19 Daubechies $\frac{9}{7}$ synthesis lowpass and highpass filters

n	LPF	HPF
0	1.1150870525	0.6029490182
±1	0.5912717631	−0.2668641184
±2	−0.0575435263	−0.0782232665
±3	−0.0912717631	0.0168641184
±4		0.0267487574

responses are shown in Figures 8.11a,b. It is easily verified, as per Section 4.5.2, that the Daubechies's $\frac{9}{7}$ FIR filters form a biorthogonal wavelet system. In this case, the analysis lowpass filter weights sum up to 1 and the corresponding synthesis filter weights sum up to 2, instead of $\sqrt{2}$.

For reversible compression, the analysis and synthesis filters used are called Le Gall's $\frac{5}{3}$ filters. The impulse responses of Le Gall's $\frac{5}{3}$ filters are listed in Table 8.20. Figures 8.12a,b show the corresponding magnitude and phase responses, respectively. The Le Gall's filter bank also forms a biorthogonal wavelet system.

8.4.6 Scalar Quantization of the DWT Coefficients

In JPEG2000 standard, the step size $Q(b)$ of the uniform quantizer for subband "b" is calculated from

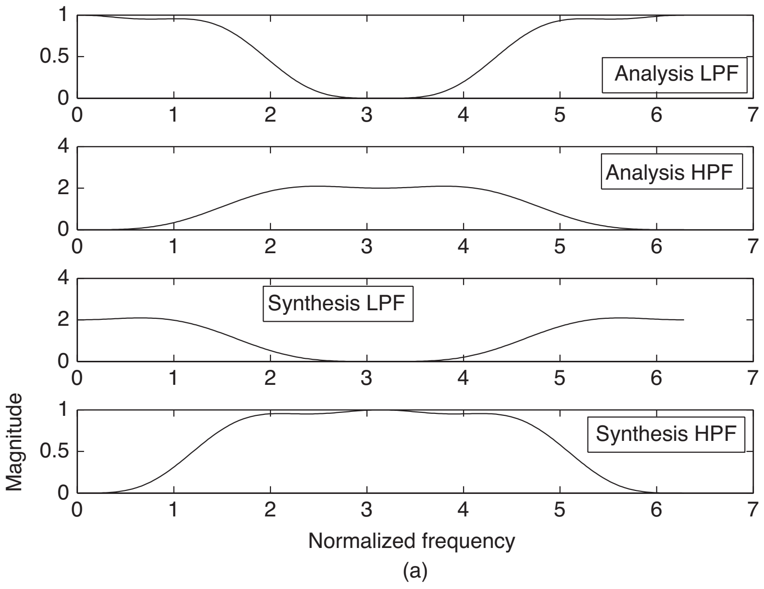
$$Q(b) = 2^{R_b+g_b-\varepsilon_b}\left(1+\frac{\mu_b}{2^{11}}\right) \tag{8.12}$$

where R_b is the nominal dynamic range of subband "b," g_b is the gain of the subband, and ε_b and μ_b are the exponent and mantissa of the coefficients that are either explicitly signaled in the bit stream for each subband or implicitly signaled only for the LL band. The gain is unity in the lowpass direction and 2 in the highpass direction. Thus, the HL and LH bands at each level have a gain of 2, while the HH band has a gain of 4. With the step size defined in equation (8.12), a coefficient in a given subband is quantized using

$$X_{qb}(u,v) = \text{sign}\left(X_b(u,v)\right)\left\lfloor \frac{|X_b(u,v)|}{Q(b)} \right\rfloor \tag{8.13}$$

Table 8.20 Le Gall's $\frac{5}{3}$ analysis/synthesis filter impulse responses

	Analysis Filter Bank		Synthesis Filter Bank	
Sample Index	Lowpass	Highpass	Lowpass	Highpass
0	6/8	1	1	6/8
±1	2/8	−1/2	1/2	−2/8
±2	−1/8			−1/8



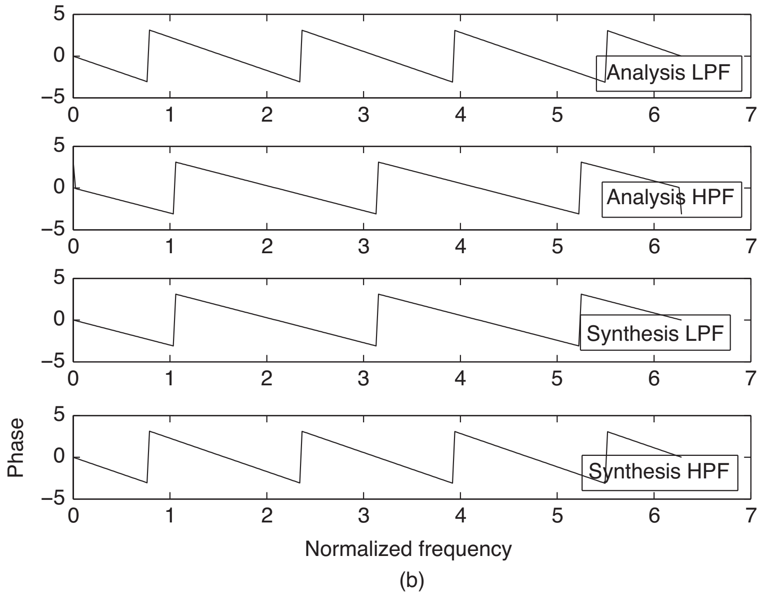
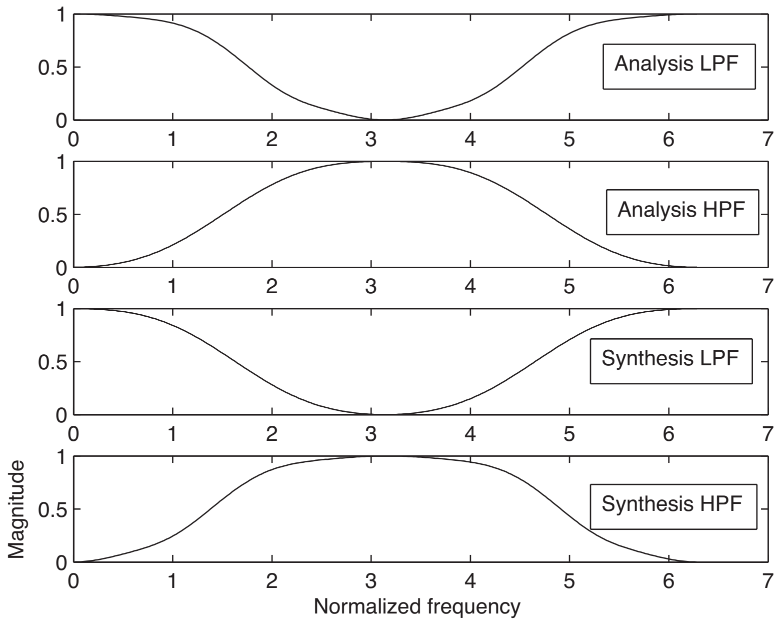
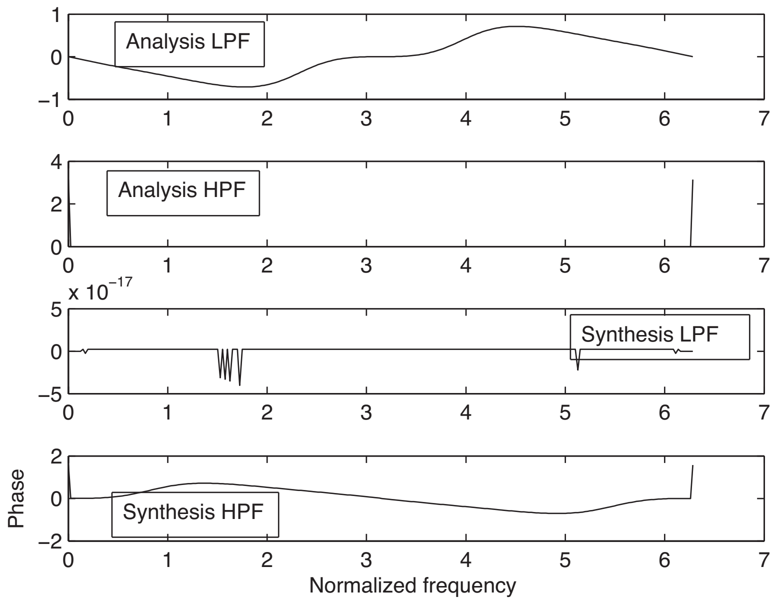


Figure 8.11 Frequency response of the $\frac{9}{7}$ Daubechies filter bank for irreversible JPEG2000 compression: (a) magnitude of the frequency response of the analysis lowpass, highpass, and synthesis lowpass, and highpass FIR filters whose impulse responses are listed in Tables 8.18 and 8.19, and (b) phase response of the analysis and synthesis filters.



(a)



(b)

Figure 8.12 Frequency response of the $\frac{5}{3}$ Le Gall's filter bank for reversible JPEG2000 compression: (a) magnitude of the frequency response of the analysis lowpass, highpass, and synthesis lowpass, and highpass FIR filters whose impulse responses are listed in Tables 8.18 and 8.19, and (b) phase response of the analysis and synthesis filters.

8.4.7 Frequency Weighting

In order to improve the compression efficiency without sacrificing visual quality, JPEG2000 recommends the use of frequency weighting of the quantization step sizes of the uniform quantizers [18]. These frequency weightings are applied to the different subbands of the luma component and are based on the viewing distances. Table 8.21 lists the JPEG2000 recommended frequency weightings for the various DWT coefficients of the luma component. In order to incorporate the frequency weighting, one has to multiply the quantization step size by the appropriate weightings from Table 8.21.

We now have all the ingredients for the compression of a color image using JPEG2000 standard. Let us illustrate the procedure by an example.

Example 8.3 We want to compress an RGB image that has a pixel resolution of 8 bpp using JPEG2000 compression scheme. As this example deals with lossy compression, we will have to use Daubechies's $\frac{9}{7}$ filter bank. Let the number of levels of 2D DWT be 3. Thus, we have the bands numbered 3LL, 3HL, 3LH, 3HH, 2HL, 2LH, 2HH, 1HL, 1LH, and 1HH. Let the sampling format for the chroma components be 4:2:2. The chroma components are also decomposed to three levels using the same Daubechies's $\frac{9}{7}$ filter bank. Even though JPEG2000 allows tiling of the image components, we will use a single tile of the size of the original image. Thus, we have a single tile for each of the luma and chroma components.

First, the image components are level shifted using equation (8.6) and transformed to the new coordinate system via equation (8.8). The chroma components are converted to the 4:2:2 sampling format. We then perform the 2D DWT of the components to three levels using the analysis lowpass and highpass filters of Table 8.18. The quantization step sizes of the uniform quantizers for all the subbands are determined using equation (8.12). After having computed the quantization step sizes, each coefficient in each subband at each level is quantized by first modifying the quantization steps by multiplying them by the appropriate frequency weightings and then quantizing them via equation (8.13).

The quantized DWT coefficients are dequantized by multiplying the quantized coefficients by the modified quantization step sizes. The dequantized DWT coefficients are inverse 2D wavelet transformed using the synthesis lowpass and highpass filter bank of Table 8.19. The inverse wavelet transformed chroma components are upsampled to the full resolution and then the three reconstructed components are inverse component transformed using equation (8.10) and level shifted back to obtain the reconstructed RGB image.

Figure 8.13 is the decompressed bird's image using the JPEG2000 compression scheme. The bit rate corresponds to 0.64 bpp and may be less if entropy coding is used. The SNR for the luma is 19.58 dB, while it is 19.78 and 20.25 dB for the chroma components. Note that this is not the peak SNR, which, for example, is 34.17, 40.92, and 40.91 dB, respectively, for the luma and chroma components. As can be seen from Figure 8.13, the visual quality is very good at this low bit rate with no

Table 8.21 JPEG2000 recommendation of frequency weightings

	Viewing Distance (pixels)			Viewing Distance (pixels)			Viewing Distance (pixels)		
Level	1000			2000			4000		
	HL	LH	HH	HL	LH	HH	HL	LH	HH
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	0.731668
3	1	1	1	1	1	0.727203	0.564344	0.564344	0.285968
4	1	1	0.727172	0.560841	0.560841	0.284193	0.179609	0.179609	0.043903
5	0.560805	0.560805	0.284173	0.178494	0.178494	0.043631	0.014774	0.014774	0.000573

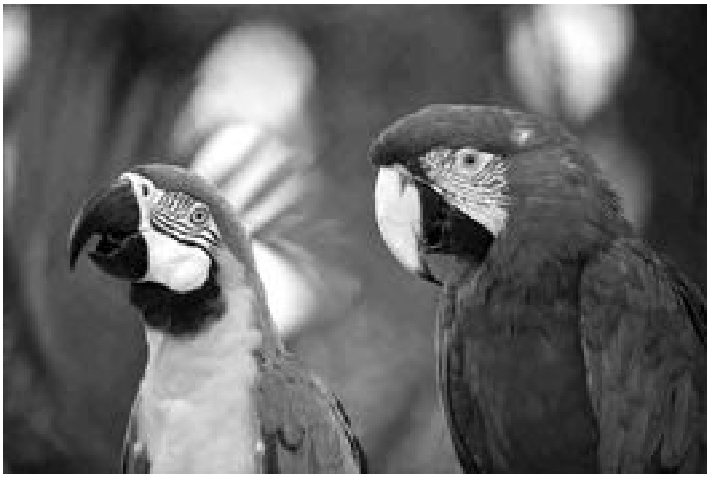


Figure 8.13 Example of irreversible compression of bird's image using JPEG2000 compression scheme. Scalar quantization is used and no entropy coding is used. The average bit rate is 0.64 bpp. The SNR for the *Y*, *Cb*, and *Cr* components are, respectively, 19.58, 19.78, and 20.25 dB. The chroma sampling format used is 4:2:2.

Table 8.22 Bit rate and SNR for the images using JPEG2000 compression scheme

Image	Bit (bpp)	*Y* (dB)	*Cb* (dB)	*Cr* (dB)
Cameraman	0.48	15.95	NA	NA
Birds	0.64	19.58	19.78	20.25
Aerial	0.52	12.12	NA	NA
Airplane	0.53	22.21	14.30	9.62
Masuda	0.51	24.08	13.95	17.09
Peppers	0.46	20.84	19.65	19.00
Hat woman	0.58	23.29	14.63	17.88
Yacht	0.52	18.10	14.42	14.68
Lighthouse	0.59	13.18	14.68	15.35

blocking artifact unlike the JPEG standard that uses DCT. Table 8.22 shows the bit rate in bpp and the SNR in dB for various images compressed using this example. As can be seen from the table, the overall average bit rate is in the neighborhood of 0.5 bpp. All reconstructed images have good visual quality with no blocking artifact. The MATLAB code for this example is listed below.

```
% Example8_3.m
% Implements JPEG2000 wavelet transform-based compression.
% Implements only the scalar quantization normative of the
% standard. Implements up to NL levels of 2D DWT using
% Daubechies' 9/7 filter bank.
% Implements 4:2:0 and 4:2:2 sampling formats specified
% by the user.
% Uses the frequency weightings recommended by JPEG2000 standard
% in the uniform scalar quantization of the DWT coefficients.
%
clear
NL = 3; % number of levels of 2D DWT
SamplingFormat = '4:2:2';
%A = imread('cameraman.tif');
A = imread('lighthouse.ras');
% make sure that the image size is divisible by NL
[X,Y,Z] = size(A);
if mod(X,2^NL) ~=0
    Height = floor(X/(2^NL))*(2^NL);
else
    Height = X;
end
if mod(Y,2^NL) ~=0
    Width = floor(Y/(2^NL))*(2^NL);
else
    Width = Y;
end
Depth = Z;
clear X Y Z
% Do forward Irreversible Component Transformation (ICT)
% In JPEG2000 jargon, y0 corresponds to the Luma Y,
% y1 to Cb and y2 to Cr.
if Depth == 1
    y0 = double(A(1:Height,1:Width)) - 128; % DC level shift
else
    A1 = double(A(1:Height,1:Width,:))-128; % DC level shift
    y0 = 0.299*A1(:,:,1) + 0.587*A1(:,:,2) + 0.144*A1(:,:,3);
    y1 = -0.16875*A1(:,:,1) - 0.33126*A1(:,:,2) + 0.5*A1(:,:,3);
    y2 = 0.5*A1(:,:,1) - 0.41869*A1(:,:,2) - 0.08131*A1(:,:,3);
    clear A1
    switch SamplingFormat
        case '4:2:0'
            y1 = imresize(y1,[Height/2 Width/2],'cubic');
            y2 = imresize(y2,[Height/2 Width/2],'cubic');
        case '4:2:2'
            y1 = imresize(y1,[Height Width/2],'cubic');
            y2 = imresize(y2,[Height Width/2],'cubic');
    end
end
% Daubechies 9/7 Filters
% Note: The Daubechies 9/7 filter bank has 9 taps for its analysis
% Lowpass and synthesis highpass FIR filters;
% 7 taps for its analysis highpass and synthesis lowpass
% FIR filters. However, in order to obtain subbands that are
% exact multiples of 2^-n, n being a positive integer, the
% number of taps in the filters must be even. That is why you
% will notice the filter lengths to be 10 instead of 9 or 7.
% The extra tap values are zero.
%
% LO_D = analysis lowpass filter
% LO_R = synthesis lowpass filter
% HI_D = analysis highpass filter
% HI_R = synthesis highpass filter
LO_D = [0 0.0267487574 -0.0168641184 -0.0782232665 0.2668641184...
    0.6029490182 0.2668641184 -0.0782232665 -0.0168641184...
    0.0267487574];
HI_D = [0 0.0912717631 -0.0575435263 -0.5912717631...
    1.1150870525 -0.5912717631 -0.0575435263 0.0912717631 0 0];
%
LO_R = [0 -0.0912717631 -0.0575435263 0.5912717631...
    1.1150870525 0.5912717631 -0.0575435263 -0.0912717631 0 0];
HI_R = [0 0.0267487574 0.0168641184 -0.0782232665 -0.2668641184...
    0.6029490182 -0.2668641184 -0.0782232665 0.0168641184...
    0.0267487574];
%
% "dwt_Coef" cell array has NL x 4 cells.
% In each cell at level NL, the 1st component is the LL
% coefficients,
% 2nd component is HL coefficients, 3rd component LH and
% 4th component HH coefficients.
%
% Do an NL-level 2D DWT of the Y component
dwt_y0 = cell(NL,4);
for n = 1:NL
    if n == 1
        [dwt_y0{n,1},dwt_y0{n,2},dwt_y0{n,3},dwt_y0{n,4}] =...
            dwt2(y0,LO_D,HI_D,'mode','per');
    else
        [dwt_y0{n,1},dwt_y0{n,2},dwt_y0{n,3},dwt_y0{n,4}] =...
            dwt2(dwt_y0{n-1,1},LO_D,HI_D,'mode','per');
    end
end
% Quantize the Y DWT coefficients
[dwt_y0,y0Bits] = jpg2K_Quantize(dwt_y0,NL,1);
%
% Do an NL-level 2D IDWT of the Y component
% The cell array idwt_img has NL cells with each cell
% corresponding to
% the reconstructed LL image at that level.
idwt_y0 = cell(NL,1);
for n = NL:-1:1
    if n == 3
        idwt_y0{n} = idwt2(dwt_y0{n,1},dwt_y0{n,2},dwt_y0{n,3},...
            dwt_y0{n,4},LO_R,HI_R,'mode','per');
    else
        idwt_y0{n} = idwt2(idwt_y0{n+1},dwt_y0{n,2},dwt_y0{n,3},...
            dwt_y0{n,4},LO_R,HI_R,'mode','per');
    end
```

```
end
SNRy0 = 20*log10(std2(y0)/std2(y0-idwt_y0{1}));
if Depth > 1
    % Do an NL-level 2D DWT of the Cb and Cr components
    dwt_y1 = cell(NL,4);
    dwt_y2 = cell(NL,4);
    for n = 1:NL
        if n == 1
            [dwt_y1{n,1},dwt_y1{n,2},dwt_y1{n,3},dwt_y1{n,4}] =...
                dwt2(y1,LO_D,HI_D,'mode','per');
            [dwt_y2{n,1},dwt_y2{n,2},dwt_y2{n,3},dwt_y2{n,4}] =...
                dwt2(y2,LO_D,HI_D,'mode','per');
        else
            [dwt_y1{n,1},dwt_y1{n,2},dwt_y1{n,3},dwt_y1{n,4}] =...
                dwt2(dwt_y1{n-1,1},LO_D,HI_D,'mode','per');
            [dwt_y2{n,1},dwt_y2{n,2},dwt_y2{n,3},dwt_y2{n,4}] =...
                dwt2(dwt_y2{n-1,1},LO_D,HI_D,'mode','per');
        end
    end
    %
    % Quantize the Cb and Cr DWT coefficients
    [dwt_y1,y1Bits] = jpg2K_Quantize(dwt_y1,NL,2);
    [dwt_y2,y2Bits] = jpg2K_Quantize(dwt_y2,NL,3);
    %
    % % Do an NL-level 2D IDWT of the Cb and Cr components
    idwt_y1 = cell(NL,1);
    idwt_y2 = cell(NL,1);
    %
    for n = NL:-1:1
        if n == 3
            idwt_y1{n} = idwt2(dwt_y1{n,1},dwt_y1{n,2},dwt_y1{n,3},...
                dwt_y1{n,4},LO_R,HI_R,'mode','per');
            idwt_y2{n} = idwt2(dwt_y2{n,1},dwt_y2{n,2},dwt_y2{n,3},...
                dwt_y2{n,4},LO_R,HI_R,'mode','per');
        else
            idwt_y1{n} = idwt2(idwt_y1{n+1},dwt_y1{n,2},dwt_y1{n,3},...
                dwt_y1{n,4},LO_R,HI_R,'mode','per');
            idwt_y2{n} = idwt2(idwt_y2{n+1},dwt_y2{n,2},dwt_y2{n,3},...
                dwt_y2{n,4},LO_R,HI_R,'mode','per');
        end
    end
    % Compute the SNR of Cb and Cr components due to quantization
    SNRy1 = 20*log10(std2(y1)/std2(y1-idwt_y1{1}));
    SNRy2 = 20*log10(std2(y2)/std2(y2-idwt_y2{1}));
    % Upsample Cb & Cr components to original full size
    y1 = imresize(idwt_y1{1},[Height Width],'cubic');
    y2 = imresize(idwt_y2{1},[Height Width],'cubic');
end
% Do inverse ICT & level shift
if Depth == 1
    Ahat = idwt_y0{1} + 128;
    figure,imshow(uint8(Ahat))
    sprintf('Average bit rate = %5.2f bpp\n',y0Bits/(Height*Width))
    sprintf('SNR(Y) = %4.2f dB\n',SNRy0)
else
    Ahat = zeros(Height,Width,Depth);
    Ahat(:,:,1) = idwt_y0{1} + 1.402*y2 + 128;
    Ahat(:,:,2) = idwt_y0{1} - 0.34413*y1 - 0.71414*y2 + 128;
    Ahat(:,:,3) = idwt_y0{1} + 1.772*y1 + 128;
    figure,imshow(uint8(Ahat))
    sprintf('Average bit rate = %5.2f bpp\n',...
        (y0Bits+y1Bits+y2Bits)/(Height*Width))
    SNRy0 = 20*log10(std2(y0)/std2(y0-idwt_y0{1}));
    sprintf('SNR(Y) %4.2fdB\tSNR(Cb) = %4.2f dB\tSNR(Cr) =
                                       %4.2fdB\n',...
        SNRy0,SNRy1,SNRy2)
end
```

```
function [y,TotalBits] = jpg2K_Quantize(x,NL,CompNo)
% [y,TotalBits] = jpg2K_Quantize(x,NL,CompNo)
% Quantizes using JPEG2000 scalar quantization rule
% Input:
%   x = cell array containing the 2D DWT coefficients
%   NL = number of levels of octave-band 2D DWT
%   CompNo = Component number, which can be 1, 2, or 3
%   1 for Y, 2 for y1 (Cb) and 3 for y2 (Cr)
% Output:
%   y = cell array of the same size as x containing
%       quantized DWT coefficients
%   TotalBits = Total bits used up by the coefficients
%
% The input cell array x{1,2:4} contains the 1HL, 1LH, and 1HH
% DWT coefficients at level 1, etc., and x{NL,1:4}
% contains the LL, HL, LH, and HH coefficients at level NL.
%
% The quantization step sizes are usually specified in the
% headers of the bit stream. However, here, we simply
% assign the number of bits for the uniform quantizers
% and compute the appropriate quantization steps.
%
y = cell(NL,4);% output cell array
maxVal = zeros(4,1);% maximum coefficient value
%minVal = zeros(4,1);
Qstep = zeros(NL,4);% array to store the quantization steps
Qbits = zeros(NL,4);% array to store the quantizer bits
switch CompNo
    case 1
        Qstep(1,2:3)=(2^(8-0))*(1+8/(2^11));
        Qstep(1,4)=(2^(8-0))*(1+8/(2^11));
        Qstep(2,2:3)=(2^(8-3))*(1+8/(2^11));
        Qstep(2,4)=(2^(8-2))*(1+8/(2^11));
        Qstep(2,4) = Qstep(2,4)/0.731668; % uses CSF
        Qstep(3,2:3)=(2^(8-5))*(1+8/(2^11));
        Qstep(3,2:3) = Qstep(3,2:3)/0.564344; % uses CSF
        Qstep(3,4)=(2^(8-5))*(1+8/(2^11));
        Qstep(3,4)= Qstep(3,4)/0.285968; % uses CSF
        Qstep(3,1)=(2^(8-5))*(1+8/(2^11));
    case {2,3}
        Qstep(1,2:3)=(2^(8-0))*(1+8/(2^11));
        Qstep(1,4)=(2^(8-1))*(1+8/(2^11));
        Qstep(2,2:3)=(2^(8-3))*(1+8/(2^11));
        Qstep(2,4)=(2^(8-2))*(1+8/(2^11));
        Qstep(3,2:3)=(2^(8-4))*(1+8/(2^11));
        Qstep(3,4)=(2^(8-3))*(1+8/(2^11));
        Qstep(3,1)=(2^(8-5))*(1+8/(2^11));
end
TotalBits = 0;
for n = 1:NL
    if n < NL
        m1 = 2;
    else
        m1 = 1;
    end
    for m = m1:4
        maxVal(m) = max(x{n,m}(:));
        t = round(log2(maxVal(m)/Qstep(n,m)));
        if t < 0
            Qbits(n,m)= 0;
        else
            Qbits(n,m)= t;
        end
        TotalBits = TotalBits+Qbits(n,m)*size(x{n,m},1)
                      *size(x{n,m},2);
    end
end
%
for n = 1:NL
    if n < NL
        m1 = 2;
    else
        m1 = 1;
    end
    for m = m1:4
        s = sign(x{n,m});
        q2 = Qstep(n,m)/2;
        y{n,m} = s .* round(abs(x{n,m})/q2)*q2;
    end
end
```

8.5 DIGITAL CINEMA

By *digital cinema* we mean the delivery of movies in digital data stored in a hard drive and projection via digital projectors. Thus, in digital cinema, the movie is in digital data format end-to-end. The requirements of movies in digital data format are much more stringent than those of the TV. Table 8.23 is a list of parameters of the pictures in a digital cinema. It lists the requirements of movies for production/mastering and projection. Generally, images are generated and stored at a very high quality during mastering the movie and then the quality is reduced during distribution. As can be seen from the table, image resolution, pixel depth, and chroma sampling are much higher than the TV counterparts. Moreover, progressive scanning rather than interlaced scanning is used in digital cinema, which minimizes motion artifacts.

8.5.1 Digital Movie Compression

Unlike motion pictures for standard definition television, digital cinema source generates a data rate of 182.25 Mb/s, assuming a frame size of 1920 $\times$ 1080 pixels per component, a pixel depth of 10 bits/component, and a frame rate of 24 fps (frames per second). Note that we have used the fact that an Mb equals 1,024,000 bytes. As this source data rate is prohibitive for most real-time hardware system to handle, image compression is warranted. Even though a reversible compression would be welcome by the movie industries, it can only compress the source data approximately by a factor of 2, which is still not sufficient. Therefore, lossy compression is required. At about 60 Mb/s rate, digital projection of movies would be quite acceptable. Then the required compression ratio is found to be about 25:1. A compression ratio of 25:1 may be easily achieved by DCT-based MPEG-2 compression scheme. However, there are two factors that immediately disqualify MPEG-based scheme. First, DCT-based compression scheme introduces blocking artifacts. Second, motion artifacts are introduced by MPEG-2 scheme because it uses *interframe* compression. Both artifacts are objectionable to the movie industry. Therefore, we are left with the choice of *intraframe* compression only. Because JPEG2000 uses DWT as the compression vehicle and since there is no blocking artifact introduced by the wavelet-domain compression, the Society for Motion Pictures & Television Engineering (SMPTE) has adopted intraframe compression of movie pictures using

Table 8.23 Image resolution, pixel depth, and chroma sampling in digital cinema

Parameter	Production	Projection
Image size (pixels)	1920 $\times$ 1080 to 4096 $\times$ 3112	1920 $\times$ 1080 to 4096 $\times$ 3112
Color space	RGB or YCbCr	RGB or YCbCr
Bit depth	12–16 bits per component	10 bits per component
Chroma sampling format	4:4:4	4:2:2
Frame rate	24 fps	24 fps

JPEG2000 compression scheme as the recommended compression standard for digital cinema. As this chapter is on wavelet-domain compression, the topic of digital cinema is briefly introduced here. Detailed information may be obtained in one of the references cited in the chapter [19].

8.6 SUMMARY

In this chapter, we have described still image compression in the wavelet domain. As with DCT, the DWT also has an excellent energy compaction property, which is the basis for achieving image compression. Unlike the 2D DCT, the 2D DWT decomposes an image into approximation and detail coefficients. The approximation coefficients, as the name implies, are coarse approximation to the original image. The detail coefficients with horizontal, vertical, and diagonal orientations, add refinements to the approximation coefficients. Typically, the 2D DWT is repeatedly applied to the approximation coefficients to obtain a multilevel or multiscale DWT of an image. This is known as the *octave-band* DWT. Theoretically, one can iterate the 2D DWT on the approximation coefficients until left with a single coefficient in each of the LL, HL, LH, and HH bands.

In order to achieve compression in the wavelet domain, the approximation and detail coefficients have to be quantized using either uniform or Lloyd–Max quantizer. Since energy is conserved in an orthogonal wavelet transform, the total mean square errors (MSE) due to quantization equals the sum of the MSE in the individual subbands. Therefore, one can use the optimal bit allocation rule equation (8.1) to determine the individual quantizer bits satisfying a given bit budget. Example 8.1 illustrates the calculation of optimal quantizer bits for a three-level 2D DWT of an actual image, as well as the quantization and dequantization of the DWT coefficients.

In entropy coding of the quantized DWT coefficients, one can exploit certain unique features of the DWT. The so-called EZW coding is a very clever scheme that exploits the tree of zero valued coefficients to code zero runs. EZW is also a progressive coding scheme because it allows the reconstruction of an image beginning with coarse quality and progressively improving to a higher quality. Thus, EZW embeds the symbols of the scanned DWT coefficient positions and the successive approximations of the significant coefficients in a progressive fashion so that the encoding (or decoding) can stop at a given bit rate. Due to its efficiency, EZW is a part of the JPEG2000 compression standard.

As this chapter is concerned with image compression using 2D DWT, the compression method of the JPEG2000 standard has been described with an example to show how the visual quality, bit rate, and SNR are related to a given image. MATLAB codes for the examples are also listed. In developing the codes, care has been taken not to use non-MATLAB built-in functions as much as possible. This will enable the readers to easily modify the codes to suit their new applications. Finally, a brief description of digital cinema requirements and the compression vehicle used to compress digital movies has been given.

After learning the basics of still image compression, it is time to peek into the domain of video compression. In the next chapter, we will introduce the basics of video compression, which is a precursor to the popular MPEG-2 standard.

REFERENCES

1. M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Trans. Image Process.*, 1 (2), 205–220, 1992.
2. J. Froment and S. Mallat, "Second generation compact image coding with wavelets," in C. K. Chui, ed., *Wavelets: A Tutorial in Theory and Applications*, Academic Press, New York, 1992.
3. R. A. DeVore, B. Jawerth, and B. J. Lucier, "Image compression through wavelet transform coding," *IEEE Trans. Inform.*, Special Issue on Wavelet Transforms and Multiresolution Signal Analysis, 38 (20), 719–746, 1992.
4. S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Patt. Recog. Mach. Intell.*, 11 (7), 674–693, 1989.
5. J. W. Woods, editor, *Subband Image Coding*, Kluwer Academic, Boston, MA, 1991.
6. P. J. Burt and E. H. Adelson, "The Laplacian pyramid as a compact image code," *IEEE Trans. Commun.*, 31 (4), 532–540, 1983.
7. M. Vetterli and J. Kovacevic, *Wavelets and Subband Coding*, Prentice Hall, Englewood Cliffs, NJ, 1995.
8. K. Ramachandran, A. Ortega, and M. Vetterli, "Bit allocation for dependent quantization with applications to multiresolution and MPEG video coders," *IEEE Trans. Image Process.*, 3 (5), 533–545, 1994.
9. P. H. Westerink, *Subband Coding of Images*, Ph.D. Thesis, Delft University of Technology, Delft, The Netherlands, 1989.
10. H. Gharavi and A. Tabatabai, "Subband coding of monochrome and color images," *IEEE Trans. Circ. Syst.*, 35 (2), 207–214, 1988.
11. A. S. Lewis and G. Knowles, "Image compression using 2-D wavelet transform," *IEEE Trans. Image Process.*, 1 (2), 244–250, 1992.
12. J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Process.*, Special Issue on Wavelets and Signal Processing, 41 (12), 3445–3462, 1993.
13. A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, 6 (3), 243–250, 1996.
14. JPEG, *Part I: Final Draft International Standard* (ISO/IEC FDIS15444–1), ISO/IEC JTC1/SC29/WG1 N1855, August 2000.
15. M. W. Marcellin, M. Gormish, A. Bilgin, and M. Boliek, "An overview of JPEG2000," Proc. Data Compression Conf., Snowbird, Utah, 2000.
16. A. Skodras, C. Christopuolos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Process. Mag.*, 36–58, 2001.

17. K. Varma and A. Bell, "JPEG2000—choices and tradeoff for encoders," *IEEE Signal Process. Mag.*, 70–75, 2004.
18. A. B. Watson et al., "Visibility of wavelet quantization noise," *IEEE Trans. Image Process.*, 6 (8), 1164–1175, 1997.
19. K. S. Thyagarajan, *Digital Image Processing with Application to Digital Cinema*, Elsevier, New York, 2006.

PROBLEMS

8.1. Design a linear phase lowpass FIR filter having a normalized cutoff frequency of 0.5, passband ripple of 0.2 dB and a minimum stopband attenuation of 40 dB. Determine its step response. Filter a real image with the designed filter using row column operation to assess the ringing effect. How is the ringing effect if you increased the filter length? Explain.

8.2. Compute the 2D DWT of a real intensity image of your choice up to three levels using the "db2" wavelet and calculate the optimal quantizer bits for the DWT coefficients to meet a bit budget of 0.5 bpp. Repeat the problem using the biorthogonal "bior1.5" wavelet and compare the bit assignment, if any.

8.3. Design PDF optimized nonuniform quantizers (Lloyd–Max) for the 2D DWT coefficients in Problem 8.2 and quantize and dequantize the intensity image. Repeat the procedure for uniform quantization. Compare the SNRs for the two cases.

8.4. Perform a three level 2D DWT of a real intensity image using "db2" wavelet and determine the optimal quantizer bits for a bit budget of 1 bpp. Use a DCT-based quantization for the LL band and uniform quantization for the detail coefficients. Dequantize the quantized coefficients, compute the 2D IDWT, and calculate the resulting SNR.

8.5. For the 8×8 pixels of an image with 8 bpp shown below, calculate the level shifted three-level 2D DWT using "bior3.9" wavelet. Perform the EZW coding up to three passes and reconstruct the pixels and determine the MSE between the original and reconstructed pixels.

186	191	187	188	187	181	155	144
179	185	185	180	186	190	178	154
164	176	174	162	171	177	179	169
157	171	173	159	168	176	179	184
179	178	182	175	175	174	173	174
176	174	177	176	172	170	170	164
154	148	153	155	149	146	143	140
132	134	135	132	132	129	118	111

9

BASICS OF VIDEO COMPRESSION

9.1 INTRODUCTION

The previous chapters described the basics of still image compression. They are based on removing the spatial or *intraframe* correlation. On the other hand, videos and movies are made up of a temporal sequence of frames and are projected at a proper rate (24 fps for movie and 30 fps for TV) to create the illusion of motion. This means that there exists a high correlation between adjacent temporal frames so that when projected at a proper rate, smooth motion is seen. Correlation between adjacent temporal frames is called *interframe* correlation. To illustrate this idea, we calculate the autocorrelation of two frames of a sequence at time instants t_1 and t_2, as defined by

$$R(t_1, t_2) = E\{f[m, n, t_1]\, f[m, n, t_2]\} \tag{9.1}$$

In equation (9.1), $E\{\mathbf{X}\}$ is the expected value of the random variable $\mathbf{X}$. If the image sequence is stationary, then the autocorrelation is a function only of the time difference and not of the individual time instants. Then, equation (9.1) can be written as

$$R(\tau) = E\{f[m, n, t_1]\, f[m, n, t_1 + \tau]\} \tag{9.2}$$

Still Image and Video Compression with MATLAB By K. S. Thyagarajan.

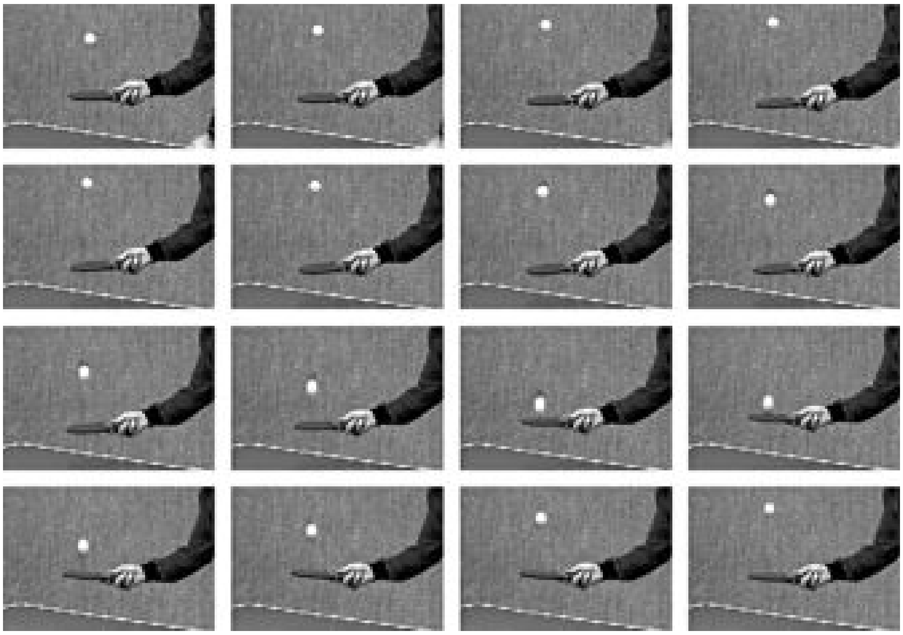


Figure 9.1 Collage of the first 16 frames in the Table Tennis sequence used to calculate the temporal correlation.

In practice, the joint probability density function at two different instants of time may not be available a priori, and therefore, we will use the following as the estimate of the true autocorrelation:

$$\hat{R}(\tau) = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} f[m, n, t_1] f[m, n, t_1 + \tau] \tag{9.3}$$

where M and N are the number of rows and columns of the images. Figure 9.1 is a collage of the first 16 frames of the Table Tennis sequence. The normalized temporal autocorrelation of this sequence is shown in Figure 9.2. As can be seen from the figure, the correlation is very high (approximately 0.98) even after about half a second interval, assuming a 30 fps rate. Similarly, Figure 9.3 shows the first 16 frames of the Trevor sequence and its temporal autocorrelation is shown in Figure 9.4. Again, we see a correlation as high as 0.98 after about half a second interval. As a third example, the Claire sequence (frames 10–25) and its temporal autocorrelation are shown in Figures 9.5 and 9.6, respectively. All three examples demonstrate the fact that there is a high correlation among the frames in the temporal dimension. Therefore, we can exploit both intraframe and interframe pixel correlation to achieve a higher degree of compression of video and movie.

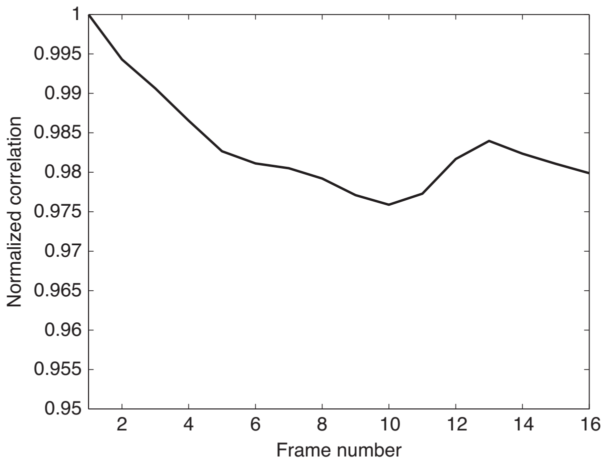


Figure 9.2 Normalized temporal autocorrelation of the Table Tennis sequence in Figure 9.1.

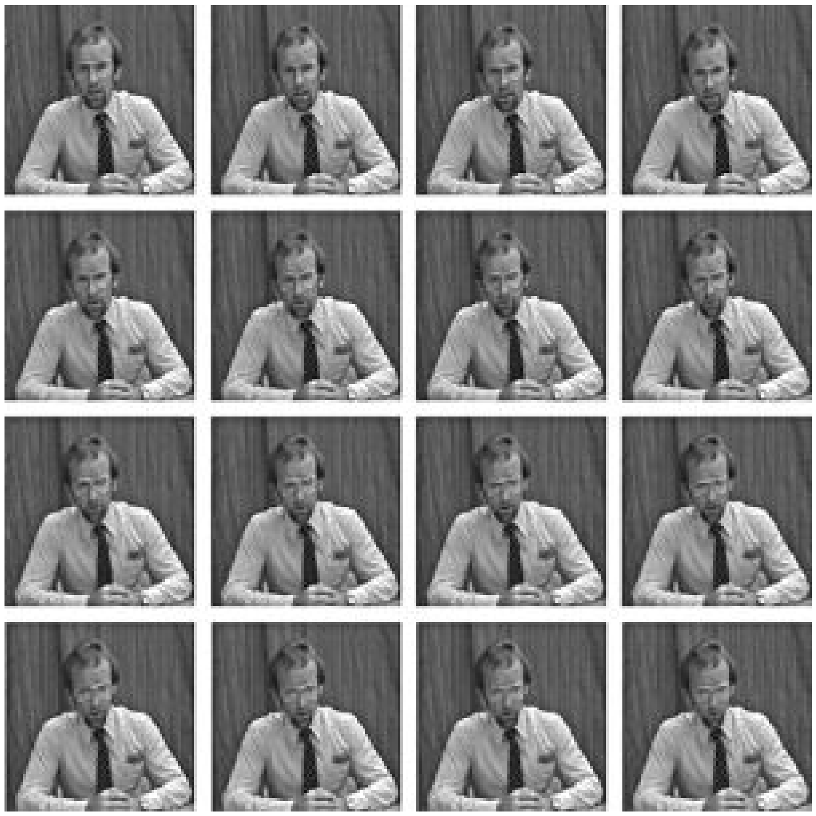


Figure 9.3 Collage of the first 16 frames in the Trevor sequence used to calculate the temporal correlation.

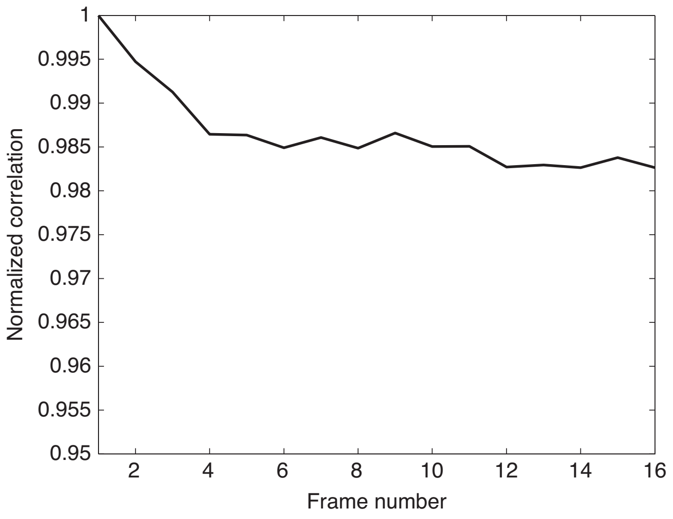


Figure 9.4 Normalized temporal autocorrelation of the Trevor sequence in Figure 9.3.

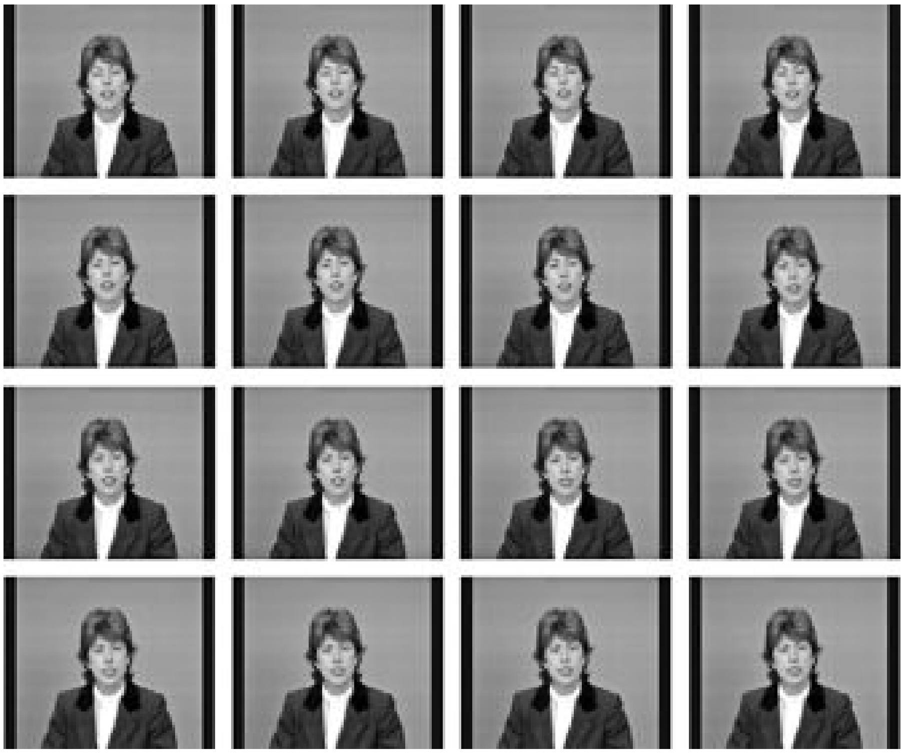


Figure 9.5 Collage of the first 16 frames in the Claire sequence used to calculate the temporal correlation.

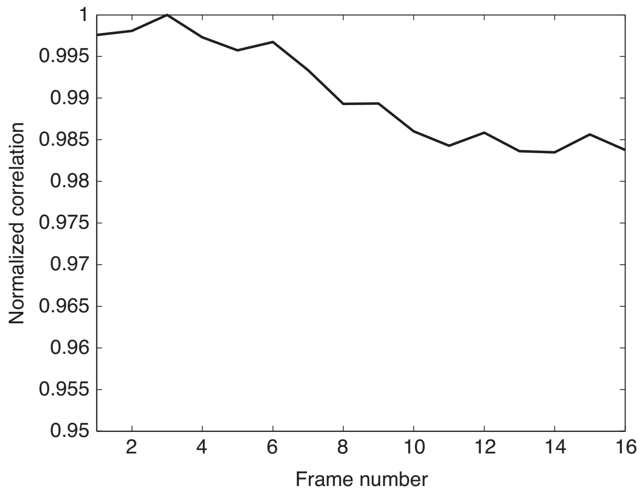


Figure 9.6 Normalized temporal autocorrelation of the Claire sequence in Figure 9.5.

9.2 VIDEO CODING

9.2.1 Simple Differencing

There are different approaches to removing the interframe correlation [1]. The simplest of all is frame differencing. Instead of transmitting each frame separately, the difference of the current and previous frames is transmitted or stored. Because the difference frame has reduced correlation, the range of pixel values in the differential frame is reduced and the probability density function of the differential frame has the characteristic Laplacian distribution. Therefore, this will require less number of bits to encode than the original frame and will result in compression. Figures 9.7a,b show frames 40 and 41 of the Table Tennis sequence. The simple differential frame is shown in Figure 9.7c and its histogram in Figure 9.7d. The resulting variance of the differential frame is 657, while the variances of the individual frames are 1255 and 1316. As a result of the reduction in the variance of the difference frame, it is possible to achieve compression. A point to remember is that in simple differencing there is no criterion or *cost function* involved, therefore, it is suboptimal. One can do better by minimizing a suitable objective or cost function. This is achieved using an optimal prediction rule.

9.2.2 Optimal Linear Prediction

Another approach is to use optimal prediction as we did for spatial prediction in Chapter 6. Thus, the temporal difference is described by subtracting the predicted frame from the actual frame:

$$e[m,n] = f[m,n,t_2] - \alpha\, f[m,n,t_1] \tag{9.4}$$

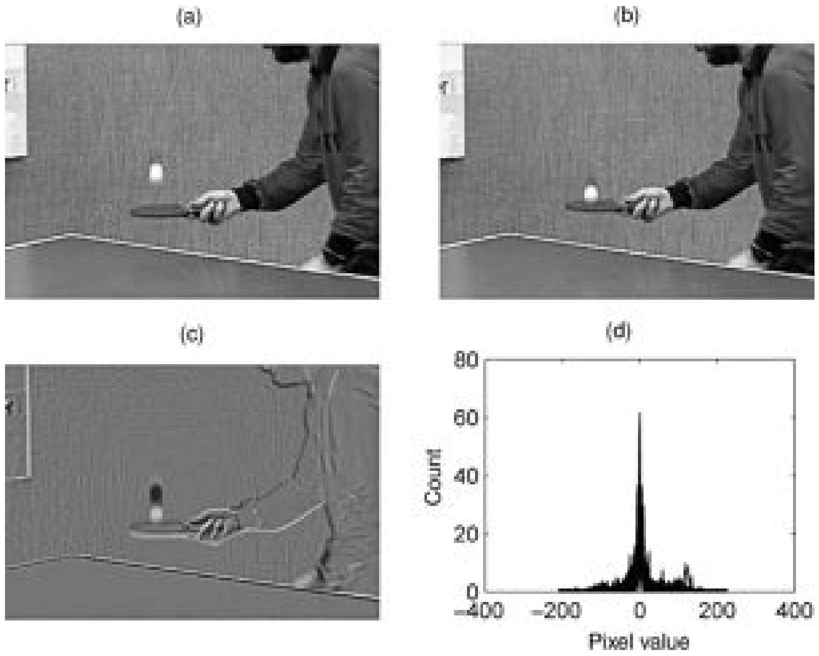


Figure 9.7 An example of simple frame differencing: (a) frame 40 of the Table Tennis sequence, (b) frame 41 of the Table Tennis sequence, (c) simple differential frame, and (d) histogram of the differential frame in c). The variance of the differential frame is 657, while the variances of the frames 40 and 41 are1255 and 1316, respectively.

The coefficient α is obtained by minimizing the mean square error (MSE) between the two adjacent temporal frames:

$$\alpha^* = \min E\left\{e^2[m,n]\right\} \tag{9.5}$$

By differentiating the MSE, $E\left\{e^2[m,n]\right\}$ with respect to α and setting the derivative to zero, we obtain

$$\begin{aligned} \frac{\partial E\left\{e^2\right\}}{\partial \alpha} &= \frac{\partial E\left\{f[m,n,t_2] - \alpha\, f[m,n,t_1]\right\}^2}{\partial \alpha} \\ &= 2E\left\{\left[\left(f[m,n,t_2] - \alpha\, f[m,n,t_1]\right)\right]\left(-f[m,n,t_1]\right)\right\} = 0 \end{aligned} \tag{9.6}$$

The optimal coefficient that minimizes the MSE between two temporally adjacent frames is then obtained from equation (9.6) and is given by

$$\alpha^* = \frac{E\left\{f[m,n,t_1]\, f[m,n,t_2]\right\}}{E\left\{f^2[m,n,t_1]\right\}} \tag{9.7}$$

In deriving the optimal predictor coefficient, we implicitly assumed that the two adjacent frames belong to a zero-mean random field. However, if the mean is not zero, then one can estimate it for each frame and subtract it from the frame and then use equation (9.7) to find the optimal coefficient value [2]. Further, if the joint probability distribution is not known a priori as is the typical situation, then we can calculate the optimal weight as given by

$$\hat{\alpha} = \frac{\sum_{m=1}^{M} \sum_{n=1}^{N} f[m,n,t_1]\, f[m,n,t_2]}{\sum_{m=1}^{M} \sum_{n=1}^{N} f^2[m,n,t_1]} \tag{9.8}$$

In equation (9.8), we omitted the normalization factor $1/(MN)$ because it is applied to both the numerator and the denominator. Observe that the numerator in equation (9.8) with the normalization constant $1/(MN)$ is the estimate of the temporal autocorrelation function in equation (9.3).

The same two frames 40 and 41 of the Table Tennis sequence, the corresponding differential frame, and its histogram are shown in Figure 9.8 using optimal

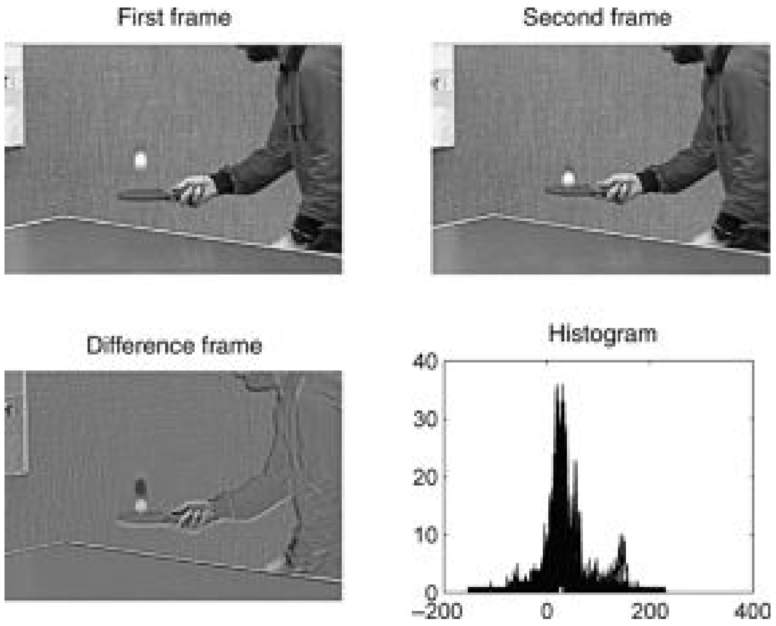


Figure 9.8 An example of optimal frame differencing: (a) frame 40 of the Table Tennis sequence, (b) frame 41 of the Table Tennis sequence, (c) differential frame using optimal prediction with a prediction coefficient value of 0.7615, and (d) histogram of the differential frame in (c). The variance of the differential frame is 585, while the variances of the frames 40 and 41 are 1255 and 1316, respectively. There is a reduction of 11% in the optimally predicted differential frame compared with that of simple differencing.

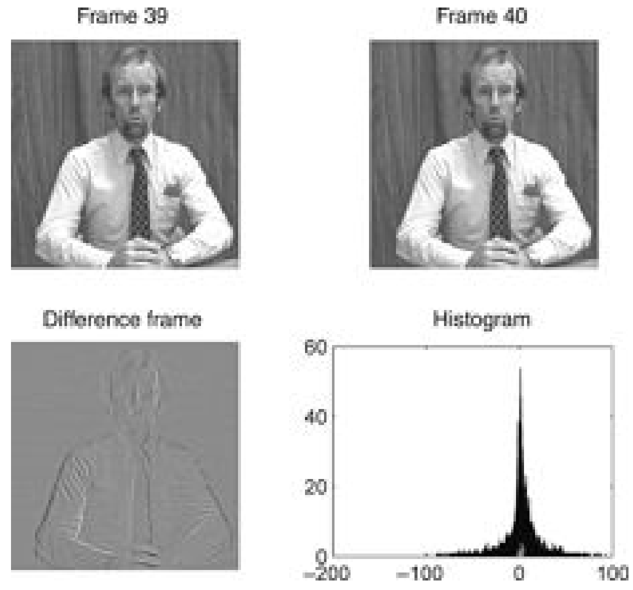


Figure 9.9 An example of optimal frame differencing: (a) frame 39 of the Trevor sequence, (b) frame 40 of the Trevor sequence, (c) differential frame using optimal prediction with a prediction coefficient value of 0.9783, and (d) histogram of the differential frame in (c). The variance of the differential frame is 68.5, while the variances of the frames 39 and 40 are 1799 and 1790, respectively.

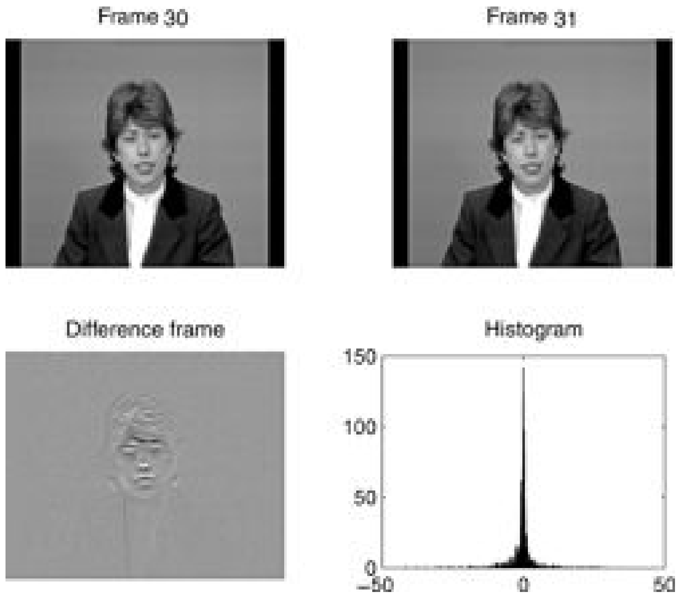


Figure 9.10 An example of optimal frame differencing: (a) frame 30 of the Claire sequence, (b) frame 31 of the Claire sequence, (c) differential frame using optimal prediction with a prediction coefficient value of 0.9981, and (d) histogram of the differential frame in (c). The variance of the differential frame is 3.71, while the variances of the frames 39 and 40 are 2879 and 2871, respectively.

prediction. The optimal predictor coefficient value is found to be 0.7615 and the resulting variance of the prediction error is 585. Although there is not a significant improvement in the histogram from that corresponding to the simple differencing, the variance of the prediction error is reduced by about 11%. However, the histogram is still wide indicating that there is a lot of residual value left. We must remember that the prediction error variance depends on the amount of motion between consecutive frames. The Table Tennis sequence in Figure 9.8 has a lot of motion involved. In comparison, frames 39 and 40 of the Trevor sequence have less motion, with the result that the error variance is 68.5 (see Figure 9.9). The Claire sequence (frames 30 and 31) has even less motion, and the optimal prediction error variance is only 3.71 (Figure 9.10). The corresponding optimal predictor coefficient is 0.9783. The MATLAB M file "TemporalPredict.m" to generate Figures 9.8–9.10 is listed below.

As we see from Figures 9.8–9.10, even optimal prediction is still not acceptable from the point of view of achieving a higher compression. Therefore, we must find alternative method to reduce the prediction error variance in order to achieve a much higher compression.

```
% TemporalPredict.m
% Computes the difference between two successive
% frames in an image sequence using either
% simple differencing or optimal linear prediction.
% Parameter "DiffType" can be "simple" for simple
  differencing or
% "optimal" for optimal prediction. Refer to Eq. (9-8)
% for the coefficient value.


 clear
inFile1 = 'tt040.ras';% table tennis sequence
inFile2 = 'tt041.ras';
%inFile1 = 'twy039.ras';% Trevor sequence
%inFile2 = 'twy040.ras';
%inFile1 = 'clairey030.yuv'; % Claire sequence
%inFile2 = 'clairey031.yuv';
A = imread(inFile1); % frame #1
B = imread(inFile2); % frame #2
[Height,Width] = size(A);
DiffType = 'optimal'; % options: "simple" and "optimal"
% simple temporal differencing
switch DiffType
    case 'simple'
        pf = double(B) - double(A);
    case 'optimal'
        A1 = double(A)-mean2(double(A)); % remove the
             mean of ref. frame
        B1 = double(B)-mean2(double(B)); % remove the
             mean of current frame
        V = sum(sum((double(A)-mean2(double(A))).*...
            (double(A)-mean2(double(A))))); % calculate
            the sum of
        % squares of the mean removed reference frame
        a = sum(sum(A1 .* B1))/V;
        pf = double(B) - a*double(A); % calculate the
             prediction error
end
figure, subplot(2,2,1), imshow(A),title('1st frame')
subplot(2,2,2), imshow(B),title('2nd frame')
sprintf(['MSE with ' DiffType ' temporal
    differencing = %3.3f'],std2(pf)*std2(pf))
subplot(2,2,3),imshow(pf,[]),title('difference frame')
[hMC,BinMC] = hist(pf,256);
subplot(2,2,4),plot(BinMC,hMC,'k'),title('histogram')
```

9.2.3 Motion-Compensated Prediction

As mentioned above, a reason for large variance in the differential frame is due to the motion of objects between temporally adjacent frames. If we can determine how much an object has moved from the previous frame to the current frame, then it is possible to align the object in the current frame with that in the previous frame and obtain the difference. This will result in zero error, at least in principle. In practice, this is not the case for several reasons [3]. An object's motion is estimated using variations in pixel values. These variations in pixel values might be due to lighting conditions, electronic noise, or apparent object motion. Further, since the 2D image is a perspective projection of the 3D object, there is difficulty in assessing whether the motion is due to rigid body or deformable body. A deformable body motion implies that different parts of the same body may undergo motion to different extent. Thus, one appreciates the degree of difficulty involved in estimating object motion between temporally adjacent frames of an image sequence.

In what follows, we will assume that the object motion is only translational, that is, left-and-right and up-and-down, and that the motion is due to rigid body. In order to determine the motion of an object in a frame, one has to first define the object boundaries in the current and reference frames. To make the task simpler, we will only use rectangular blocks of a constant size. The first task is to estimate the translational motion of a rectangular block of pixels between the current and previous or *reference* frames known as the *motion estimation*. This will yield a *motion vector* with components in the horizontal and vertical directions. The magnitude of the components will be in number of pixels. Once the motion vector is determined, the rectangular block in the current frame can be aligned with that in the reference frame and the corresponding differential pixels can be found. The process of aligning and differencing objects between successive frames is called *motion-compensated (MC) prediction* [4, 5]. Since we are dealing with images defined on rectangular grids, motion can only be estimated to an accuracy of a pixel. However, motion can be estimated to an accuracy of less than a pixel only approximately because it involves interpolating the values between pixels.

Motion Estimation We want to estimate the translational motion of a block of pixels in the current frame with respect to the previous frame. This can be accomplished by *phase-correlation* method or pel-recursive method, by *block-matching* method in the spatial domain [6], or by the use of *optical flow equation* (OFE).

Phase-Correlation Method The phase-correlation relies on the fact that the normalized cross-correlation function in the two-dimensional (2D) Fourier domain estimates the relative phase shift between two image blocks [7]. This requires the computation of the 2D Fourier transform of the individual blocks in question. Suppose that $b\,[m, n, k]$ and $b\,[m, n, k-1]$ are the rectangular blocks in the current and previous frames, respectively, then the cross-correlation between the two blocks is defined as the convolution of the individual blocks and is given by

$$c_{k,k-1}\,[m, n] = b\,[m, n, k] \otimes \otimes b\,[m, n, k-1] \tag{9.9}$$

The symbol $\otimes\otimes$ denotes 2D convolution. The 2D Fourier transform of equation (9.9) gives the complex-valued cross-power spectrum. Thus,

$$C_{k,k-1}\,(u, v) = B_k\,(u, v)\,B_{k-1}^*\,(u, v) \tag{9.10}$$

where $B_j\,(u, v)\,, j = k-1, k$ is the 2D Fourier transform of $b\,[m, n, j]\,, j = k-1, k$, u is the Fourier frequency in the vertical direction and v the Fourier frequency in the horizontal direction, and $*$ denotes complex conjugate. If the motion between the blocks is d_x in the horizontal direction and d_y in the vertical direction, then the two Fourier transforms are related by

$$B_k\,(u, v) = B_{k-1}\,(u, v) \exp\left(j\left(ud_y + vd_x\right)\right) \tag{9.11}$$

By normalizing equation (9.10) by the magnitude of $C_{k,k-1}\,(u, v)$, we obtain

$$\hat{C}_{k,k-1}\,(u, v) = \frac{B_k\,(u, v)\,B_{k-1}^*\,(u, v)}{\left|B_k\,(u, v)\,B_{k-1}^*\,(u, v)\right|} = \exp\left(-j\left(ud_y + vd_x\right)\right) \tag{9.12}$$

The 2D inverse Fourier transform of equation (9.12) is the cross-correlation in the spatial domain and is found to be

$$\hat{c}_{k,k-1}\,(m, n) = \delta\left[m - d_y, n - d_x\right] \tag{9.13}$$

Equation (9.13) corresponds to an impulse in the 2D plane located at $\left[m - d_y, n - d_x\right]$, which is the motion vector. In practice, phase-correlation technique is not used for motion estimation because of high computational complexity and phase ambiguity involved.

Optical Flow Method Optical flow or simply optic flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and the scene. It can be used to estimate the motion vectors in a video sequence. Consider an intensity image $f(x, y, t)$ with continuous distribution in the 2D space and time. If the intensity is constant along a motion trajectory, then its total derivative with respect to t is zero, that is,

$$\frac{df(x, y, t)}{dt} = 0 \tag{9.14}$$

Using the chain rule of differentiation and using $\mathbf{x} = \begin{bmatrix} x & y \end{bmatrix}^T$, equation (9.14) can be rewritten as

$$\frac{df(\mathbf{x}, t)}{dt} = \frac{\partial f(\mathbf{x}, t)}{\partial x}\frac{dx}{dt} + \frac{\partial f(\mathbf{x}, t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(\mathbf{x}, t)}{\partial t} = 0 \tag{9.15}$$

Since the components of the velocity vector in the x and y directions are $v_x(t) = dx/dt$ and $v_y(t) = dy/dt$, respectively, equation (9.15) can be written as

$$\frac{\partial f(\mathbf{x}, t)}{\partial x}v_x(t) + \frac{\partial f(\mathbf{x}, t)}{\partial y}v_y(t) + \frac{\partial f(\mathbf{x}, t)}{\partial t} = 0 \tag{9.16}$$

Equation (9.16) is called the OFE *or optical flow constraint* [3]. The objective is to estimate the motion. But since the OFE has two variables, the solution to equation (9.16) will not be unique. An alternative is to minimize the mean square of the optical flow constraint over a block of pixels, assuming that the motion vector is constant over the block [8]. Let

$$D = \sum_{\mathbf{x}\in B}\left\{\frac{\partial f(\mathbf{x}, t)}{\partial x}v_x(t) + \frac{\partial f(\mathbf{x}, t)}{\partial y}v_y(t) + \frac{\partial f(\mathbf{x}, t)}{\partial t}\right\}^2 \tag{9.17}$$

Minimization of D in equation (9.17) with respect to the velocities amounts to setting the respective partial derivatives to zero. Therefore, we have

$$\frac{\partial D}{\partial v_x} = \sum_{\mathbf{x}\in B}\left\{\frac{\partial f}{\partial x}v_x(t) + \frac{\partial f}{\partial y}v_y(t) + \frac{\partial f}{\partial t}\right\}\left\{\frac{\partial f}{\partial x}\right\} = 0 \tag{9.18a}$$

$$\frac{\partial D}{\partial v_y} = \sum_{\mathbf{x}\in B}\left\{\frac{\partial f}{\partial x}v_x(t) + \frac{\partial f}{\partial y}v_y(t) + \frac{\partial f}{\partial t}\right\}\left\{\frac{\partial f}{\partial y}\right\} = 0 \tag{9.18b}$$

In matrix notation, equation (9.18) can be written as

$$\begin{bmatrix} \sum_{\mathbf{x}\in B}\left(\frac{\partial f}{\partial x}\right)^2 & \sum_{\mathbf{x}\in B}\frac{\partial f}{\partial y}\frac{\partial f}{\partial x} \\ \sum_{\mathbf{x}\in B}\frac{\partial f}{\partial x}\frac{\partial f}{\partial y} & \sum_{\mathbf{x}\in B}\left(\frac{\partial f}{\partial y}\right)^2 \end{bmatrix} \begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix} = \begin{bmatrix} -\frac{\partial f}{\partial x}\frac{\partial f}{\partial t} \\ -\frac{\partial f}{\partial y}\frac{\partial f}{\partial t} \end{bmatrix} \tag{9.19}$$

To solve equation (9.19), one has to estimate the gradients $\partial f/\partial x$, $\partial f/\partial y$, and $\partial f/\partial t$. Since we are dealing with digital images defined on integer pixels, the gradients can be approximated using finite differences [9, 10], as given by

$$\frac{\partial f(\mathbf{x},t)}{\partial x} \approx \frac{1}{4}\left\{\begin{array}{l} f[m+1,n,k] - f[m,n,k] + f[m+1,n+1,k] \\ -f[m,n+1,k] + f[m+1,n,k+1] - f[m,n,k+1] \\ +f[m+1,n+1,k+1] - f[m,n+1,k+1] \end{array}\right\} \tag{9.20}$$

$$\frac{\partial f(\mathbf{x},t)}{\partial y} \approx \frac{1}{4}\left\{\begin{array}{l} f[m,n+1,k] - f[m,n,k] + f[m+1,n+1,k] \\ -f[m+1,n,k] + f[m,n+1,k+1] - f[m,n,k+1] \\ +f[m+1,n+1,k+1] - f[m+1,n,k+1] \end{array}\right\} \tag{9.21}$$

$$\frac{\partial f(\mathbf{x},t)}{\partial t} \approx \frac{1}{4}\left\{\begin{array}{l} f[m,n,k+1] - f[m,n,k] + f[m+1,n,k+1] \\ -f[m+1,n,k] + f[m,n+1,k+1] - f[m,n+1,k] \\ +f[m+1,n+1,k+1] - f[m+1,n+1,k] \end{array}\right\} \tag{9.22}$$

Motion estimation through the use of the OFE can be described as follows.

Procedure to Estimate Motion Using OFE

1. For each pixel at location $[m,n] \in B$, compute the gradients $\partial f/\partial x$, $\partial f/\partial y$, and $\partial f/\partial t$ using the approximations in equations (9.20) through (9.22).
2. Compute the quantities $\sum_{\mathbf{x}\in B}(\partial f/\partial x)^2$, $\sum_{\mathbf{x}\in B}(\partial f/\partial y)^2$, $\sum_{\mathbf{x}\in B}(\partial f/\partial x)(\partial f/\partial y)$, $\sum_{\mathbf{x}\in B}(\partial f/\partial x)(\partial f/\partial t)$, and $\sum_{\mathbf{x}\in B}(\partial f/\partial y)(\partial f/\partial t)$.
3. Solve for the estimate $\begin{bmatrix}\hat{v}_x & \hat{v}_y\end{bmatrix}^T$ from

$$\begin{bmatrix} \hat{v}_x(t) \\ \hat{v}_y(t) \end{bmatrix} = \begin{bmatrix} \sum_{\mathbf{x}\in B}\left(\frac{\partial f}{\partial x}\right)^2 & \sum_{\mathbf{x}\in B}\frac{\partial f}{\partial y}\frac{\partial f}{\partial x} \\ \sum_{\mathbf{x}\in B}\frac{\partial f}{\partial x}\frac{\partial f}{\partial y} & \sum_{\mathbf{x}\in B}\left(\frac{\partial f}{\partial y}\right)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\frac{\partial f}{\partial x}\frac{\partial f}{\partial t} \\ -\frac{\partial f}{\partial y}\frac{\partial f}{\partial t} \end{bmatrix} \tag{9.23}$$

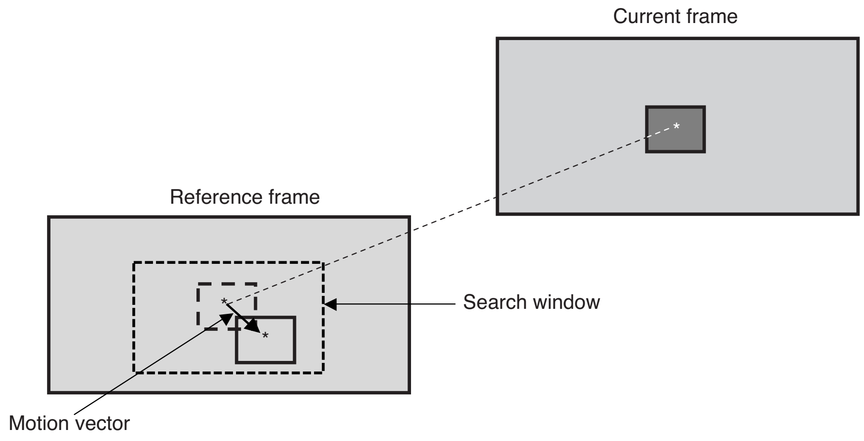


Figure 9.11 A diagram illustrating the process of matching a rectangular block of pixels in the current frame to that in the reference frame within a search window that is larger than the rectangular block but smaller than the image.

Block Matching Method. Block matching is a spatial domain process as well. In block matching technique, a block of pixels in the current frame is matched to a same size block of pixels in the reference frame using a suitable matching criterion. Because the matching process is computationally intensive and because the motion is not expected to be significant between adjacent frames, the matching process is limited to a *search window* of a much smaller size than the image frame [11, 12]. Figure 9.11 illustrates the idea of block matching. The block size is assumed to be $M_1 \times N_1$ and the search window of size $(M_1+2\,M_2) \times (N_1 + 2N_2)$. In the MPEG-2 standard, the macroblock size is 16 × 16 pixels and the search window is of size 32 × 32 pixels. There are several matching criteria to choose from for motion estimation. A few of them are described here.

Minimum MSE Matching Criterion

In minimum MSE case, the best matching block is the one for which the MSE between the block in the current frame and the block within the search window W in the reference frame is a minimum. The MSE between the blocks with candidate displacements d_x and d_y is defined as

$$\mathrm{MSE}\left(d_x, d_y\right) = \frac{1}{M_1 N_1} \sum_{(m,n)\in \mathrm{W}} \left(b\,[m,n,k] - b\left[m - d_y, n - d_x, k - 1\right]\right)^2 \qquad (9.24)$$

The estimate of the motion vector then corresponds to that vector for which the MSE is a minimum, that is,

$$\begin{bmatrix} \hat{d}_x \\ \hat{d}_y \end{bmatrix} = \arg \underbrace{\min}_{(d_x, d_y)} \mathrm{MSE}\left(d_x, d_y\right) \qquad (9.25)$$

Mean Absolute Difference Matching Criterion

The number of arithmetic operations in equation (9.24) can be reduced if we use the Mean absolute difference (MAD) as the best block-matching criterion. MAD between the current and reference blocks is defined as

$$\mathrm{MAD}\left(d_x, d_y\right) = \frac{1}{M_1 N_1} \sum_{(m,n)\in \mathrm{W}} \left| b\left[m, n, k\right] - b\left[m - d_y, n - d_x, k - 1\right] \right| \tag{9.26}$$

The estimate of the motion vector is then given by

$$\begin{bmatrix} \hat{d}_x \\ \hat{d}_y \end{bmatrix} = \arg \underbrace{\min}_{\left(d_x, d_y\right)} \mathrm{MAD}\left(d_x, d_y\right) \tag{9.27}$$

MAD is also known as the *city block distance* and is the most popular matching criterion especially suitable for VLSI implementation and is the recommended criterion for Moving Picture Experts Group (MPEG) standard [6]. In equation (9.26) if we omit the normalizing constant $1/(M_1 N_1)$, the resulting matching criterion is called the *sum of absolute difference* (SAD). Thus, SAD is defined as

$$\mathrm{SAD}\left(d_x, d_y\right) = \sum_{(m,n)\in \mathrm{W}} \left| b\left[m, n, k\right] - b\left[m - d_y, n - d_x, k - 1\right] \right| \tag{9.28}$$

Matching Pixel Count Matching Criterion

In matching pixel count (MPC), each pixel inside the rectangular block within the search window is classified as matching or mismatching pixel according to

$$C\left(m, n;\ d_x, d_y\right) = \begin{cases} 1 & \text{if } \left| b\left[m, n, k\right] - b\left[m - d_x, n - d_y, k - 1\right] \right| \le T \\ 0 & \text{otherwise} \end{cases} \tag{9.29}$$

where T is a predetermined threshold [12]. Then the MPC is the count of matching pixels and is given by

$$\mathrm{MPC}\left(d_x, d_y\right) = \sum_{(m,n)\in \mathrm{W}} C\left(m, n; d_x, d_y\right) \tag{9.30}$$

From equation (9.30), the estimate of the motion vector is obtained as that d_x, d_y for which the MPC is a maximum, that is,

$$\begin{bmatrix} \hat{d}_x \\ \hat{d}_y \end{bmatrix} = \arg \underbrace{\max}_{\left(d_x, dy\right)} \mathrm{MPC}\left(d_x, d_y\right) \tag{9.31}$$

Search Techniques Once a matching criterion is chosen, the task of estimating the motion vector involves searching for the block within the search window that satisfies the chosen criterion. Some of the search techniques are discussed as follows.

Exhaustive or Full Search To find the best matching moving block using any of the criteria described above, one has to compute the appropriate matching metric within the search window for all possible integer-pel displacements. This results in the exhaustive search, also known as the *full search*. Full search is the optimal search method but is computationally intensive. For instance, if $M_1 = N_1 = M_2 = N_2 = 8$, then the number of arithmetic operations required per block can be obtained as follows:

$$\text{Number of searches/block} = 2M_2 \times 2N_2 = 256 \tag{9.32a}$$

$$\text{Number of OPS/search} = M_1 \times N_1 = 64 \tag{9.32b}$$

For minimum MSE criterion,

$$\begin{aligned} 1\ \text{OP} &= (M_1 - 1) \times (N_1 - 1)\,\text{ADD} + M_1 \times N_1\text{MUL} \\ &= 49\ \text{ADD} + 64\ \text{MUL} \end{aligned} \tag{9.32c}$$

Therefore, total number of arithmetic operations required for the full search using the minimum MSE criterion is

$$\begin{aligned} &\text{Total number of OPS/block} = \text{Number of search/block} \times \text{OPS/search} \\ &\quad \times (\text{ADD} + \text{MUL})/\text{OP} = 256 \times 64 \times (49\ \text{ADD} + 64\ \text{MUL}) \\ &\quad \approx 1,048,576\,(\text{ADD} + \text{MUL}) \end{aligned} \tag{9.32d}$$

For CIF image size which is 352 × 240 pixels, the number of 8 × 8 blocks is 1320. Therefore, the total number of arithmetic operations in estimating the motion vectors for all the blocks is approximately 1,384,120,320 additions and multiplications!

Three-Step Search As the name implies, the motion vector corresponding to the best matching block is obtained in three steps, each step involving only nine calculations of the chosen metric [13, 14]. It must be pointed out, though, that the three-step search is suboptimal because it does not search exhaustively for the best matching block. The only penalty is an increase in bit rate because of the increase in the prediction error variance as a result of mismatched motion vectors. Figure 9.12 illustrates the three-step search procedure. In the first step, the matching metric is computed at nine pixel locations marked “1” within the search window including the pixel position marked “X.” The pixel separation is 4. As an example if the minimum value of the metric is found at the lower left pixel location, then the second search is centered at that pixel and the metric is again calculated at the pixels marked “2” with a pixel separation of 2. In the third step, the search is narrowed to the pixels marked “3” with 1 pixel separation and the estimate of the motion vector corresponds to the

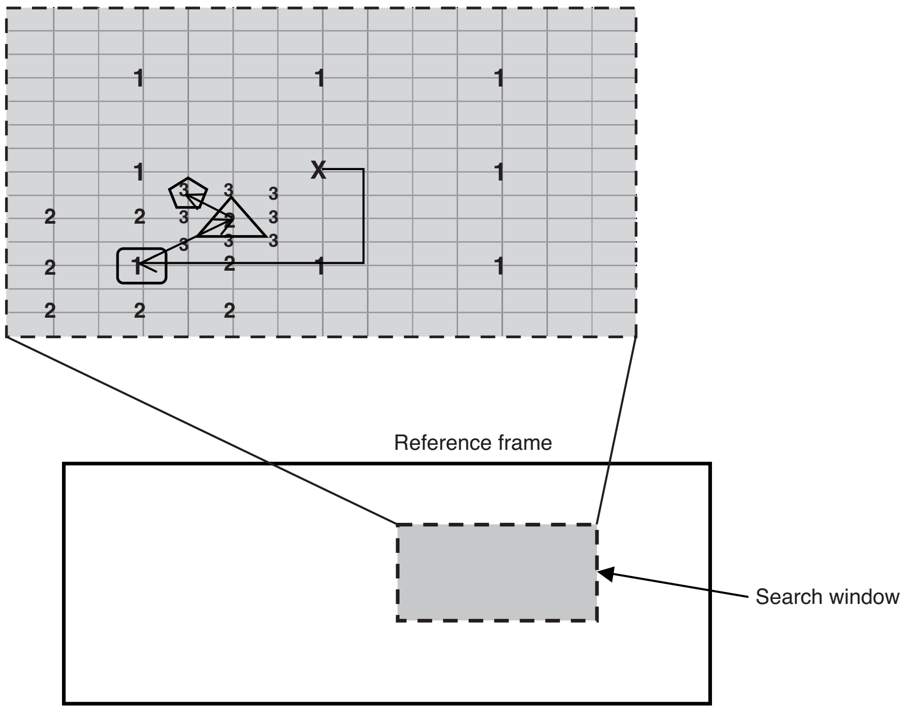


Figure 9.12 A heuristic search technique showing the three-step process of searching for the best matching block. In the first step, the lower left pixel marked "1" is the matching location. In the second step pixel "2" inscribed in a triangle is the matching location, and in the third step the pixel "3" inscribed in a pentagon is the final best matching pixel location.

"3" pixel that results in the least value of the metric. In this example, the final best matching block is located at the pixel "3" enclosed in a pentagon. With the same parameters used for the MSE case, the number of arithmetic operations required for the three-step search is more than an order of magnitude less. Incidentally, the three-step search technique is also known as the *heuristic* or *logarithmic* search [11].

Pyramidal or Hierarchical Motion Estimation Another approach to reducing the number of searches is to estimate the amount of motion progressively from lower resolution to highest resolution images. The temporally adjacent images are first decomposed into a number of levels, with each higher level having a lower resolution. This is known as the *pyramidal* structure or *multiresolution* representation [15–20]. Figure 9.13 shows a three-level pyramid of frame 40 of the Table Tennis sequence. Level 1 is the original image. The level-2 image is obtained by first lowpass filtering the original image using a 2D Gaussian lowpass filter and then subsampling the filtered image by a factor of 2 in both spatial dimensions. Thus, the size of the level-2 image is $\frac{1}{4}$ that of the original or level-1 image. Similarly, the third-level image is obtained by lowpass filtering the level-2 image by the same Gaussian lowpass filter and then subsampling the filtered image by a factor of 2 in both horizontal and

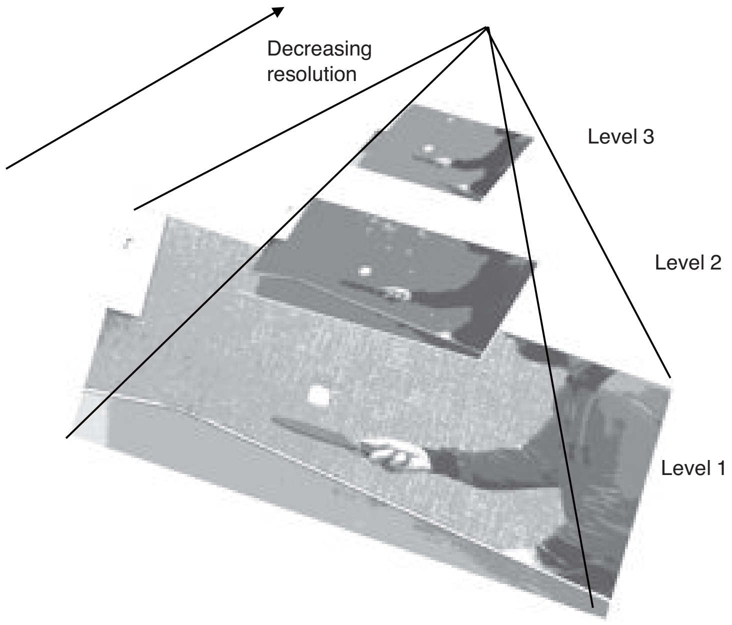


Figure 9.13 Three-level pyramid of frame 40 of the Table Tennis sequence. The lower resolution (higher level) image is obtained by lowpass filtering the previous lower level image followed by subsampling by a factor of 2 in both spatial dimensions. Each higher level image is $\frac{1}{4}$ the size of the previous lower level image.

vertical dimensions. Note that due to repeated lowpass filtering, the higher level images are blurred.

After constructing the pyramids from the two temporally adjacent images, motion is estimated first in the lowest resolution (highest level) reference image. Because of the reduction in size of the level-3 image by a factor of 4 in this case, one should consider 2×2 blocks and a search window of size 4×4. The estimate of the motion vectors will be crude at this level. Once the motion vector is found at level 3 for the current block, it is refined by searching in the next lower level reference image in the neighborhood of the motion vector obtained from the higher level. To do this, the vector and the pixel coordinates must be appropriately scaled, in this case by 2 in each dimension. This process is continued until the search is carried out at the highest resolution (lowest level) reference image. At each higher resolution (lower level), the size of the search window may be reduced from the typical size used with a single reference image. It must be pointed out that like the hierarchical search, the pyramidal search scheme is also suboptimal and yields a poorer estimate of the amount of motion. In comparison to the full search technique, the total number of OPS required to estimate the motion vectors for all the blocks is reduced by more than an order of magnitude, where one OP consists of 1 addition and 1 multiplication. The current and reference frames in this example calculation belong to the Table Tennis sequence frames 40 and 41, respectively.

To summarize what we have discussed thus far, rigid body translational 2D motion on a block basis is determined to single pixel accuracy by finding the block for which the chosen metric within a search window is a minimum. This results in an exhaustive or full search and may be computationally intensive for real-time application. Alternatives to the full search scheme are heuristic and multiresolution schemes, both of which are suboptimal. Let us now work out a couple of examples to illustrate the motion estimation process and MC prediction using the full search and the pyramidal schemes.

Example 9.1 Estimate the motion vectors to single pixel accuracy and compute the MC prediction frame corresponding to the two temporally adjacent frames 40 and 41 of the Table Tennis sequence. Here, frame 40 is the reference frame and frame 41 is the current frame. Both images are intensity images and of size 352 × 240 pixels. Use a block size of 8 × 8 pixels and a search window of size 16 × 16 pixels and use SAD as the matching criterion since it is well suited to the VLSI implementation. Use the full search method. Additionally, estimate half-pixel and quarter-pixel motion vectors and compare the performances to those of the integer-pixel case in terms of prediction error variances and histograms.

Solution The image is divided into 8 × 8 blocks and scanned in a raster scanning order. For a given 8 × 8 rectangular block in the current frame (41), SAD is calculated between it and a same size block within a 16 × 16 search window in the reference frame (40). The search window is centered at the center of the current block in question. To account for the pixels at the boundaries, we have to pad both images by 8 pixels all around the boundaries. Therefore, the size of either image will be (352 +16) × (240 + 16) = 368 × 256. The vector of displacement that results in the least value for the SAD metric is the estimate of the motion vector for that block. This is repeated for all the 8 × 8 blocks in the image. After finding the motion

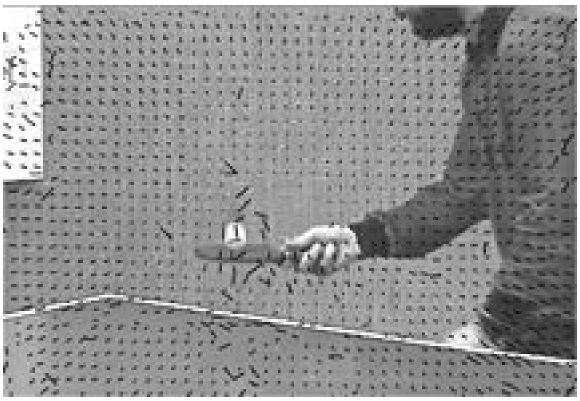


Figure 9.14 Integer-pel motion estimation using the full search technique. Motion vectors are superimposed on the current frame (frame 41).

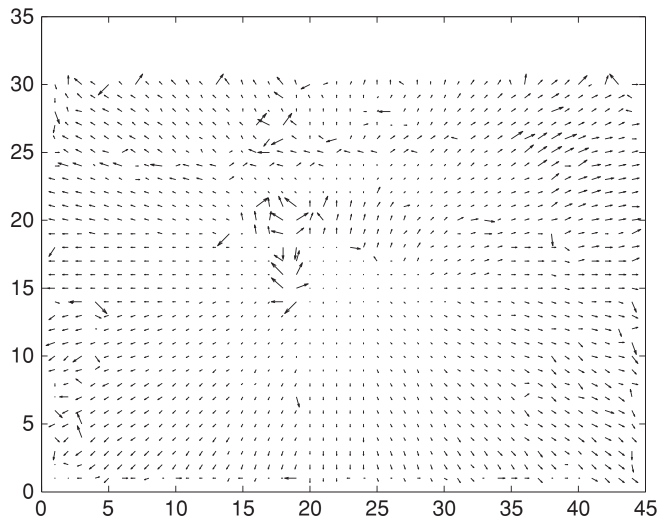


Figure 9.15 Quiver plot of the motion vectors for frame 41. Block size is 8 × 8 pixels and the search window size is 16 × 16 pixels. Magnitude and angle of each motion vector are indicated by the length and orientation of the arrows in the quiver plot.

vector, the difference between the block in the current frame and the block displaced by the motion vector in the reference frame is obtained as the MC prediction error. The histogram of the MC prediction error is then calculated.

Figure 9.14 shows the motion vectors superimposed on frame 41 and Figure 9.15 is a quiver plot of the same motion vectors. The magnitude and angle of the motion vectors are indicated by the length and direction of the arrows of the lines in the quiver plot. The MC prediction error image is shown in Figure 9.16. Compare this

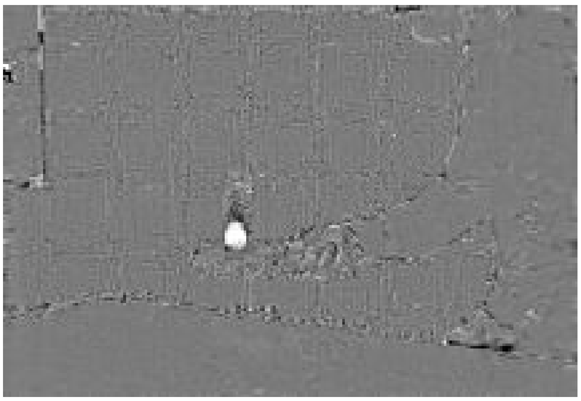


Figure 9.16 MC prediction error image between frames 41 and 40 using full search technique and SAD metric. The variance of the MC prediction error image is 56.

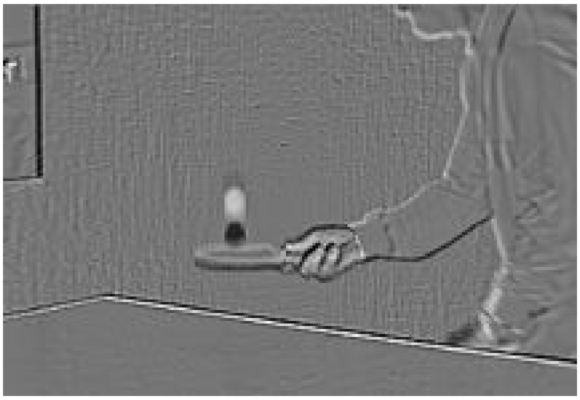


Figure 9.17 Prediction error image using simple differencing. The variance of the resulting prediction error image is 657.

with the simple frame differencing shown in Figure 9.17, where we see a lot of residuals left due to the motion of objects. This is further exemplified by the histograms of the differential images with and without motion compensation shown in Figures 9.18 and 9.19, respectively. The histogram of the prediction error with motion compensation is much narrower and closer to the typical Laplacian distribution. Quantitatively speaking, the prediction error variances with and without motion compensation are 56 and 657, respectively.

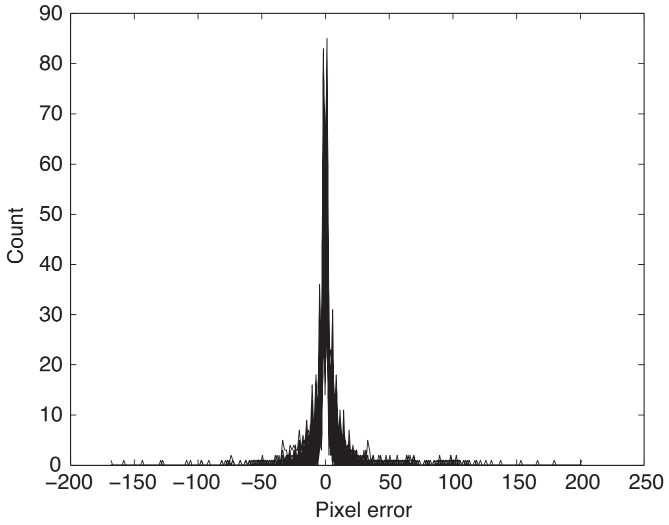


Figure 9.18 Histogram of the MCprediction error image. The reference frame is 40 and the current frame is 41 of the Table tennis sequence.

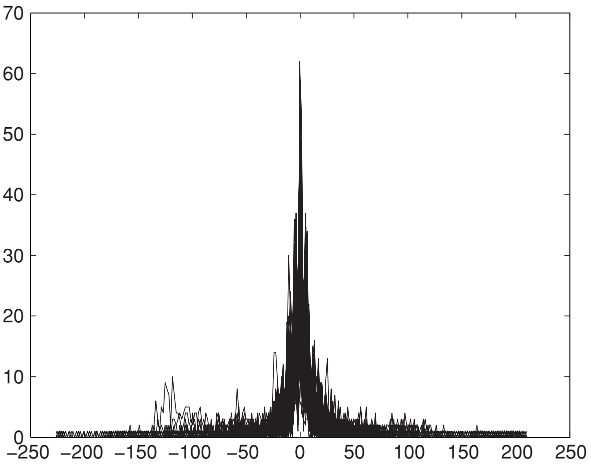


Figure 9.19 Histogram of the simple difference image. The reference frame is 40 and the current frame is 41 of the Table tennis sequence.

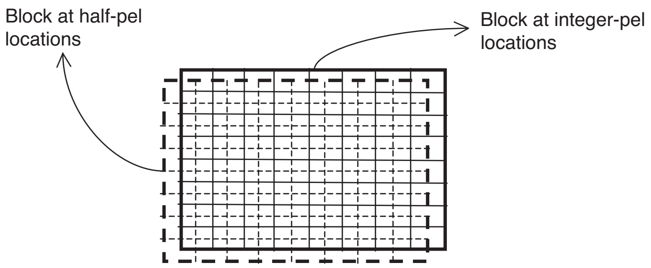


Figure 9.20 A diagram illustrating fractional-pel prediction. The solid lines denote pixels defined on integer grid and the dotted lines indicate interpolated pixels at half- or quarter-pel positions.

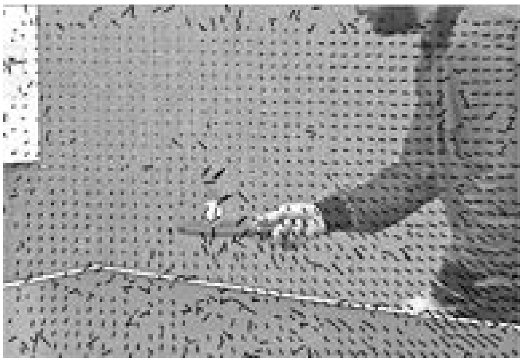


Figure 9.21 Motion vectors superimposed on the current frame 41 of the Table Tennis sequence using half-pel prediction.

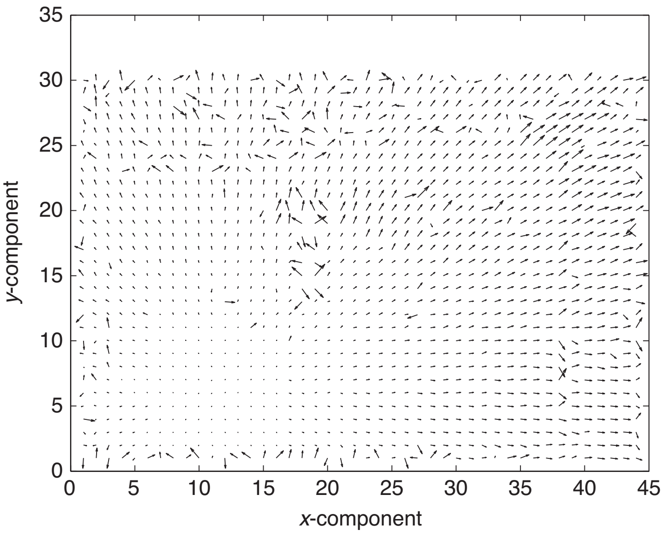


Figure 9.22 Quiver plot of motion vectors of Figure 9.21 using half-pel prediction.

Let us now work on half and quarter-pel prediction. Since the image pixels are defined on integer locations, it is necessary to interpolate the pixels between integer locations. This is diagrammatically shown in Figure 9.20, where the dotted rectangular block is the interpolated pixels. Thus, for each integer-pel location in the search window, pixels are interpolated either at half or quarter-pel position and then the block matching metric is calculated. Interpolation may be classified as "nearest neighbor," "linear," "cubic," or "spline." MATLAB has the built-in function "interp2," which

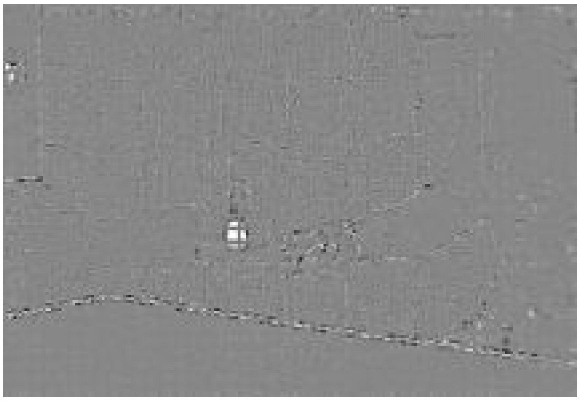


Figure 9.23 MC prediction error image using half-pel prediction and full search. The block matching metric used is SAD. Linear interpolation is used to estimate the pixels at half-pel positions.

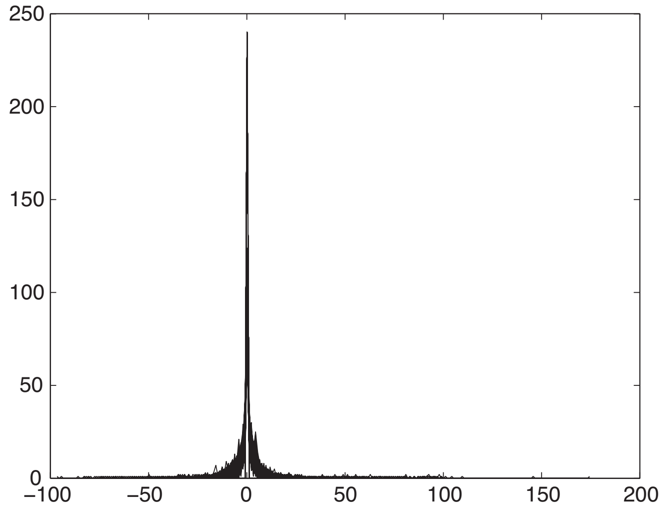


Figure 9.24 Histogram of the MC image of Figure 9.23. The variance of the error image with half-pel motion estimation is 39.5, which represents a 29.5% reduction from that for the integer-pel case.

can implement any of the above-mentioned interpolation rules. It is found that linear interpolation works faster and better than the cubic and spline interpolations.

The quiver plot superimposed on the current frame for half-pel motion is shown in Figure 9.21 and just the quiver plot is shown in Figure 9.22. One can see finer motion in Figure 9.21 as compared with the integer-pel case (Figure 9.14). As a result, the prediction error image is more noise-like with much less edges as seen from Figure 9.23. This is further evident from the histogram of the MC prediction error shown

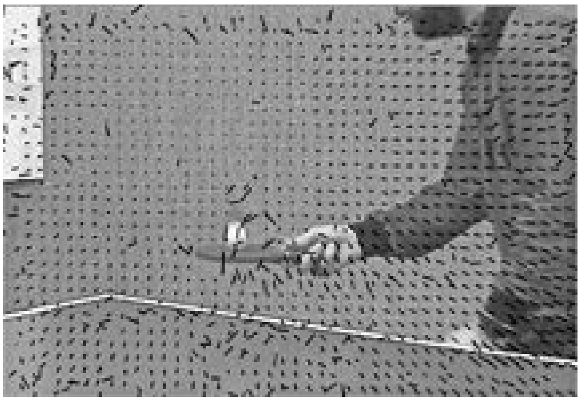


Figure 9.25 Motion vectors superimposed on the current frame 41 of the Table Tennis sequence using quarter-pel prediction.

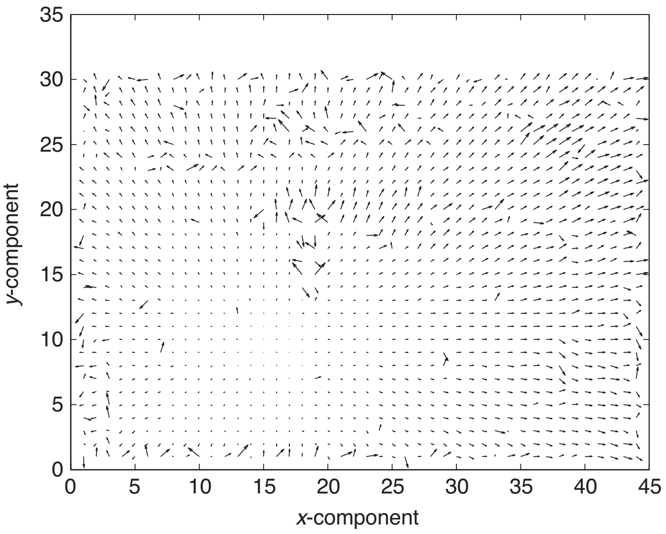


Figure 9.26 Quiver plot of motion vectors of Figure 9.25 using quarter-pel prediction.

in Figure 9.24, which is narrower compared with that of the integer-pel case. The variance of the half-pel prediction error image is 39.5, which represents a 29.5% reduction from that for the integer-pel case.

For the quarter-pel prediction, Figure 9.25–9.28 show the superimposed quiver plot, just the quiver plot, the prediction error image, and the histogram of the error image, respectively. Again, we see marked improvement in the prediction with quarter-pel displacement. The variance of the quarter-pel prediction error image is

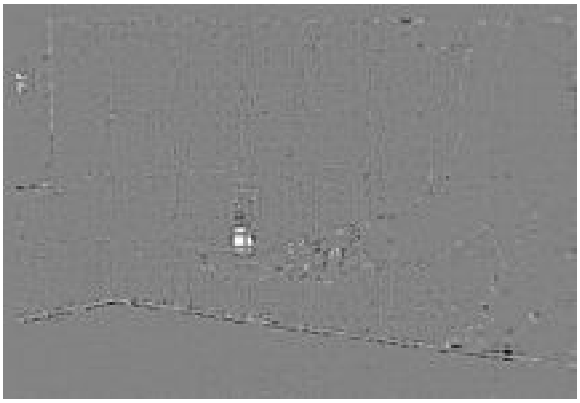


Figure 9.27 MC prediction error image using quarter-pel prediction and full search. The block matching metric used is SAD. Linear interpolation is used to estimate the pixels at quarter-pel positions.

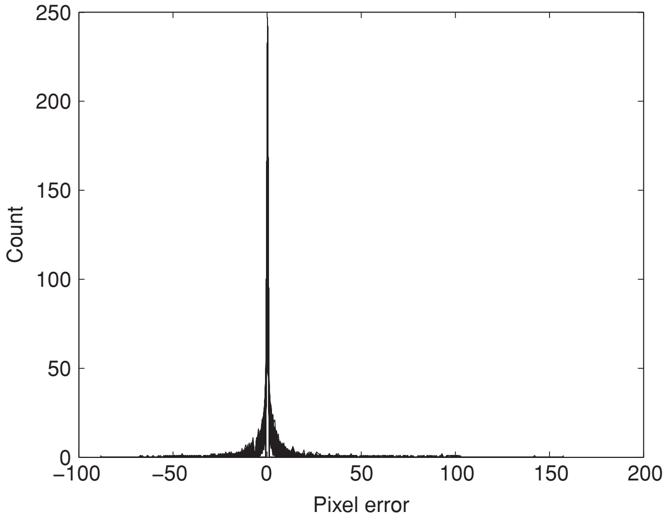


Figure 9.28 Histogram of the MC image of Figure 9.27. The variance of the error image with half-pel motion estimation is 28.9, which represents a 48.4% reduction from that for the integer-pel case.

28.9, which represents a 48.4% reduction from that for the integer-pel case. The MATLAB code for example 9.1 is listed below.

```
% Example9_1.m
% Computes the motion vectors and motion compensated
% prediction error image corresponding to two
% temporally adjacent image frames using the
% Sum of Absolute Difference (SAD) metric and
% full search procedure.
% It can estimate the motion to integer pel, half pel, or
% quarter pel accuracy.
% Motion blocks are of size 8 x 8 pixels and the
% search window size is 16 x 16 pixels.


clear
inFile1 = 'tt040.ras';% table tennis sequence
inFile2 = 'tt041.ras';
%inFile1 = 'twy039.ras';% Trevor sequence
%inFile2 = 'twy040.ras';
%inFile2 = 'twy041.ras';
%inFile1 = 'clairey030.yuv'; % Claire sequence
%inFile2 = 'clairey036.yuv';
A = imread(inFile1); % frame #1
B = imread(inFile2); % frame #2
```

```
N = 8;% block size is N x N pixels
W = 16; % search window size is W x W pixels
PredictionType = 'full'; % Options: "full", "half" and
  "quarter"
% Make image size divisible by 8
[X,Y,Z] = size(A);
if mod(X,8)~=0
    Height = floor(X/8)*8;
else
    Height = X;
end
if mod(Y,8)~=0
    Width = floor(Y/8)*8;
else
    Width = Y;
end
Depth = Z;
clear X Y Z
%
t1 = cputime; % start CPU time
% call appropriate motion estimation routine
switch PredictionType
    case 'full'
        [x,y,pf] = MCpredict_Full(A,B,N,W);
    case 'half'
        [x,y,pf] = MCpredict_Half(A,B,N,W);
    case 'quarter'
        [x,y,pf] = MCpredict_Quarter(A,B,N,W);
end
t2 = cputime; % end CPU time (t2-t1) is the time for
     motion estimation
figure,quiver(x,y,'k'),title('Motion Vector')
sprintf('MSE with MC = %3.3f',std2(pf)*std2(pf))
sprintf('MSE without MC = %3.3f',...
    std2(single(A)-single(B))*std2(single(A)-single(B)))
figure,imshow(pf,[]),title('MC prediction')
figure,imshow(single(A)-single(B),[]),title('prediction
   without MC')
[hNoMC,BinNoMC] = hist(single(A)-single(B),256);
[hMC,BinMC] = hist(pf,256);
figure,plot(BinMC,hMC,'k'),title('histogram of p frame
   with MC')
figure,plot(BinNoMC,hNoMC,'k')
title('histogram of p frame with no MC')


function [x,y,pf] = MCpredict_Full(A,B,N,W)
% [x,y,pf] = MCpredict_Full(A,B,N,W)
% Computes motion vectors of N x N moving blocks
% in an intensity image using full search and
% does a motion compensated prediction to a single pel
  accuracy
% Input:
%   A = reference frame
%   B = current frame
%   N = block size (nominal value is 8, assumed square)
%   W = search window (2N x 2N)
% Output:
%   x = horizontal component of motion vector
%   y = Vertical component of motion vector
%   pf = motion compensated prediction image, same size as
    input image


[Height,Width] = size(A);
% pad input images on left, right, top, and bottom
% padding by replicating works better than padding
  w/ zeros, which is
% better than symmetric which is better than circular
A1 = double(padarray(A,[W/2 W/2],'replicate'));
B1 = double(padarray(B,[W/2 W/2],'replicate'));
x = int16(zeros(Height/N,Width/N));% x-component of
   motion vector
y = int16(zeros(Height/N,Width/N));% y-component of
   motion vector
% Find motion vector by exhaustive search to a single pel
  accuracy
figure,imshow(B), title('Superimposed motion vectors')
hold on % display image & superimpose motion vectors
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
        for u = -N:N
            for v = -N:N
                d = B1(r+1:r+N,c+1:c+N)-A1(r+u+1:
                    r+u+N,c+v+1:c+v+N);
                d = sum(abs(d(:)));% city block distance
                    between pixels
                if d < D
                    D = d;
                    x(rblk,cblk) = v; y(rblk,cblk) = u;
                end
            end
        end
        quiver(c+y(rblk,cblk),r+x(rblk,cblk),...
            x(rblk,cblk),y(rblk,cblk),'k','LineWidth',1)
    end
end
hold off
% Reconstruct current frame using prediction error &
  reference frame
N2 = 2*N;
pf = double(zeros(Height,Width)); % prediction frame
Br = double(zeros(Height,Width)); % reconstructed frame
for r = 1:N:Height
    rblk = floor(r/N) + 1;
    for c = 1:N:Width
        cblk = floor(c/N) + 1;
        x1 = x(rblk,cblk); y1 = y(rblk,cblk);
        pf(r:r+N-1,c:c+N-1) = B1(r+N:r+N2-1,c+N:c+N2-1)...
            -A1(r+N+y1:r+y1+N2-1,c+N+x1:c+x1+N2-1);
        Br(r:r+N-1,c:c+N-1) = A1(r+N+y1:r+y1+N2-
1,c+N+x1:c+x1+N2-1)...
            + pf(r:r+N-1,c:c+N-1);
    end
end
%
figure,imshow(uint8(round(Br))),title('Reconstructed image')


function [x,y,pf] = MCpredict_Half(A,B,N,W)
% [x,y,pf] = MCpredict_Full(A,B,N,W)
% Computes motion vectors of N x N moving blocks
% in an intensity image using full search and
% does a motion compensated prediction to a half pel accuracy
% Input:
%   A = reference frame
%   B = current frame
%   N = block size (nominal value is 8, assumed square)
%   W = search window (2N x 2N)
% Output:
%   x = horizontal component of motion vector
%   y = Vertical component of motion vector
%   pf = motion compensated prediction image, same size as
         input image


[Height,Width] = size(A);
% pad input images on left, right, top, and bottom

A1 = double(padarray(A,[W/2 W/2],'symmetric')); % reference
     block
B1 = double(padarray(B,[W/2 W/2],'symmetric')); % current
```

```
     block
NumRblk = Height/N;
NumCblk = Width/N;
x = zeros(NumRblk,NumCblk);% x-component of motion vector
y = zeros(NumRblk,NumCblk);% %y-component of motion vector
pf = double(zeros(Height,Width)); % prediction frame
% Find motion vectors by exhaustive search to 1/2 pel accuracy
figure,imshow(B), title('Superimposed motion vectors')
hold on % display image & superimpose motion vectors
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
            for u = -N:N
                for v = -N:N
                    %
                    RefBlk = A1(r+u+1:r+u+N,c+v+1:c+v+N);
                    CurrentBlk = B1(r+1:r+N,c+1:c+N);
                    [x2,y2]=meshgrid(r+u+1:r+u+N,c+v+1:c+v+N);
                    [x3,y3] = meshgrid(r+u-0.5:r+u+N-1,
                              c+v-0.5:c+v+N-1);
                    % interpolate at 1/2 pel accuracy
                    z1 = interp2(x2,y2,RefBlk,x3,y3,'*linear');
                    Indx = isnan(z1);
                    z1(Indx == 1) = CurrentBlk(Indx==1);
                    %
                    dd = CurrentBlk - round(z1);
                    d = sum(abs(dd(:)));
                    if d < D
                        D = d;
                        U = u+0.5; L = v+0.5;
                        pf(r-N+1:r,c-N+1:c) = dd;
                    end
                end
            end
        x(rblk,cblk) = L; % Motion in the vertical direction
        y(rblk,cblk) = U; % Motion in the horizontal direction
        quiver(c+y(rblk,cblk),r+x(rblk,cblk),...
            x(rblk,cblk),y(rblk,cblk),'k','LineWidth',1)
    end
end
hold off
% Reconstruct current frame using prediction error &
  reference frame
N2 = 2*N;
Br = double(zeros(Height,Width));
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
        cblk = floor(c/N);
        x1 = x(rblk,cblk); y1 = y(rblk,cblk);
        Indr1 = floor(r+y1+1); Indr2 = floor(r+y1+N);
        if Indr1 <= 0
            Indr1 = 1;
        end
        if Indr2 > Height +N2
            Indr2 = Height + N2;
        end
        Indc1 = floor(c+x1+1); Indc2 = floor(c+x1+N);
        if Indc1 <= 0
            Indc1 = 1;
        end
        if Indc2 > Width +N2
            Indc2 = Width + N2;
        end
        RefBlk = A1(Indr1:Indr2,Indc1:Indc2);
        %
        [x2,y2] = meshgrid(Indr1:Indr2,Indc1:Indc2);
        [x3,y3] = meshgrid(r-N+1:r,c-N+1:c);
        z1 = interp2(x2,y2,RefBlk,x3,y3,'*linear');
        Indx = isnan(z1);
        z1(Indx==1) = RefBlk(Indx == 1);
        Br(r-N+1:r,c-N+1:c)= round(pf(r-N+1:r,c-N+1:c) + z1);
    end
end
figure,imshow(uint8(round(Br))),title('Reconstructed image')

function [x,y,pf] = MCpredict_Quarter(A,B,N,W)
% [x,y,pf] = MCpredict_Quarter(A,B,N,W)
% Computes motion vectors of N x N moving blocks
% in an intensity image using full search and
% does a motion compensated prediction to a quarter pel
  accuracy
% Input:
%   A = reference frame
%   B = current frame
%   N = block size (nominal value is 8, assumed square)
%   W = search window (2N x 2N)
% Output:
%   x = horizontal component of motion vector
%   y = Vertical component of motion vector
%   pf = motion compensated prediction image, same size as
    input image


[Height,Width] = size(A);
% pad input images on left, right, top, and bottom
```

```
A1 = single(padarray(A,[W/2 W/2],'symmetric')); % reference
     block
B1 = single(padarray(B,[W/2 W/2],'symmetric')); % current
     block
NumRblk = Height/N;
NumCblk = Width/N;
x = zeros(NumRblk,NumCblk);% x-component of motion vector
y = zeros(NumRblk,NumCblk);% %y-component of motion vector
pf = single(zeros(Height,Width)); % predicted frame
% Find motion vectors to 1/4 pel accuracy
figure,imshow(B), title('Superimposed motion vectors')
hold on % display image & superimpose motion vectros
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
            for u = -N:N
                for l = -N:N
                    RefBlk = A1(r+u+1:r+u+N,c+l+1:c+l+N);
                    CurrentBlk = B1(r+1:r+N,c+1:c+N);
                    [x2,y2]=meshgrid(r+u+1:r+u+N,c+l+1:c+l+N);
                    [x3,y3]=meshgrid(r+u-0.25:r+u+N-1,
                              c+l-0.25:c+l+N-1);
                    % interpolate at 1/4 pel
                    z1=interp2(x2,y2,RefBlk,x3,y3,'*linear');
                    Indx = isnan(z1);
                    z1(Indx == 1) = CurrentBlk(Indx==1);
                    dd = CurrentBlk - round(z1);
                    d = sum(abs(dd(:)));
                    if d < D
                        D = d;
                        U = u+0.25; L = l+0.25;
                        pf(r:r+N-1,c:c+N-1) = dd;
                    end
                end
            end
        x(rblk,cblk) = L; % Motion in the vertical direction
        y(rblk,cblk) = U; % Motion in the horizontal direction
        quiver(c+y(rblk,cblk),r+x(rblk,cblk),...
            x(rblk,cblk),y(rblk,cblk),'k','LineWidth',1)
    end
end
hold off
% Reconstruct current frame using prediction error &
  reference frame
%
N2 = 2*N;
Br = single(zeros(Height,Width));
```

```
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
        cblk = floor(c/N);
        x1 = x(rblk,cblk); y1 = y(rblk,cblk);
        Indr1 = floor(r+y1+1); Indr2 = floor(r+y1+N);
        if Indr1 <= 0
            Indr1 = 1;
        end
        if Indr2 > Height +N2
            Indr2 = Height + N2;
        end
        Indc1 = floor(c+x1+1); Indc2 = floor(c+x1+N);
        if Indc1 <= 0
            Indc1 = 1;
        end
        if Indc2 > Width +N2
            Indc2 = Width + N2;
        end
        RefBlk = A1(Indr1:Indr2,Indc1:Indc2);
        [x2,y2] = meshgrid(Indr1:Indr2,Indc1:Indc2);
        [x3,y3] = meshgrid(r-N+1:r,c-N+1:c);
        z1 = interp2(x2,y2,RefBlk,x3,y3,'*linear');
        Indx = isnan(z1);
        z1(Indx==1) = RefBlk(Indx == 1);
        Br(r-N+1:r,c-N+1:c)= round(pf(r-N+1:r,c-N+1:c) + z1);
    end
end
figure,imshow(uint8(round(Br))),title('Reconstructed image')
```

Example 9.2 Estimate the motion vectors to single pixel accuracy and compute the MC prediction frame corresponding to the two temporally adjacent frames 40 and 41 of the Table Tennis sequence using the hierarchical approach. These two images are the same as those used in Example 9.1. Use a three-level pyramid structure. Use a 2D Gaussian lowpass filter and a decimation factor of 2 in both horizontal and vertical dimensions. Discuss the results.

Solution We start with the two original images. First, filter the two images using the Gaussian lowpass filter. The MATLAB function "*fspecial*" designs the 2D Gaussian lowpass filter. It has the options to select the filter size and sharpness. It also has the option to set the size of the filtered image to that of the input image using the parameter "*same*." Further, it allows padding the input image with zeros or replication or symmetry. After filtering the images, we down sample them by 2 in both spatial dimensions. This is accomplished by simply retaining every other pixel in each row and every other row. Figure 9.29 shows the three-level multiresolution images of frames 40 and 41. The left image is frame 40 and the right is frame 41. Due

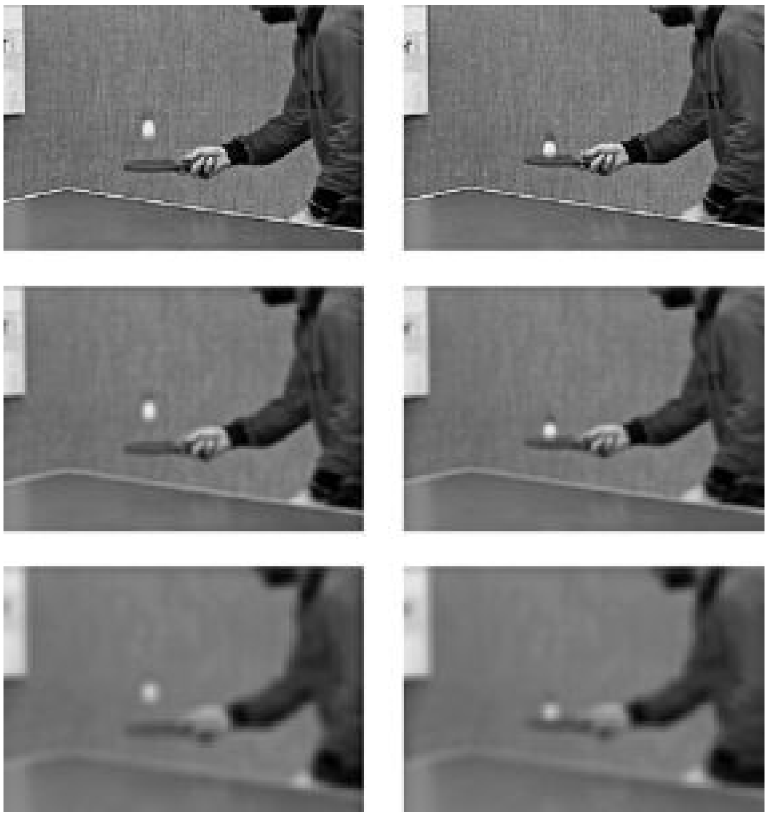


Figure 9.29 A three-level pyramid of frames 40 and 41 of the Table Tennis sequence. Left image: frame 40 (reference frame) with the top image at full resolution, the middle image at level 2, and the bottom image at level 3; right image: frame 41 (current frame) with the top image at full resolution, the middle image at level 2, and the bottom image at level 3. A 2D Gaussian lowpass filter is used for filtering the images. Although the images at each higher level are $\frac{1}{4}$ the size of the previous lower level image, displayed images are all of the same size.

to plotting all the images in the same figure, the images at the three levels appear to be of the same size. MATLAB interpolates the images to have the same size when displaying. However, the level-2 image is only $\frac{1}{4}$ the original image and the level-3 image is $\frac{1}{16}$ the original image. The motion vectors at the three levels are shown in Figures 9.30–9.32. The MC prediction error image can be seen in Figure 9.33. The histograms of the prediction errors with and without motion compensation are shown in Figures 9.34a,b, respectively. Again, we see that the histogram of the MC prediction error image is much narrower than that of without motion compensation and resembles the familiar Laplacian distribution. The prediction error variance using the hierarchical approach turns out to be about 97, which is much more than that for the full search technique. As anticipated, this increase in error variance is

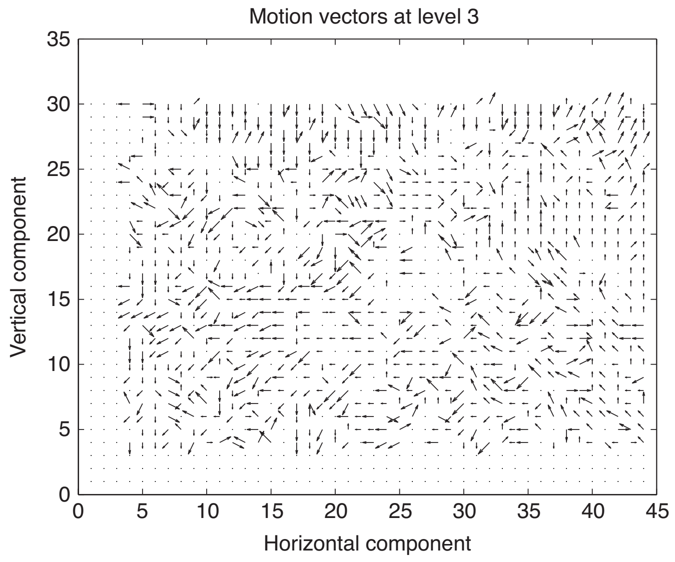


Figure 9.30 Quiver plot of the motion vectors at level 3 of the hierarchical search method. The corresponding pyramids are shown in Figure 9.29.

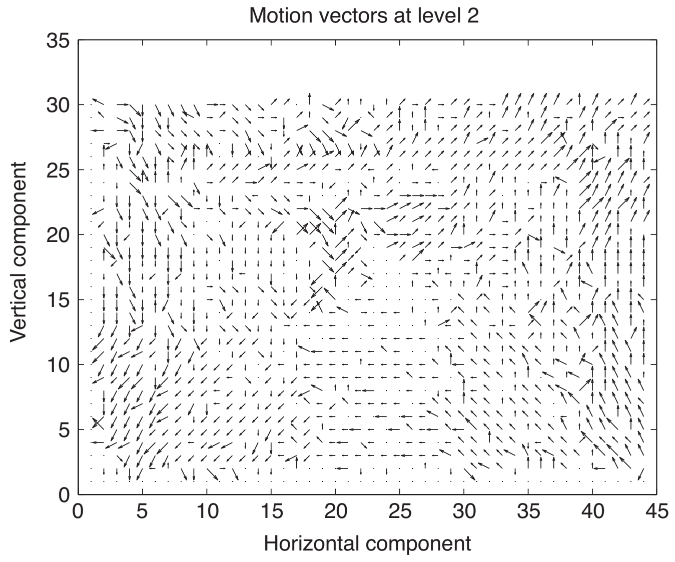


Figure 9.31 Quiver plot of the motion vectors at level 2 of the hierarchical search method. The corresponding pyramids are shown in Figure 9.29.

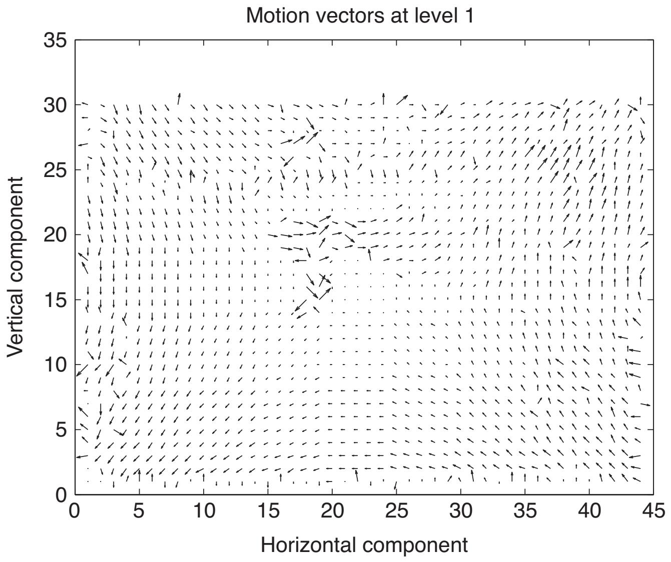


Figure 9.32 Quiver plot of the motion vectors at level 1 of the hierarchical search method. The corresponding pyramids are shown in Figure 9.29.

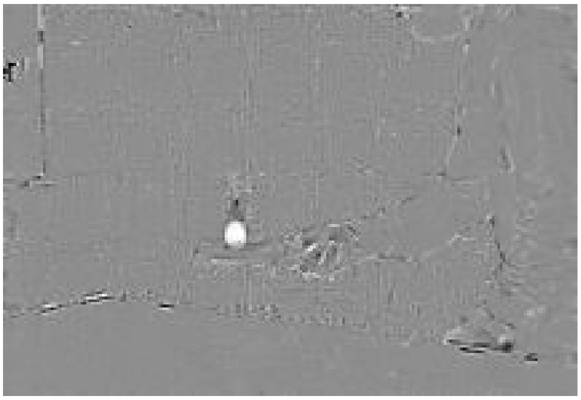
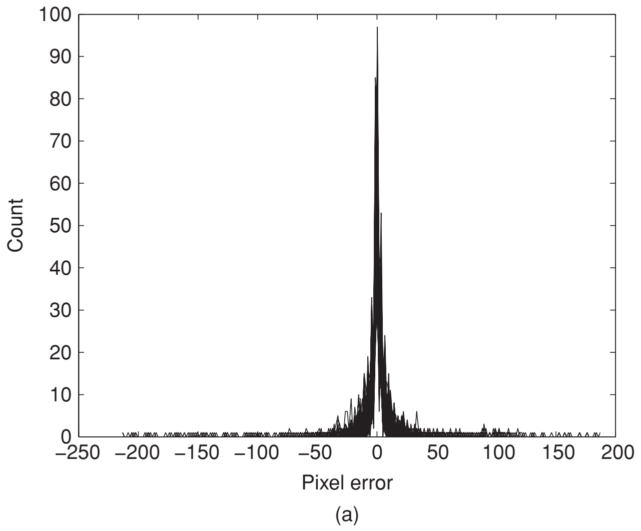


Figure 9.33 MC prediction error image using the hierarchical search. The matching metric used is SAD.



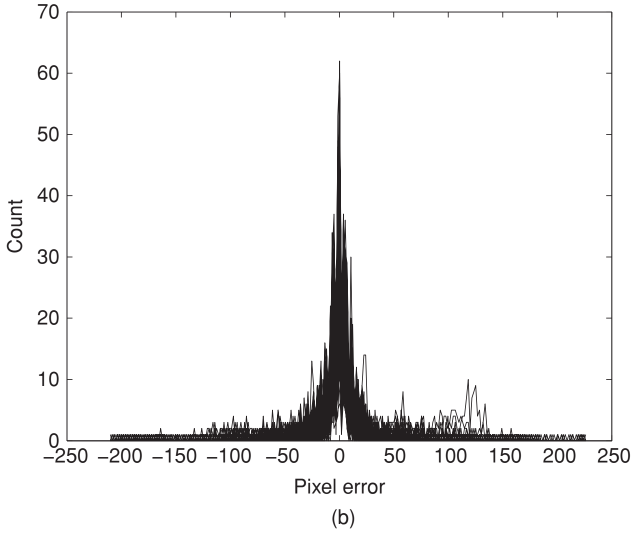


Figure 9.34 Histograms of the prediction error images: (a) with motion compensation using hierarchical search procedure and (b) simple differencing.

the result of not searching exhaustively for the minimum metric, which is SAD in this case.

```
% Example9_2.m
% Computes the motion vectors and motion compensated
% prediction error image corresponding to two
% temporally adjacent image frames using the
% Sum of Absolute Difference (SAD) metric and
% Hierarchical search procedure.
% Motion blocks are of size 8 x 8 pixels
% at the full resolution image.
% Number of levels used is 3. At each higher level,
% the Gaussian lowpass filtered image is subsampled
% by a factor of 2.
% Level 1 is the original or full resolution image,
% level 2 is 1/4 of the original image and
% level 3 image is 1/16 the original image.
% Only integer pel prediction is used in this example.


clear
L = 3; % # levels in the pyramid
P1 = cell(L,1); % cell structure to store the pyramid of
     reference frame
P2 = cell(L,1); % cell structure to store the pyramid of
     current frame
inFile1 = 'tt040.ras';% table tennis sequence
inFile2 = 'tt041.ras';
%inFile1 = 'twy039.ras';% Trevor sequence
%inFile2 = 'twy040.ras';
%inFile1 = 'clairey030.yuv'; % Claire sequence
%inFile2 = 'clairey036.yuv';
P1{1} = imread(inFile1); % original image
P2{1} = imread(inFile2);
[Height,Width] = size(P1{1});
% make the image size divisible by 2^(L-1)
if mod(Height,2^(L-1))~= 0
    Height = floor(Height/(2^(L-1)))*(2^(L-1));
end
if mod(Width,2^(L-1))~= 0
    Width = floor(Width/(2^(L-1)))*(2^(L-1));
end
% Create the L-level pyramid
M = Height; N = Width;
for l = 2:L
    P1{l} = imfilter(P1{l-1},fspecial('gaussian',7,2.5),
            'symmetric','same');
    P2{l} = imfilter(P2{l-1},fspecial('gaussian',7,2.5),
            'symmetric','same');
    % Subsample lowpass filtered image by 2 in both dimensions
    P1{l} = P1{l}(1:2:M,1:2:N);
    P2{l} = P2{l}(1:2:M,1:2:N);
    M = M/2; N = N/2;
end
%
figure,subplot(3,2,1),imshow(P1{1}),title('3-level
   reference frame')
```

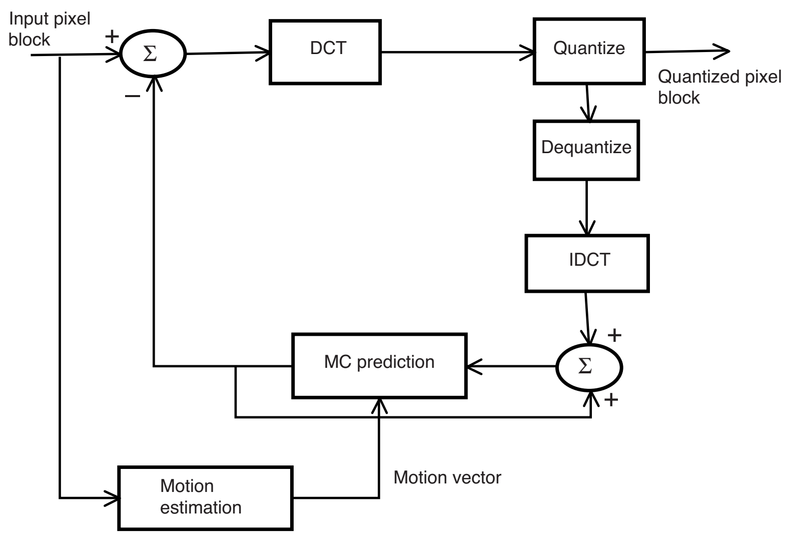
```
subplot(3,2,3),imshow(P1{2})
subplot(3,2,5),imshow(P1{3})
subplot(3,2,2),imshow(P2{1}),title('3-level current frame')
subplot(3,2,4),imshow(P2{2})
subplot(3,2,6),imshow(P2{3})
t1 = cputime; % start counting CPU time
% Find motion vectors & prediction error using
  hierarchical search
[x,y,pf] = MCpyramid_Full(P1,P2);
t2 = cputime; % stop counting CPU time; t2-t1
     is the time for ME
% quiver plot the MV at level 3
figure,quiver(y{3},x{3},'k'),title(['Motion Vectors
    at Level ' num2str(3)])
xlabel('Horizontal component')
ylabel('Vertical component')
% quiver plot the MV at level23
figure,quiver(y{2},x{2},'k'),title(['Motion Vectors
    at Level ' num2str(2)])
xlabel('Horizontal component')
ylabel('Vertical component')
% quiver plot the MV at level 1
figure,quiver(y{1},x{1},'k'),title(['Motion Vectors
    at Level ' num2str(1)])
xlabel('Horizontal component')
ylabel('Vertical component')
% Display prediction error image
figure,imshow(pf,[]), title('Prediction error image')
% Calculate prediction error variance w/ & without MC
sprintf('MC prediction error variance = %3.3f\n',std2(pf)^2)
sprintf('MSE without MC = %3.3f',...
    std2(single(P2{1})-single(P1{1}))*std2(single(P2{1})-
       single(P1{1})))
% Plot the histograms of the prediction error w/ & without MC
[hNoMC,BinNoMC] = hist(single(P2{1})-single(P1{1}),256);
[hMC,BinMC] = hist(pf,256);
figure,plot(BinMC,hMC,'k'),title('histogram of p frame
    with MC')
figure,plot(BinNoMC,hNoMC,'k')
title('histogram of p frame with no MC')


function [x,y,pf] = MCpyramid_Full(A,B)
% [x,y,pf] = M Cpyramid_Full(A,B)
% Computes motion vectors of 8 x 8 moving blocks
% in an intensity image using hierarchical search and
% does a motion compensated prediction to a single pel
  accuracy
```

```
% Input:
%   A = reference frame - cell array w/ 3 cells
%   B = current frame - cell array w/ 3 cells
%
% Output:
%   x = horizontal component of motion vector
%   y = Vertical component of motion vector
%   pf = motion compensated prediction image, same size as
          input image


% Input images A & B each have 3 levels in their pyramids
% Level 1 is the full resolution image of size Height x Width
% Level 2 is the next lower resolution of size
  Height/2 x Width/2
% Level 3 is the lowest resolution image of size
  Height/4 x Width/4
%
% pad input images on left, right, top, and bottom
A1{3} = double(padarray(A{3},[8 8],'symmetric'));
B1{3} = double(padarray(B{3},[8 8],'symmetric'));
A1{2} = double(padarray(A{2},[8 8],'symmetric'));
B1{2} = double(padarray(B{2},[8 8],'symmetric'));
A1{1} = double(padarray(A{1},[8 8],'symmetric'));
B1{1} = double(padarray(B{1},[8 8],'symmetric'));
% Start with block size 2 x 2
N = 2; L = 3;
% x & y are cell arrays w/ 3 elements to store motion
  vector components
% corresponding to the three levels
x = cell(L,1); y = cell(L,1);
[Height,Width] = size(A{3});% Initial image size
corresponding to level 3
x{L} = int16(zeros(Height/N,Width/N));
y{L} = int16(zeros(Height/N,Width/N));
%
% Estimate motion at level 3
for r = N+6:N:Height
    rblk = floor(r/N);
    for c = N+6:N:Width
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
        for u = -2:2
            for v = -2:2
                Indr1 = r+u+1;
                Indr2 = Indr1 + 1;
                Indc1 = c+v+1;
                Indc2 = Indc1 + 1;
                d = B1{L}(r+1:r+N,c+1:c+N)-A1{L}
                    (Indr1:Indr2,Indc1:Indc2);
                d = sum(abs(d(:)));% city block distance
                    between pixels
                if d < D
                    D = d;
                    x{L}(rblk,cblk) = v; y{L}(rblk,cblk) = u;
                end
            end
        end
    end
end
% Estimate motion at level 2
% block size is 4 x 4
N = 4; L = 2;
Height = 2*Height; Width = 2*Width; % double
         the height & width
for r = N+4:N:Height
    rblk = floor(r/N);
    for c = N+4:N:Width
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
        x1 = x{L+1}(rblk,cblk)/4; y1 = y{L+1}(rblk,cblk)/4;
        for u = -2:2
            for v = -2:2
                Indr1 = r+u+y1+1;
                Indr2 = Indr1 + 3;
                Indc1 = c+v+x1+1;
                Indc2 = Indc1 + 3;
                d = B1{L}(r+1:r+N,c+1:c+N)-A1{L}
                    (Indr1:Indr2,Indc1:Indc2);
                d = sum(abs(d(:)));% city block distance
                    between pixels
                if d < D
                    D = d;
                    x{L}(rblk,cblk) = v; y{L}(rblk,cblk) = u;
                end
            end
        end
    end
end
% Estimate motion at level 1 (highest resolution)
% block size is 8 x 8
N = 8; L = 1;
Height = 2*Height; Width = 2*Width; % double
         the height & width
for r = N:N:Height
    rblk = floor(r/N);
    for c = N:N:Width
```

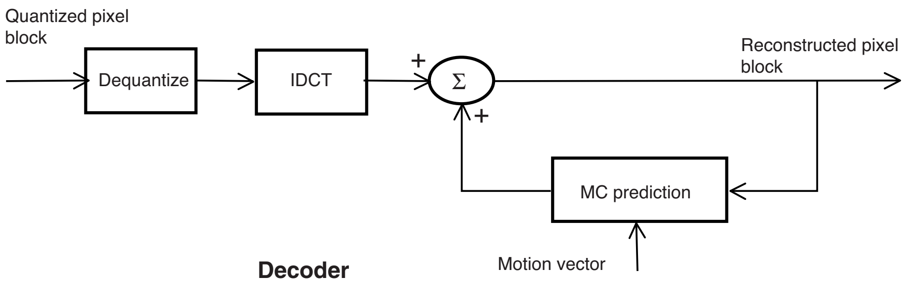
```
        cblk = floor(c/N);
        D = 1.0e+10;% initial city block distance
        x1 = x{L+1}(rblk,cblk)/4; y1 = y{L+1}(rblk,cblk)/4;
        for u = -6:6
            for v = -6:6
                Indr1 = round(r+u+y1+1);
                Indr2 = Indr1 + 7;
                Indc1 = round(c+v+x1+1);
                Indc2 = Indc1 + 7;
                d = B1{L}(r+1:r+N,c+1:c+N)-A1{L}
                    (Indr1:Indr2,Indc1:Indc2);
                d = sum(abs(d(:)));% city block distance
                    between pixels
                if d < D
                    D = d;
                    x{L}(rblk,cblk) = v; y{L}(rblk,cblk) = u;
                end
            end
        end
    end
end
% Reconstruct current frame using prediction error &
  reference frame
N = 8;
[Height,Width] = size(A{1});
N2 = 2*N;
pf = double(zeros(Height,Width)); % prediction frame
Br = double(zeros(Height,Width)); % reconstructed frame
for r = 1:N:Height
    rblk = floor(r/N) + 1;
    for c = 1:N:Width
        cblk = floor(c/N) + 1;
        x1 = x{1}(rblk,cblk); y1 = y{1}(rblk,cblk);
        pf(r:r+N-1,c:c+N-1) = B1{1}(r+N:r+N2-1,c+N:c+N2-1)...
            -A1{1}(r+N+y1:r+y1+N2-1,c+N+x1:c+x1+N2-1);
        Br(r:r+N-1,c:c+N-1) = A1{1}(r+N+y1:r+y1+N2-1,
                              c+N+x1:c+x1+N2-1)...+ pf(r:r+N-
1,c:c+N-1);
    end
end
%
figure,imshow(uint8(round(Br))),title('Reconstructed image')
```

9.2.4 MC Predictive Coding

After having learnt the method of pixel prediction with motion compensation, we now describe a technique to compress video images using MC prediction. Figure 9.35a is a block diagram depicting video encoding using MC prediction. The current



(a)



(b)

Figure 9.35 Block diagram depicting video coding using MC predictive coding: (a) encoder and (b) decoder.

input image is divided into rectangular blocks in a raster scanning order and input to the encoder. The motion, if any, between the current and the reference blocks is estimated and the resulting motion vector is used in the predictor to compensate for the motion and find the predicted block. This predicted pixel block is subtracted from the current input block, the differential block is forward discrete cosine transform (DCT) transformed, and the DCT coefficients are quantized. The quantized DCT coefficients of the MC prediction error are dequantized, inverse DCT transformed, and added to the predicted pixels to recreate the input pixels. Thus, the video coder operates in a closed loop with the decoder in the encoder loop. At the decoder, the quantized pixel blocks are dequantized, inverse DCT transformed, and added to the MC pixel block to reconstruct the image.

There are a few observations to be made:

1. At the encoder side, motion may be estimated between the current and reference blocks using uncompressed original image stored in a frame buffer. Since the encoder and decoder are synchronized and since uncompressed image is not available to the decoder, we must use only the previously decoded image as the reference image and estimate the motion using this reference frame. As a result, the estimate of motion will be incorrect and will gradually worsen as time progresses. To mitigate this deterioration in the quality of the compressed video, one must frequently use a fresh frame called the *key frame*. This prevents the gradual loss in quality. Key frames will be compressed by themselves without any reference to previous frames using any of the *intraframe* coding techniques that we discussed in a previous chapter.
2. Because the MC prediction errors are already decorrelated, applying 2D DCT to the error block does not offer any significant advantage in terms of achieving a higher compression.
3. The coefficients of the quantized DCT of the error block must be entropy coded for transmission or storage. However, in the block diagrams in Figure 9.35, we have omitted the entropy coder and decoder.
4. Motion vectors must also be entropy coded and must be made available at the decoder so that motion compensation can be performed at the decoder.

At the decoder (see Figure 9.35b), the received quantized error block is dequantized, inverse DCT transformed, and added to the previously reconstructed block. Thus, the video sequence is compressed and decompressed. The achievable compression ratio will depend on block motion, quantization strategy, and the entropy coder.

In the following example, we will take a real video sequence and compress and decompress the frames to obtain the performance of the MC video coder in terms of signal-to-noise ratio (SNR) and peak signal-to-noise ratio (PSNR).

Example 9.3 Consider the Table Tennis sequence from frames 41 to 49, inclusive. Use the scheme depicted in Figure 9.35 to interframe code the 10 frames. Take the block size to be 8×8 pixels and use full search method to estimate the block motion with SAD as the matching metric. Use a constant value of 16 as the quantization step size for all the DCT coefficients. Compute the SNR for each decoded frame.

Solution We read frame 40 first which will be the reference frame. Then we read frames 41–49 sequentially. For each frame, we estimate the motion of each 8×8 block within a search window of size 16×16 pixels in the reference frame using full search and SAD block matching metric. After estimating the motion, the current block is aligned with that in the reference frame using the estimated motion

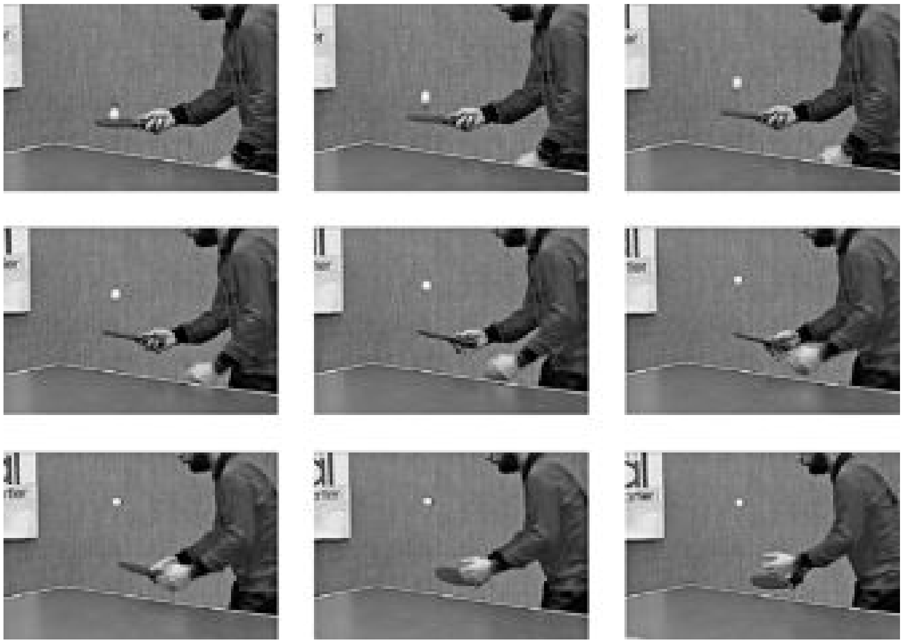


Figure 9.36 Collage of nine consecutive frames from the Table Tennis video.

vector, difference calculated, and 2D DCT applied to the differential block. The DCT coefficients are then quantized and dequantized by uniform quantizer with a quantization step of 16. The 2D inverse DCT transformed block is added to that in the reference block to reconstruct the pixels. Once the entire frame has been coded, the reconstructed frame becomes the reference frame for the next input frame. This is repeated until the entire video sequence is encoded.

Figure 9.36 is the collage of nine frames (41–49) from the Table Tennis sequence. The reconstructed nine frames using the MC video coding are shown in Figure 9.37. The SNR for the nine frames is plotted in Figure 9.38, where the solid line is for the MC predictive coding and the dotted line is the SNR for the simple prediction case. For the simple differencing case, the quantization step size is 19 instead of 16. There is a 2 dB gain in the SNR across the nine frames for the MC predictive coding over simple difference coding. Although there is not a significant difference in the appearance of the reconstructed frames (Figures 9.39a,b), the difference between the MC coded image and image with simple differencing (Figure 9.39c) shows some patches corresponding to areas where there is motion. As a matter of fact, the variance of the image in Figure 9.38c is about 31. The SNRs for the Trevor and Claire sequences are shown in Figures 9.40 and 9.41, respectively. For both Trevor and Claire sequences, frames 41–49 in the respective sequence

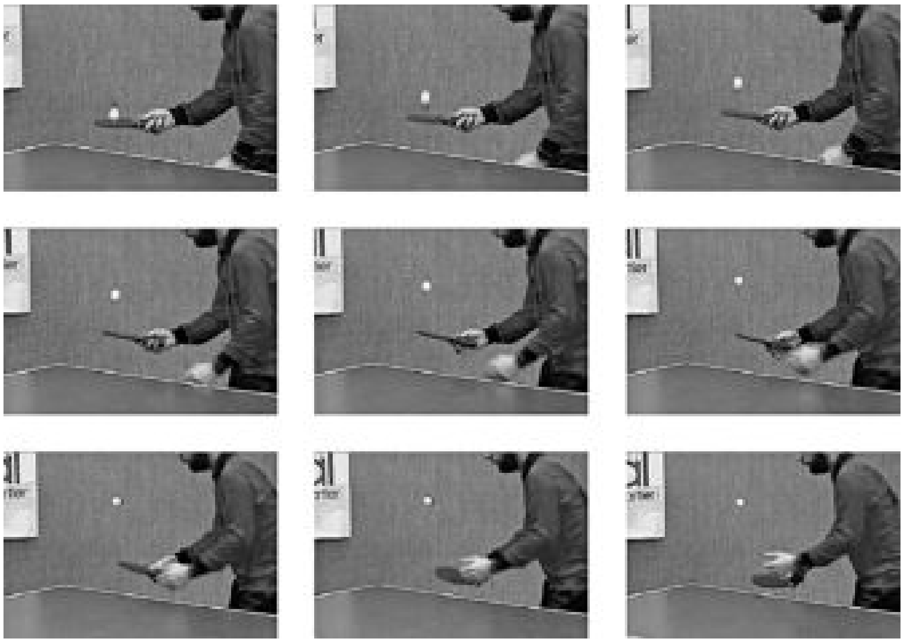


Figure 9.37 Reconstructed nine frames of the images in Figure 9.36 using the MC predictive coding scheme of Figure 9.35.

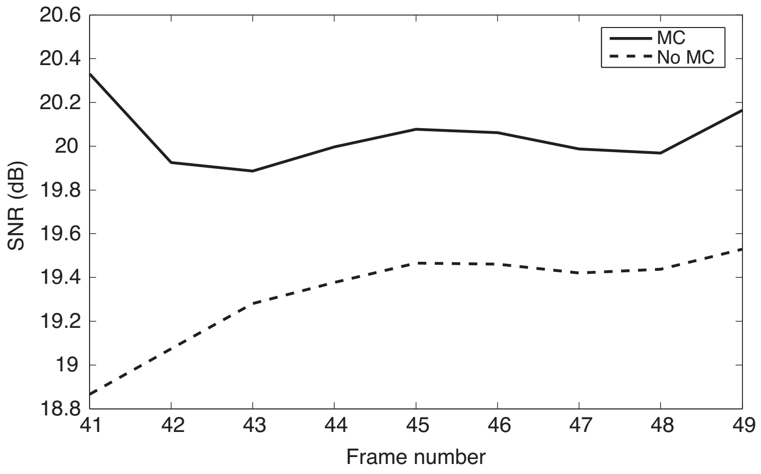


Figure 9.38 Plot of SNR in dB versus frame number of the Table Tennis video using the scheme in Figure 9.35. The solid line corresponds to the MC prediction and the dotted line to the simple differencing.

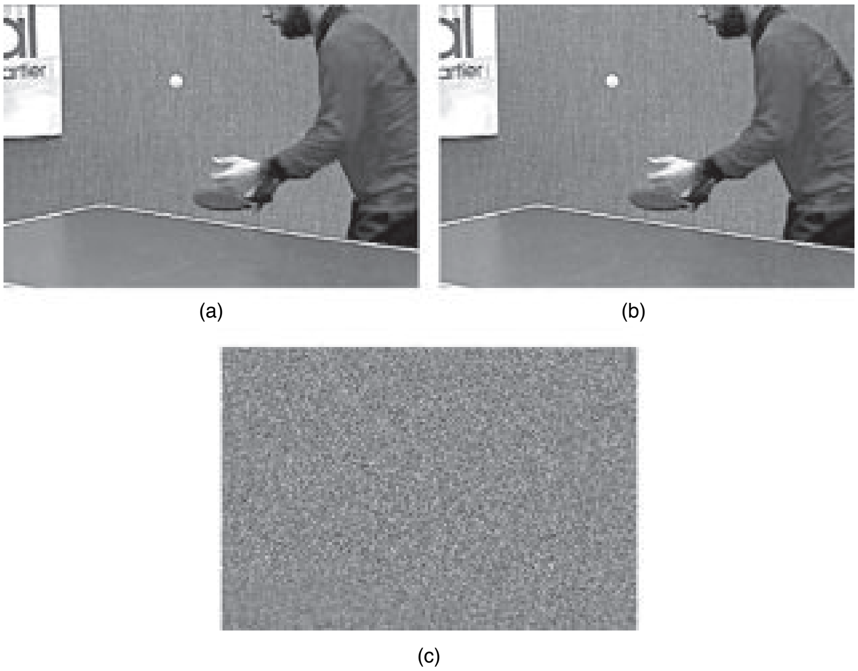


Figure 9.39 Reconstructed image frame 49: (a) reconstruction using MC prediction, (b) reconstruction using simple differencing, and (c) difference between the images in (a) and (b).

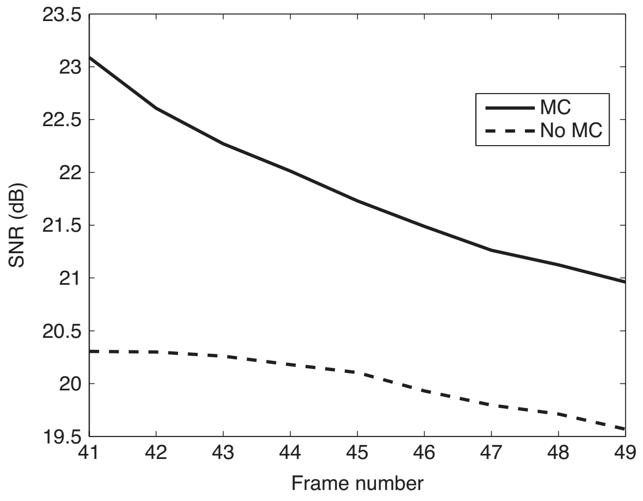


Figure 9.40 Plot of SNR in dB versus frame number of the Trevor video using the scheme in Figure 9.35. The solid line corresponds to the MC prediction and the dotted line to the simple differencing.

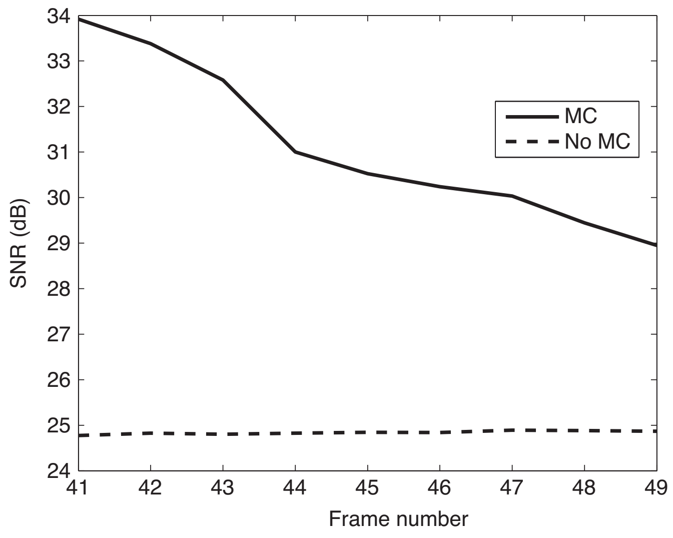


Figure 9.41 Plot of SNR in dB versus frame number of the Claire video using the scheme in Figure 9.35. The solid line corresponds to the MC prediction and the dotted line to the simple differencing.

are used. We observe the same trend in the SNR for the MC prediction and simple differencing cases.

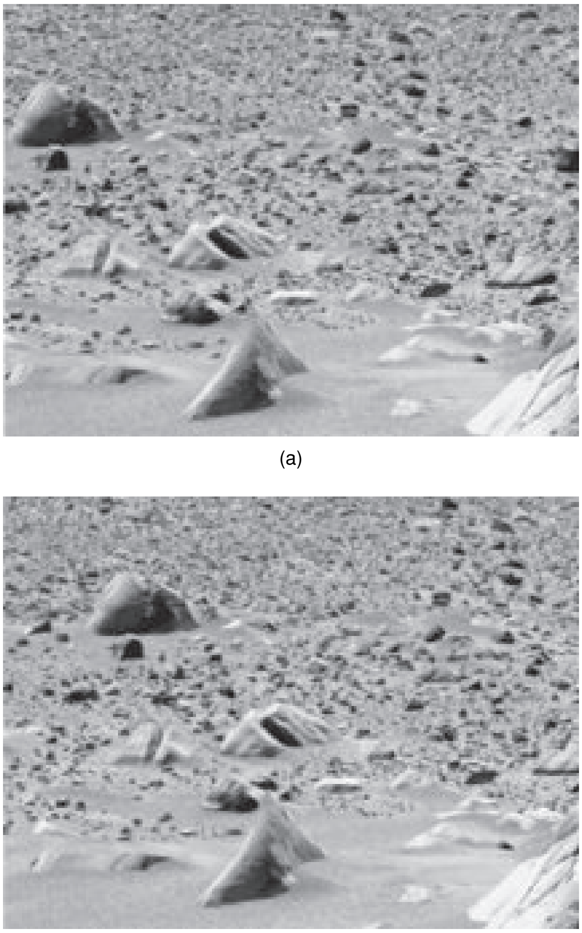
```
% Example9_3.m
% Video coding using motion compensated prediction
% Block motion is estimated to an integer pel accuracy
% using full search and SAD matching metric.
% Block size is 8 x 8 pixels.
% Search window size is 16 x 16 pixels.
% The differential block is 2D DCT transformed, quantized
% using uniform quantizer with constant quantization
  step of 16.
% The reconstructed block becomes the reference block for
% the next input frame.
% Only intensity (Luma) image sequence is accepted.
% Note: Both Table Tennis and Trevor sequences have ".ras"
% extension while the Claire sequence has ".yuv" extension.


clear
inFile1 = 'tt040.ras';% Table Tennis sequence
%inFile1 = 'twy040.ras'; % Trevor sequence
%inFile1 = 'clairey040.yuv'; % Claire sequence
% "strIndx points to the letter "0" in the file name.
strIndx = regexp(inFile1,'0');
```

```
A = imread(inFile1); % frame #1
N = 8;% block size is N x N pixels
W = 16; % search window size is W x W pixels
% Make image size divisible by 8
[X,Y,Z] = size(A);
if mod(X,8)~=0
    Height = floor(X/8)*8;
else
    Height = X;
end
if mod(Y,8)~=0
    Width = floor(Y/8)*8;
else
    Width = Y;
end
Depth = Z;
clear X Y Z
% pad the initial frame left & right and top & bottom
A = double(padarray(A,[W/2 W/2],'replicate')); % for MC
    prediction
A1 = double(padarray(A,[W/2 W/2],'replicate')); % for sim-
ple differencing
N2 = 2*N;
F = int16(41:49); % number of frames to encode
% Find motion vector by exhaustive search to a single pel
  accuracy
Cnt = 1;
% arrays to store SNR and PSNR
snr = zeros(length(F),1); snr1 = zeros(length(F),1);
psnr = zeros(length(F),1); psnr1 = zeros(length(F),1);
figure
figure
for f = F
    inFile = strcat(inFile1(1:strIndx(1)),num2str(f),'.ras');
    %inFile = strcat(inFile1(1:strIndx(1)),num2str(f),'.yuv');
    B = imread(inFile); % read the current frame
    figure(1),subplot(3,3,Cnt),imshow(B)
    B = double(padarray(B,[W/2 W/2],'replicate')); % pad the
        current frame
    for r = N:N:Height
        rblk = floor(r/N);
        for c = N:N:Width
            cblk = floor(c/N);
            D = 1.0e+10;% initial city block distance
            for u = -N:N
                for v = -N:N
                    d = B(r+1:r+N,c+1:c+N)-A(r+u+1:
                        r+u+N,c+v+1:c+v+N);
                    d = sum(abs(d(:)));% city block distance
                        between pixels
                    if d < D
                        D = d;
                        x1 = v; y1 = u; % motion vector
                    end
                end
            end
            % MC compensated difference coding
            temp = B(r+1:r+N,c+1:c+N)...
                -A(r+1+y1:r+y1+N,c+1+x1:c+x1+N);
            TemP = dct2(temp); % DCT of difference
            s = sign(TemP); % extract the coefficient sign
            TemP = s .* round(abs(TemP)/16)*16; % quantize/
                   dequantize DCT
            temp = idct2(TemP); % IDCT
            Br(r-N+1:r,c-N+1:c) = A(r+1+y1:r+y1+N,c+1+x1:
            c+x1+N) +...temp; % reconstructed block
            % simple difference coding
            T = B(r+1:r+N,c+1:c+N)-A1(r+1:r+N,c+1:c+N);
            Tdct = dct2(T); % DCT of differential block
            s = sign(Tdct);
            % quantize/dequantize & IDCT
            Tidct = idct2(s .* round(abs(Tdct)/19)*19);
            % reconstructed block
            Bhat(r-N+1:r,c-N+1:c) = A1(r+1:r+N,c+1:c+N) +
                                    Tidct;
        end
    end
    figure(2),subplot(3,3,Cnt),imshow(uint8(round(Br)))
    % Calculate the respective SNRs and PSNRs
    snr(Cnt) = 20*log10(std2(B(N+1:Height+N,N+1:Width+N))/...
        std2(B(N+1:Height+N,N+1:Width+N)-Br));
    snr1(Cnt) = 20*log10(std2(B(N+1:Height+N,N+1:Width+N))/...
        std2(B(N+1:Height+N,N+1:Width+N)-Bhat));
    psnr(Cnt) = 20*log10(255/std2(B(N+1:Height+N,N+1:
                Width+N)-Br));
    psnr1(Cnt) = 20*log10(255/std2(B(N+1:Height+N,N+1:
                 Width+N)-Bhat));
    % replace previous frames by the currently
      reconstructed frames
    A = Br;
    A = double(padarray(A,[W/2 W/2],'replicate'));
    A1 = Bhat;
    A1 = double(padarray(A1,[W/2 W/2],'replicate'));
    Cnt = Cnt + 1;
end
figure,plot(F,snr,'k','LineWidth',2), hold on
plot(F,snr1,'k--','LineWidth',2), title('SNR (dB)')
```

```
legend('MC','No MC',0)
xlabel('Frame #'), ylabel('SNR (dB)'), hold off
figure,plot(F,psnr,'k','LineWidth',2), hold on
plot(F,psnr1,'k--','LineWidth',2), title('PSNR (dB)')
legend('MC','No MC',0)
xlabel('Frame #'), ylabel('PSNR (dB)'), hold off
```

9.3 STEREO IMAGE COMPRESSION

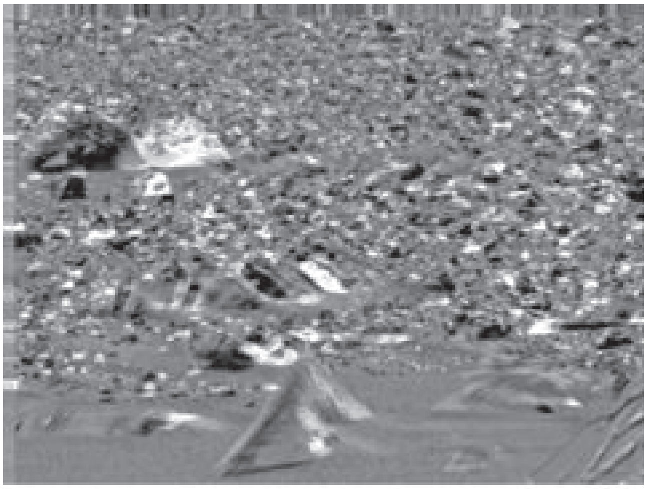
A stereo image pair consists of the left and right views of the same scene. When two views of the same scene are presented to the human visual system, depth is perceived.



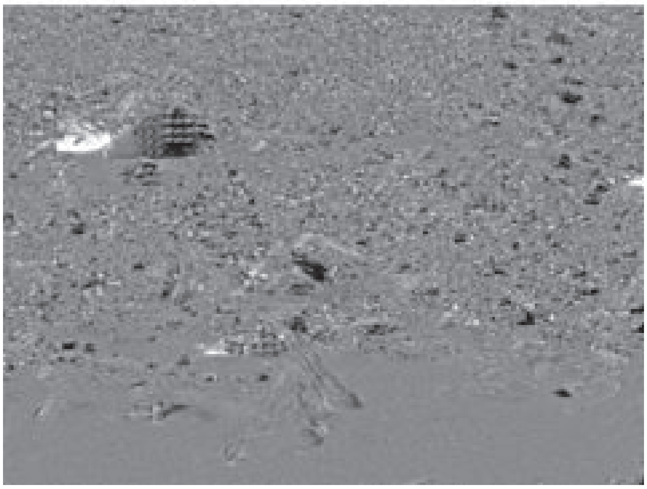
(a)

(b)

Figure 9.42 Stereo image pair of the Martian land: (a) original left image, and (b) original right image. (Courtesy Mark SubbaRao, Adler Planetarium and Astronomy Museum)



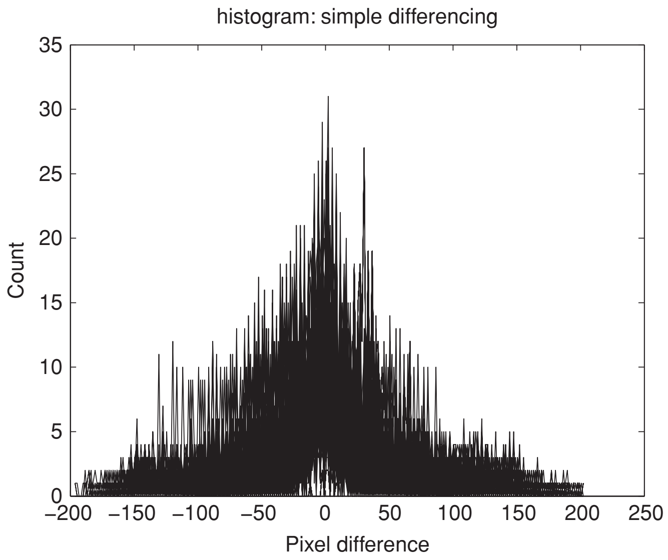
(a)



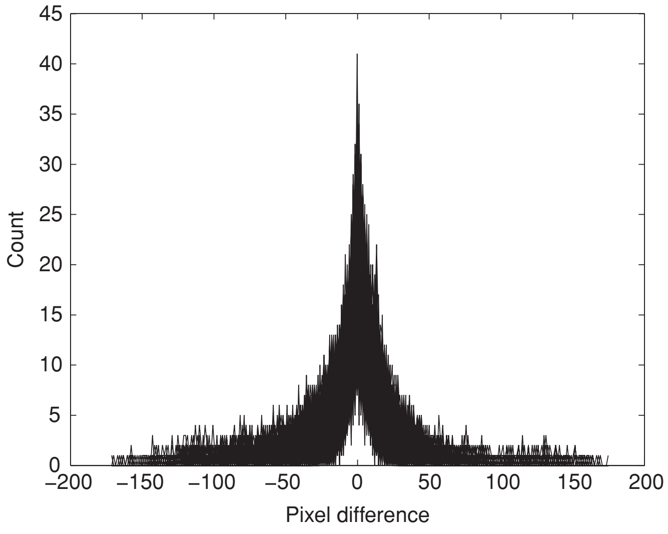
(b)

Figure 9.43 Difference between the right and left images of the stereo image pair of Figure 9.42: (a) simple differencing and (b) DC differencing.

The left and right images of a stereo image pair are very similar. The disparities in the two images are due to the differences in the coordinates of the scene relative to the two eyes. It may be surprising to introduce stereo image compression in this chapter. However, reducing correlation between a pair of stereo images is very similar to reducing temporal correlation in a video sequence. This is the reason for introducing the topic in this chapter. The interested readers may consult Dinstein [21–26] for additional in-depth information on stereo image compression.



(a)



(b)

Figure 9.44 Histogram of the difference stereo image: (a) simple differencing and (b) DC differencing. The variances of the simple difference and DC difference images are 2331 and 635, respectively.

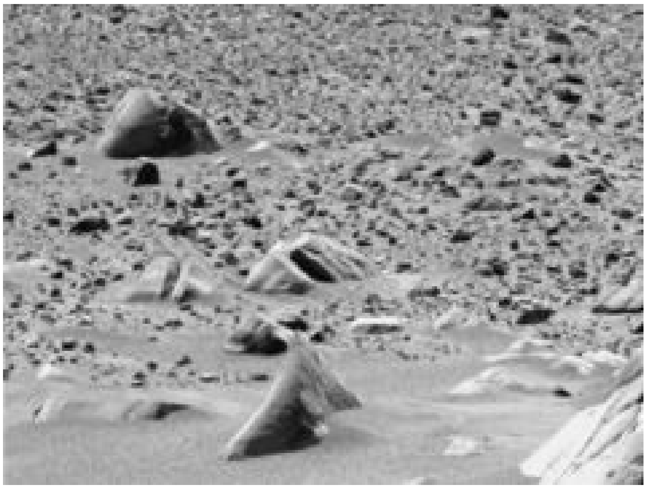


Figure 9.45 Reconstructed right image of the Martian land stereo image using DC coding.

Consider the stereo image pair of the Martian land shown in Figures 9.42a,b. As expected, the left and right images are very similar in content. However, on closer observation one finds that there are some differences, mainly in the positioning of the various objects. The rocks in the two images are oriented slightly differently. At the bottom left, the “riverbed” in the left image is visible more than that in the right image. Due to these slight disparities, it is more efficient to compress the difference in the two images rather than compressing the images individually. Simple differencing will result in a lot of left over residuals, as is evident from Figure 9.43a. Most of the rocks are visible. Therefore, compressing the simple differential image is not efficient.

There are many ways to estimate the disparities between the left and right images of a stereo image pair. Here, we will use the block motion estimation as a means of estimating the disparity between a block in the right image relative to a block in the left image. Once the disparity is estimated, the block in question in the right image can be aligned with that in the left image and the difference block determined. Figure 9.43b is the disparity-compensated (DC) right differential image, which has fewer details than the simple difference image of Figure 9.43a. This is also clearly evident from the histograms of the simple difference and DC difference images shown in Figures 9.44a,b, respectively, where we see the histogram of the DC difference image to be narrower than that of the simple difference image. The variance of the DC difference image is 635 while that of the simple difference image is 2331, which is almost four times larger. Finally, the reconstructed right image is shown in Figure 9.45. The left image is not compressed because it is already compressed with Joint Photographic Experts Group (JPEG) compression scheme. Otherwise the left image has to be compressed first and then the right DC differential image is estimated from the quantized left image. An important point to be mentioned here is that the original stereo image pair is already JPEG compressed and one can obtain better results if uncompressed images are used to start with.

9.4 SUMMARY

In this chapter, we have described the basic idea behind video compression. Due to high correlation in the temporal dimension, prediction in the temporal domain removes or reduces the correlation between frames in a video sequence and results in compression. Temporal prediction is achieved either by simple differencing of two temporally adjacent frames or by optimal linear prediction or by MC differencing. We showed examples to prove the fact that MC prediction results in the lowest variance in the differential image.

Motion of objects between the current and reference frames is estimated using different methods, such as phase correlation, optical flow constraint, and block matching. Block motion estimation requires the minimization or maximization of a cost function. Cost functions such as minimum MSE, minimum MAD, minimum SAD, and so on have been described in connection with block motion estimation. The metrics MAD and SAD are well suited to VLSI implementation and are, therefore, used in MPEG compression standards. In order to estimate motion of a block in the current frame relative to the reference frame, the reference image has to be searched at each pixel location. As this is prohibitive in terms of computational load, a smaller region around the block in question called the search window is usually used in practice.

Exhaustive search for the matching block results in the best motion vector. Because the full search is computationally too intensive for real-time applications, alternative search procedures such as heuristic and hierarchical search techniques are used when computational saving is a key factor. We have described the three-step and pyramidal methods of search with examples. Some examples with MATLAB code for block motion estimation using integer-pel, half-pel, and quarter-pel prediction rules are given. Next, using MC prediction a video coding method is described, which uses 2D DCT of the residuals and uniform quantization in a loop. It was pointed out that due to the use of previously encoded reference frame in the encoder loop to predict motion in the current frame, errors tend to accumulate and the quality of the reconstructed image becomes unacceptable after a few frames and that key frames must be used every so often.

We have introduced stereo image compression here because of the similarity of a stereo image pair to a pair of temporally adjacent images in terms of high correlation. Because of the high correlation between left and right images of a stereo pair higher compression is achieved by coding the difference between the image pair rather than coding the pair individually. This is similar to the MC prediction of video, which is the reason for introducing stereo compression in this chapter. Stereo compression is explained using a stereo image of the Martian land. Figures 9.43a,b clearly demonstrate the fact that disparity-compensated prediction results in much smaller residual variance compared with the simple differencing scheme. The same idea can be extended to stereo video compression as well.

Having learnt the basic principle behind video compression, we will proceed to describe the MPEG standards in the next chapter. MPEG standard is based on the video coder described here.

REFERENCES

1. A. N. Netravali and B. G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, New York, 1988.
2. N. S. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice Hall, Englewood Cliffs, NJ, 1984.
3. A. M. Tekalp, *Digital Video Processing*, Prentice Hall, Upper Saddle River, NJ, 1995.
4. F. Rocca, "Television bandwidth compression utilizing frame-to-frame correlation and movement compensation," in Symposium on Picture Bandwidth Compression, MIT Cambridge, MA, 1969, Gordon & Breach, 1972.
5. F. Rocca and S. Zanoletti, "Bandwidth reduction via movement compensation on a model of the random video process," *IEEE Trans. Commun.*, 960–965, 1972.
6. H. G. Musmann, P. Pirsch, and H. J. Grallert, "Advances in picture coding," *Proc. IEEE*, 73 (4), 523–548, 1985.
7. G. A. Thomas and B. A. Hons, "Television motion measurement for DATV and other applications," *Tech. Rep.*, BBC-RD-1987–11, 1987.
8. B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," Proc. DARPA Image Understanding Workshop, 121–130, 1981.
9. B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artif. Intell.*, 17, 185–203, 1981.
10. J. S. Lim, *Two-Dimensional Signal and Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1990.
11. J. R. Jain and A. K. Jain, "Displacement measurement and its application in interframe image coding," *IEEE Trans. Commun.*, 29, 1799–1808, 1981.
12. H. Gharavi and M. Mills, "Block-matching motion estimation algorithms: new results," *IEEE Trans. Circ. Syst.*, 37, 649–651, 1990.
13. T. Koga et al., "Motion-compensated interframe coding for video conferencing," in NTC 81 Proc., G5.3.1–5, New Orleans, LA, 1981.
14. S. Kappagantula and K. R. Rao, "Motion compensated predictive coding," in Proc. Int. Tech. Symp. SPIE, San Diego, CA, 1983.
15. M. Bierling, "Displacement estimation by hierarchical block-matching," *Proc. Vis. Commun. Image*, SPIE, 1001, 942–951, 1988.
16. P. J. Burt, "Multiresolution techniques for image representation, analysis, and "smart" transmission," in SPIE Conf. Vis. Commun. Image Proc., 2–15, 1989.
17. P. J. Burt and E. H. Adelson, "The Laplacian pyramid as a compact image code," *IEEE Trans. Commun.*, 31, 532–540, 1983.
18. K. M. Uz, *Multiresolution Systems for Video Coding*, Ph.D. Thesis, Columbia University, New York, 1992.
19. K. M. Uz, M. Vetterli, and D. LeGall, "A multiresolution approach to motion estimation and interpolation with application to coding of digital HDTV," in Proc. IEEE Int. Symp. Circ. Syst., pp. 1298–1301, New Orleans, 1990.
20. K. M. Uz, M. Vetterli, and D. LeGall, "Interpolative multiresolution coding of advanced television with compatible subchannels," *IEEE Trans. CAS Video Technol.*, Spec. Issue Signal Process. Adv. Telev., 1(1), 86–99, 1991.

21. I. Dinstein et al., “On stereo image coding,” Ninth Int. Conf. on Pattern Recognition, IEEE Computer Society, Israel, 1988.
22. M. G. Perkins, “Data compression of stereo pairs,” *IEEE Trans. Commun.*, 40 (4), 684–896, 1992.
23. P. D. Gunatilake et al., “Compression of stereo video streams,” *Int. Workshop on HDTV ‘93*, Elsevier, Ottawa, Ontario, Canada, 1993.
24. S. Sethuraman et al., “Multiresolution based hierarchical disparity estimation for stereo image pair compression,” Applications of Subbands and Wavelets (Newark, NJ), NJIT, 1994.
25. S. Sethuraman et al., “A multiresolution framework for stereoscopic image sequence compression,” *Proc. of the First IEEE Int. Conf. on Image Process.*, Austin, TX, vol. II, pp. 361–365, 1994.
26. M. S. Moellenhoff and M. W. Maier, “Transform coding of stereo image residuals,” *IEEE Trans. Image Process.*, 7 (6), 804–812, 1998.

PROBLEMS

9.1. Consider the linear prediction of the pixel $f[m,n,k]$ in the current frame k from the pixels in the current frame and previous frame $k-1$ described by

$$\hat{f}[m,n,k] = a_1 f[m,n-1,k] + a_2 f[m,n,k-1] + a_3 f[m,n-1,k-1]$$

Determine the values of the predictor coefficients that result in the minimum MSE $E\left\{f[m,n,k] - \hat{f}[m,n,k]\right\}^2$ and the corresponding minimum MSE.

9.2. Read any two frames of the Table Tennis sequence and compute the optimal coefficients of the linear predictor described in problem 9.1. Using the coefficients calculated, generate the prediction error image and calculate its variance. Compare this value with the variance of the error image generated using the predictor $\hat{f}[m,n,k] = a\ f[m,n,k-1]$.

9.3. For any two frames of the Trevor sequence, estimate the motion of 8×8 blocks using the phase correlation method described in Section 9.1.3.1.

9.4. Implement the procedure in Section 9.1.3.1 to estimate the motion between blocks in two temporally adjacent frames of the Claire sequence using optical flow method. Show the motion vectors using quiver plot.

9.5. Compare the number of arithmetic operations required to estimate motion between blocks in two temporally adjacent frames using exhaustive search for the matching criteria of minimum MSE, MAD, SAD, and MPC metrics. Assume an aspect ratio of 4:3 for the images.

9.6. Implement the three-step search technique to estimate motion between two consecutive frames of a video using SAD as the metric. Compare the resulting performance with that for the exhaustive search technique.

9.7. Construct three-level pyramids of two consecutive frames of a video sequence using the “db2” wavelet. Note that this is achieved by computing the 2D DWT of the LL band at each level. Now estimate motion between blocks by applying the hierarchical method.

9.8. In the MC predictive coder compute the optimal bits of uniform quantizers for the differential image to achieve an average bit rate of 1 bpp. Then implement the codec to quantize and dequantize a frame of image using the previous frame as the reference image.

9.9. Consider the stereo image pair of the Martian land. Use half-pel prediction to estimate the disparity between the right and left images, and then compute the resulting variance of the differential image. How much improvement have you achieved over integer-pel prediction?

9.10. Repeat Problem 9.9 for quarter-pel prediction for the same stereo image pair.

10

VIDEO COMPRESSION STANDARDS

10.1 INTRODUCTION

In order for a compressed video bit stream to be decodable uniformly by various platforms and devices, the bit stream format must be predefined. Just as there is a standard for say, a power adapter, so that it can be plugged into a socket wherever that standard is compliant, so also there must be a standard for a video compressor, which will enable all standard-compliant compressed video data to be decoded anywhere. Thus, interoperability is the chief reason for establishing a video compression standard. We have already described the Joint Photographic Experts Group (JPEG) standard for still image compression. In this chapter we will attempt to give a brief description of the MPEG standards for video compression. Our intent will be to describe the video coding aspects of the standard, and we will refrain from discussing the syntax of the video bit stream.

MPEG is an acronym for Moving Picture Experts Group. It is a group formed under the auspices of the International Organization for Standardization (ISO) and the International Electro-technical Commission (IEC). MPEG was given a formal status within the ISO/IEC. The original work started by the MPEG group culminated in the standard called MPEG-1 in 1992, ISO/IEC 11172 [1]. The MPEG-1 standard itself comprises five parts. These are part 1: systems; part 2: video; part 3: audio; part 4: compliance testing; and part 5: software simulation. MPEG-1 has been targeted for multimedia applications. It was optimized for compressing progressively scanned CIF images of size 352 $\times$ 240 pixels at 30 fps at data rates of about 1.5 Mb/s.

The standard does not support interlaced video. MPEG-1 delivers an image quality comparable to VHS.

With increasing demand for digital television (TV), MPEG group in 1994 came up with a second standard called MPEG-2 aimed at broadcast TV, ISO/IEC 131818 [2]. It supports interlaced scanning, larger picture sizes, and data rates at 10 Mb/s or higher. MPEG-2 is deployed for the distribution of digital TV, including standard definition television (SDTV), DVD, and high definition television (HDTV). MPEG-2 video part of the standard defines five *profiles* to support a wide range of capabilities from MPEG-1 to very advanced HDTV compression. A straightforward extension of MPEG-1 is the main profile (MP) in MPEG-2.

MPEG-4 was standardized in 1998 and is aimed at very low data rates as well as content-based interactivity on CD-ROM, DVD, and digital TV and universal access, which includes error-prone wireless networks, MPEG-4 [3, 4]. MPEG-4 is the multimedia standard for the fixed and wireless web, enabling integration of multiple applications. A brief description of the features of MPEG-4 follows later in the chapter.

MPEG-7 was standardized in 2004 [5]. MPEG-7, titled *Multimedia Content Description Interface*, provides a rich set of tools for the description of multimedia content. It provides a comprehensive set of audiovisual *description tools* to create descriptions, which will form the basis for applications that enable the needed effective and efficient access to multimedia content.

In what follows, we will attempt to describe the basic ideas behind the video compression part of MPEG-1 and MPEG-2 standards and give brief explanations of the ideas behind MPEG-4 and advanced video coding (AVC). The MPEG standards' documents are grouped into two classes. The *Normative* sections define the actual standard. In fact, the standard only defines the compressed bit stream format and the decoder simply follows the standards' rule in reconstructing the video. The *Informative* section of the standards contains the necessary background information on the encoding/decoding principles.

10.2 MPEG-1 AND MPEG-2 STANDARDS

10.2.1 MPEG Video Layer Terminology

Group of Pictures As pointed out earlier, the MPEG standard specifies only the coded bit stream at the output of the encoder and the decoder simply follows the bit stream syntax to decompress the video sequence. The MPEG video bit stream consists of many layers, and the outermost layer is the video sequence layer. A video sequence is made up of groups of pictures with each group consisting three kinds of pictures, namely, I-picture, P-picture, and B-picture. As the name implies, I-pictures or intracoded pictures exploit spatial redundancy only and are, therefore, coded by themselves without reference to other pictures in the GOP [6]. On the other hand, P-pictures (predictive-coded pictures) and B-pictures (bi-directionally predictive-coded pictures) are compressed with reference to either I- or P-pictures.

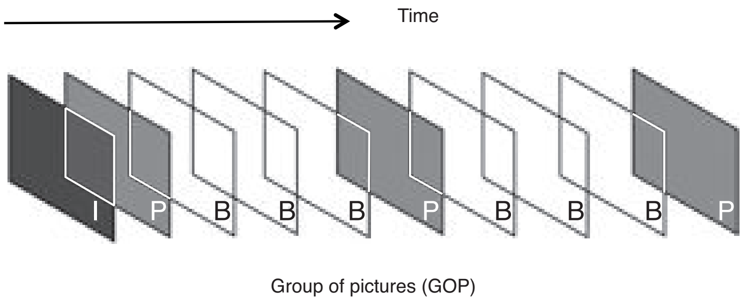


Figure 10.1 A typical GOP in MPEG coding. The order shown is the order in which the pictures are decoded.

A typical group of pictures (GOP) in an MPEG video at the coder side is illustrated in Figure 10.1. This is the order in which the decoder should decode the compressed bit stream. The first P-picture is compressed using the previous I-picture as the reference picture. The first three B-pictures are compressed using the first I-, P-pictures, and the next P-picture. Note that it is not necessary to have a GOP with all the three types of pictures. Instead, a GOP may contain only I- and P-pictures. Also, an MPEG bit stream may have closed or open GOPs. In a closed GOP, P- and B-pictures are predicted from other I- and P-pictures from the same GOP, whereas in an open GOP P- and B-pictures may be predicted from I- and P-pictures from outside the GOP. Because the B-pictures are referenced to both previous and future pictures, the order in which the decoded pictures are presented to the viewer is different from that in which it is coded. Figure 10.2 shows the display order of the GOP in Figure 10.1. Observe the B-pictures follow the I-picture and the P-pictures follow the B-picture, and so on.

Macroblock Each picture in an input video sequence is divided into *macroblocks*, which are the building blocks of an MPEG video. Each macroblock is made up of one 16 × 16 luma pixels, one 8 × 8 Cb pixels, and one 8 × 8 Cr pixels. However, the compression based on discrete cosine transform (DCT) is performed on each 8 × 8 block of pixels in all the three components.

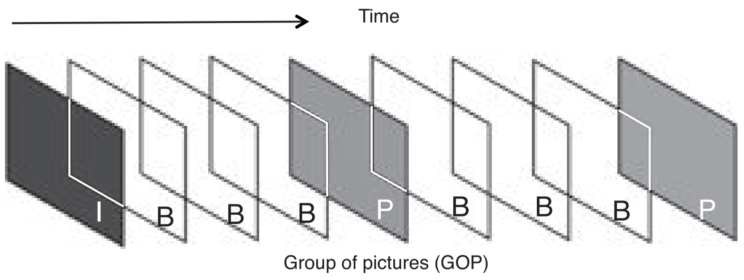


Figure 10.2 Display order of the GOP in Figure 10.1.

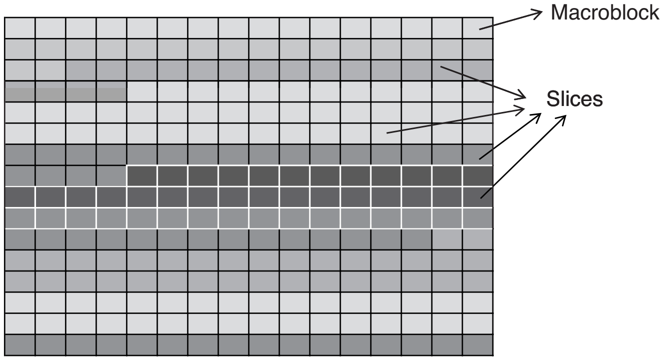


Figure 10.3 Slice structure used in MPEG: a slice is made up of contiguous macroblocks.

Slice The MPEG-compressed bit rate is variable. For the transmission of the compressed bit stream through a constant bit rate channel, output buffer and rate control strategy are required. In order to accommodate rate control strategy for constant output bit rate and signaling changes to coding parameters, each picture in the sequence is divided into *slices*. This division into slices gives more flexibility and control over the output bit rate. A slice is made up of several contiguous macroblocks in a raster scan order, as shown in Figure 10.3.

10.2.2 MPEG Coding

With these nomenclatures we will now describe the coding principle behind the MPEG standards. As we go along, we will compare and contrast the MPEG-1 and MPEG-2 standards. A block diagram showing the process of compressing a video sequence applicable to the MPEG standards is shown in Figure 10.4. The system shown in Figure 10.4 is suitable for I- and P-picture coding. As can be seen from the figure, the MPEG video coding is essentially predictive in nature. It decorrelates the pictures in the temporal domain by obtaining the difference between the current and previous frames using motion estimation and compensation. Compression of video is achieved because the differential frames have much lower dynamic range in intensities with Laplacian-like distribution. A detailed explanation of the components of the MPEG video coding depicted in Figure 10.4 is as follows.

Preprocess This block performs the conversion of the input RGB sequence into the YCbCr sequence as well as chroma sampling format conversion to 4:2:0 or 4:2:2. MPEG-2 allows 4:4:4 sampling format as well, where all three components have the full native resolution. The 4:4:4 sampling format is usually used at the digital mastering side. MPEG-2 allows interlaced video, while MPEG-1 only uses progressive video. The chroma sampling formats for MPEG-1 and MPEG-2 are shown in Figures 10.5a–e. There is a difference in the 4:2:0 sampling format between the two standards. As seen from Figures 10.5b,c, the chroma sample is placed symmetrically

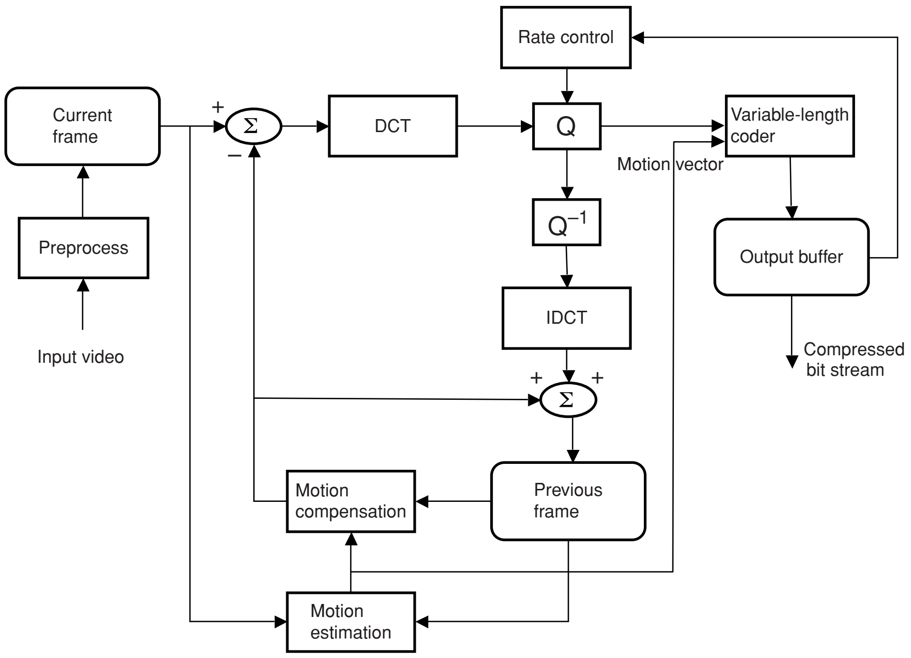


Figure 10.4 Block diagram of MPEG video coding. This system is suitable for coding I- and P-pictures.

between the four Y samples in the MPEG-1 standard, whereas it is aligned vertically in the MPEG-2 standard. In the case of interlaced video, chroma samples coexist with Y samples only in the top field and in the bottom field, only Y samples exist (see Figures 10.5d,e). For the progressive case in MPEG-2, the arrangement of the samples is identical to that shown in Figure 10.5c.

Motion Estimation As we pointed out in the last chapter, motion compensated differencing is superior to simple differencing. Therefore, the encoder has to first estimate the motion of each macroblock in the current frame with reference to the previous frame. This processing block estimates the motion using the SAD metric and also generates the motion vectors for each macroblock [7–9]. Motion vectors are sent to the entropy coder for transmission or storage. Motion is estimated only for the luma blocks and the same motion vectors are used for the chroma blocks with suitable scaling to take into account the sampling formats used. The actual search procedure used in the estimation of block motion is left to the encoder, which depends on the availability of a suitable real-time hardware processor.

Motion Compensation Once the motion of a macroblock is estimated, it is then used to align with the appropriate block in the reference frame and to find the difference between the current macroblock and the aligned block in the reference frame.

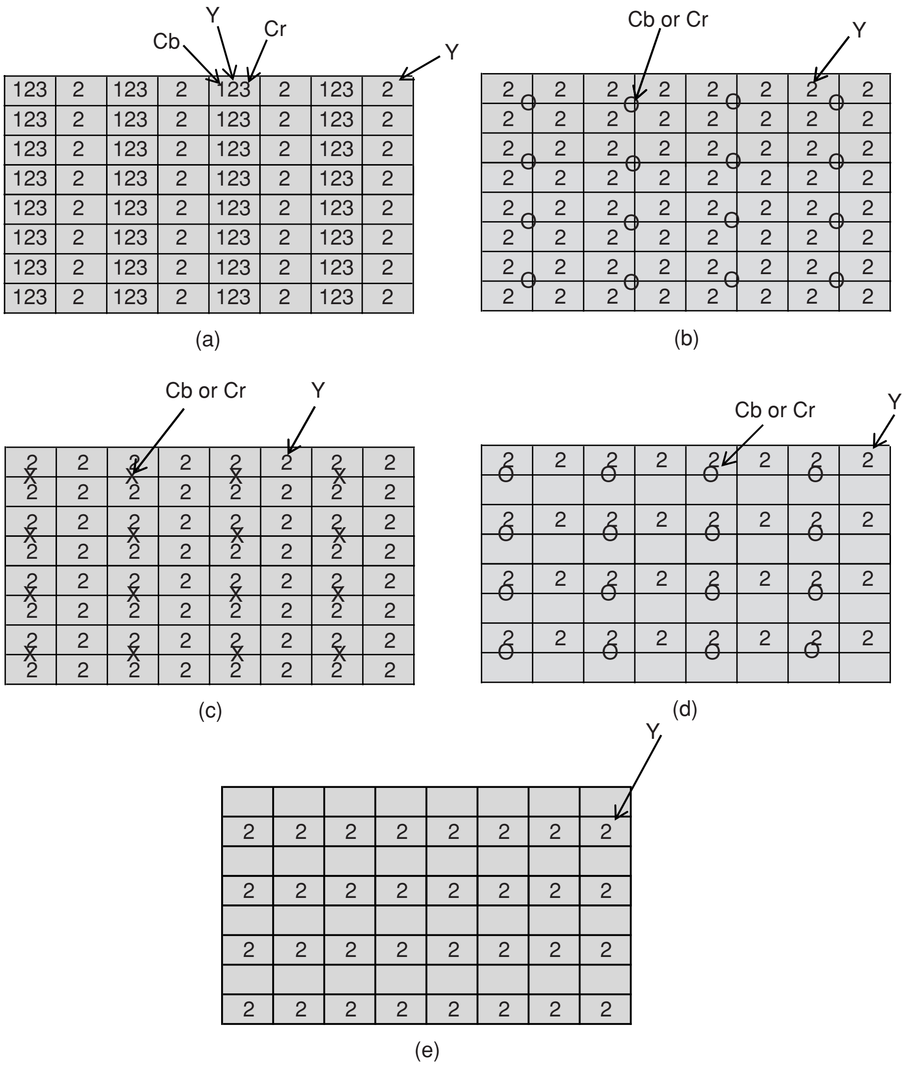


Figure 10.5 Chroma sampling format for MPEG-1 and MPEG-2 standards: (a) 4:2:2 sampling: numerals 1 and 3 refer to the Cb and Cr samples and 2 refers to the Y sample, (b) 4:2:0 sampling format for MPEG-1 in which the chroma samples are placed symmetrically between Y samples, (c) 4:2:0 sampling format for MPEG-2 where the chroma samples are aligned vertically between Y samples, (d) 4:2:0 sampling format for interlaced video in MPEG-2—the top field where the chroma samples coexist with Y samples, and (e) the bottom field where no chroma samples exist.

Table 10.1 MPEG default Luma quantization matrix

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

The reference macroblocks are obtained from the "previous frame" buffer. The MC-predicted block is added to the differential block from the inverse discrete cosine transform (IDCT) block and stored back in the "previous frame" buffer. This forms a closed loop at the coding side. Note that the same loop is also available at the decoder for the reconstruction of the video.

DCT This processing block accepts the differential pixel block as input and outputs the 2D DCT of the differential pixels. Although the luma block is of size 16×16 pixels, the 2D DCT is performed on each 8×8 pixels.

Quantization The process of quantizing the 2D DCT coefficients in the MPEG standards is the same as that used in the JPEG standard. For intracoding, the default quantization matrix used in MPEG-1 for the luma component is shown in Table 10.1. The distribution of the quantization step sizes in Table 10.1 corresponds roughly to the frequency response of the human visual system for the given viewing distance of approximately six times the screen width and picture resolution of 360×240 pixels [6]. The amount of quantization of the DCT coefficients can be increased or decreased by multiplying the quantization matrix by a *quantizer scale* parameter.

The default quantization matrix for non-intracoding in the MPEG standards 1 and 2 has the same value of 16 for all its elements as shown in Table 10.2. Remember that non-intracoding involves the coding of the differential images. The differential images are already decorrelated and are, therefore, essentially white noise. Hence,

Table 10.2 MPEG non-intracoding default quantization matrix

16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16

Table 10.3 MPEG-1 zigzag scanning of DCT coefficients: each entry represents the scan order of the coefficients

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

their DCTs will have approximately the same value for all the coefficients. That is the reason for using a constant value for the quantization matrix for non-intracoding.

Zigzag Scanning After the DCT coefficients have been quantized, the quantized coefficients are scanned in a zigzag order before entropy coding. The zigzag scanning orders for MPEG-1 and MPEG-2 are listed in Tables 10.3 and 10.4, respectively. The entries in the two tables correspond to the order of the coefficients scanned in a raster order. Figures 10.6a,b show the zigzag scanning pattern as a continuous line starting from coefficient 0 and ending in coefficient 63.

Variable-Length Coding

I-picture Coding

DC Coefficient

As mentioned earlier, I-pictures are coded by themselves without reference to previous or future pictures. This implies that I-picture coding is purely spatial. The DC and AC coefficients of the 8×8 DCTs are entropy coded differently after quantization and zigzag scanning. The reason for the difference in the coding of the DC and AC coefficients is as follows. The DC coefficient of an 8×8 DCT is a measure of the average value of that block; in fact, it is 8 times the block average. Because there

Table 10.4 MPEG-2 alternative zigzag scanning of DCT coefficients: each entry represents the scan order of the coefficients

0	4	6	20	22	36	38	52
1	5	7	21	23	37	39	53
2	8	19	24	34	40	50	54
3	9	18	25	35	41	51	55
10	17	26	30	42	46	56	60
11	16	27	31	43	47	57	61
12	15	28	32	44	48	58	62
13	14	29	33	45	49	59	63

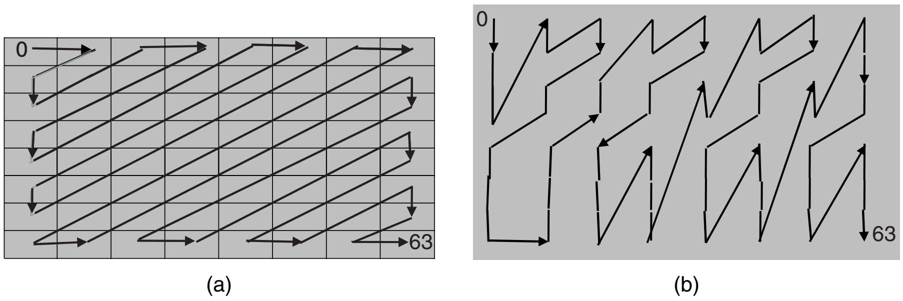


Figure 10.6 Zigzag patterns for scanning the quantized DCT coefficients: (a) patterns common to both MPEG-1 and MPEG-2 standards and (b) additional pattern used in MPEG-2.

is correlation left between the block averages, it makes sense to code the block DC differences rather than to code each block DC separately. As the overall objective in a compression scheme is to achieve as high a compression ratio as possible for a given amount of distortion or loss of information, it is necessary to remove pixel correlation wherever and whenever possible. Thus, the difference $\Delta_{DC} = DC - D\hat{C}$, where $D\hat{C}$ is the predicted value (DC coefficient of the preceding block), is entropy coded. MPEG uses the same procedure for coding the DC coefficients as used in JPEG. The DC difference is divided into 12 *size* categories, 0 through 11. Each size has exactly 2^{size} number of entries, as shown in Table 10.5. The Huffman code for each size is also listed alongside in the table. The table also lists the code for the DC size category of the chroma components. MPEG-1 has up to size 8 with DC difference value ranging from −255 to +255, while MPEG-2 has up to size 11 with corresponding range from −2047 to +2047. To code a DC difference Δ_{DC}, its size

Table 10.5 Huffman code for Δ_{DC} of Luma and Chroma components in MPEG

Luma Code	Chroma Code	Size	Range of Values
100	00	0	0
00	01	1	−1,1
01	10	2	−3 −2,2,3
101	110	3	−7. . ..−4,4. . ..7
110	1110	4	−15. . ..−8,8,. . ..15
1110	11110	5	−31. . ..−16,16. . ..31
11110	111110	6	−63. . .−32,32. . ..63
111110	1111110	7	−127. . ..−64,64. . ..127
1111110	11111110	8	−255. . ..−128,128. . ..255
1111 1110	1111 1111 0	9	−511 . . ..-256,256. . ..511
1111 1111 0	1111 1111 10	10	−1023. . ..−512,512. . ..1023
1111 1111 1	1111 1111 11	11	−2047. . ..−1024,1024. . ..2047

category is determined first from

$$\text{size} = \left\lceil \log_2 (|\Delta_{\text{DC}}| + 1) + 0.5 \right\rceil \tag{10.1}$$

Then the code for the DC difference is the code for its size category followed by *size* number of bits, which identifies the exact value of the DC difference. For instance, the size category of the DC difference value 12 in luma component is 4. The code for the size category 4 from Table 10.5 is **110**. Then there will be an additional 4 bits to be appended to the code for the size category. The 4-bit binary code for the decimal value 12 is **1100**. Therefore, the code for the DC difference 12 is **1101100**. If the DC difference is negative, then the additional *size* number of bits will correspond to the 2's complement of the DC difference minus one. Thus, the code for -12 is as follows. The size category is still the same, namely, 4. Then the 2's complement of $-12-1 = -13$ in 4-bit binary representation is **0011**. Hence, the code for the DC difference value -12 is **1100011**. Alternatively, we can determine the binary code to be used to represent -12 as follows: first find the value by subtracting 12 from $2^{\text{size}} - 1 = 2^4 - 1 = 15$, which is 3. Then represent 3 in size-bit binary code, which is **0011**. Note that positive DC differences will have a "1" as the most significant bit (MSB), while negative DC differences will have a "0" as the MSB.

AC Coefficients

As in JPEG, the majority of the AC coefficients in an 8×8 block have zero values after quantization. After zigzag scanning, one encounters a large run of zero-valued coefficients. It is more efficient to code zero runs than to code each zero value individually. It is found that there is a certain probability associated with the pair consisting of a zero-run and a nonzero coefficient that breaks the zero run. MPEG codes the run/level pair instead of just the zero run and nonzero coefficient values separately. Here, run is the length of zero run and level is the magnitude of the nonzero coefficient that breaks the zero run. Often one encounters a set of zero-valued coefficients all the way to the end of a block. Since there is no nonzero coefficient to break the zero run, this run of zeros is coded as the end of block (EOB). Table 10.6 lists the variable-length codes (VLCs) for various run/level values used in MPEG. There are actually two tables with some entries different from each other but with majority of codes being identical [6]. It should be pointed out that in intracoding of AC coefficients, only the second code for run/level 0/1 in Table 10.6 is used. The reason for this is that in intracoding, an EOB can occur right after the DC term in which case the first run/level code will conflict with the EOB code in Table A. Note that not all possible run/level values are included in the list. Those run/level values not listed in the table are coded by a 6-bit Escape code followed by a 6-bit binary code for the run-length and a 12-bit binary code for the nonzero coefficient level. Even though these run/level values use more bits than other run/level values that are found in the table, they are highly unlikely and so do not increase the overall bit rate in any significant manner. In each code, "s"

Table 10.6 MPEG VLC for run/level pairs in an 8 × 8 DCT

Run/Level	Table A	Length	Table B	Length
0/1	1s (first code)	2	10s	3
0/1	11s (second code)	3	10s	3
0/2	0100s	5	110s	4
0/3	0010 1s	6	0111s	5
0/4	0000 110s	8	1110 0s	6
0/5	0010 0110s	9	1110 1s	6
0/6	0010 0001s	9	0001 01s	7
0/7	0000 0010 10s	11	0001 00s	7
0/8	0000 0001 1101s	13	1111 011s	8
0/9	0000 0001 1000s	13	1111 100s	8
0/10	0000 0001 0011s	13	0010 0011s	9
0/11	0000 0001 0000s	13	0010 0010s	9
0/12	0000 0000 1101 0s	14	1111 1010s	9
0/13	0000 0000 1100 1s	14	1111 1011s	9
0/14	0000 0000 1100 0s	14	1111 1110s	9
0/15	0000 0000 1011 1s	14	1111 1111s	9
0/16	0000 0000 0111 11s	15	0000 0000 0111 11s	15
0/17	0000 0000 0111 10s	15	0000 0000 0111 10s	15
0/18	0000 0000 0111 01s	15	0000 0000 0111 01s	15
0/19	0000 0000 0111 00s	15	0000 0000 0111 00s	15
0/20	0000 0000 0110 11s	15	0000 0000 0110 11s	15
0/21	0000 0000 0110 10s	15	0000 0000 0110 10s	15
0/22	0000 0000 0110 01s	15	0000 0000 0110 01s	15
0/23	0000 0000 0110 00s	15	0000 0000 0110 00s	15
0/24	0000 0000 0101 11s	15	0000 0000 0101 11s	15
0/25	0000 0000 0101 10s	15	0000 0000 0101 10s	15
0/26	0000 0000 0101 01s	15	0000 0000 0101 01s	15
0/27	0000 0000 0101 00s	15	0000 0000 0101 00s	15
0/28	0000 0000 0100 11s	15	0000 0000 0100 11s	15
0/29	0000 0000 0100 10s	15	0000 0000 0100 10s	15
0/30	0000 0000 0100 01s	15	0000 0000 0100 01s	15
0/31	0000 0000 0101 00s	15	0000 0000 0101 00s	15
0/32	0000 0000 0011 000s	16	0000 0000 0011 000s	16
0/33	0000 0000 0010 111s	16	0000 0000 0010 111s	16
0/34	0000 0000 0010 110s	16	0000 0000 0010 110s	16
0/35	0000 0000 0010 101s	16	0000 0000 0010 101s	16
0/36	0000 0000 0010 100s	16	0000 0000 0010 100s	16
0/37	0000 0000 0010 011s	16	0000 0000 0010 011s	16
0/38	0000 0000 0010 010s	16	0000 0000 0010 010s	16
0/39	0000 0000 0010 001s	16	0000 0000 0010 001s	16
0/40	0000 0000 0010 000s	16	0000 0000 0010 000s	16
1/1	011s	4	010s	4
1/2	0001 10s	7	0011 0s	6

(*Continued*)

Table 10.6 (*Continued*)

Run/Level	Table A	Length	Table B	Length
1/3	0010 0101s	9	1111 001s	8
1/4	0000 0011 00s	11	0010 0111s	9
1/5	0000 0001 1011s	13	0010 0000s	9
1/6	0000 0000 1011 0s	14	0000 0000 1011 0s	14
1/7	0000 0000 1010 1s	14	0000 0000 1010 1s	14
1/8	0000 0000 0011 111s	16	0000 0000 0011 111s	16
1/9	0000 0000 0011 110s	16	0000 0000 0011 110s	16
1/10	0000 0000 0011 101s	16	0000 0000 0011 101s	16
1/11	0000 0000 0011 100s	16	0000 0000 0011 100s	16
1/12	0000 0000 0011 011s	16	0000 0000 0011 011s	16
1/13	0000 0000 0011 010s	16	0000 0000 0011 010s	16
1/14	0000 0000 0011 001s	16	0000 0000 0011 001s	16
1/15	0000 0000 0001 0011s	17	0000 0000 0001 0011s	17
1/16	0000 0000 0001 0010s	17	0000 0000 0001 0010s	17
1/17	0000 0000 0001 0001s	17	0000 0000 0001 0001s	17
1/18	0000 0000 0001 0000s	17	0000 0000 0001 0000s	17
2/1	0101s	5	0010 1s	6
2/2	0000 100s	8	0000 111s	8
2/3	0000 0001 11s	11	1111 1100s	9
2/4	0000 0001 0100s	13	0000 0011 00s	11
2/5	0000 0000 1010 0s	14	0000 0000 1010 0s	14
3/1	0011 1s	6	0011 1s	6
3/2	0010 0100s	9	0010 0110s	9
3/3	0000 0001 1100s	13	0000 0001 1100s	13
3/4	0000 0000 1001 1s	14	0000 0000 1001 1s	14
4/1	0011 0s	6	0001 10s	7
4/2	0000 0011 11s	11	1111 1101s	9
4/3	0000 0001 0010s	13	0000 0001 0010s	13
5/1	0001 11s	7	0001 11s	7
5/2	0000 0010 01s	11	0000 0010 0s	10
5/3	0000 0000 1001 0s	14	0000 0000 1001 0s	14
6/1	0001 01s	7	0000 110s	8
6/2	0000 0001 1110s	13	0000 0001 1110s	13
6/3	0000 0000 0001 0100s	17	0000 0000 0001 0100s	17
7/1	0001 00s	7	0000 100s	8
7/2	0000 0001 0101s	13	0000 0001 0101s	13
8/1	0000 111s	8	0000 101s	8
8/2	0000 0001 0001s	13	0000 0001 0001s	13
9/1	0000 101s	8	1111 000s	8
9/2	0000 0000 1000 1s	14	0000 0000 1000 1s	14
10/1	0010 0111s	9	1111 010s	8
10/2	0000 0000 1000 0s	14	0000 0000 1000 0s	14
11/1	0010 0011s	9	0010 0001s	9

Table 10.6 (*Continued*)

Run/Level	Table A	Length	Table B	Length
11/2	0000 0000 0001 1010s	17	0000 0000 0001 1010s	17
12/1	0010 0010s	9	0010 0110s	9
12/2	0000 0000 0001 1001s	17	0000 0000 0001 1001s	17
13/1	0010 0000s	9	0010 0100s	9
13/2	0000 0000 0001 1000s	17	0000 0000 0001 1000s	17
14/1	0000 0011 10s	11	0000 0010 1s	10
14/2	0000 0000 0001 0111s	17	0000 0000 0001 0111s	17
15/1	0000 0011 01s	11	0000 0011 1s	10
15/2	0000 0000 0001 0110s	17	0000 0000 0001 0110s	17
16/1	0000 0010 00s	11	0000 0011 01s	11
16/2	0000 0000 0001 0101s	17	0000 0000 0001 0101s	17
17/1	0000 0001 1111s	13	0000 0001 1111s	13
18/1	0000 0001 1010s	13	0000 0001 1010s	13
19/1	0000 0001 1001s	13	0000 0001 1001s	13
20/1	0000 0001 0111s	13	0000 0001 0111s	13
21/1	0000 0001 0110s	13	0000 0001 0110s	13
22/1	0000 0000 1111 1s	14	0000 0000 1111 1s	14
23/1	0000 0000 1111 0s	14	0000 0000 1111 0s	14
24/1	0000 0000 1110 1s	14	0000 0000 1110 1s	14
25/1	0000 0000 1110 0s	14	0000 0000 1110 0s	14
26/1	0000 0000 1101 1s	14	0000 0000 1101 1s	14
27/1	0000 0000 0001 1111s	17	0000 0000 0001 1111s	17
28/1	0000 0000 0001 1110s	17	0000 0000 0001 1110s	17
29/1	0000 0000 0001 1101s	17	0000 0000 0001 1101s	17
30/1	0000 0000 0001 1100s	17	0000 0000 0001 1100s	17
31/1	0000 0000 0001 1011s	17	0000 0000 0001 1011s	17
EOB	10	2	0110	4
Esc	0000 01	6	0000 01	6

refers to the sign bit. It is “0” for positive coefficient value and “1” for negative coefficient value.

To elucidate the idea of entropy coding of the AC coefficients in I-pictures, consider the 8×8 pixel block of an intensity image shown in Table 10.7. Table 10.8 lists the pixel block after subtracting 128 from each pixel. The 2D DCT of the block in Table 10.8 is listed in Table 10.9. The DCT coefficients after quantization using the MPEG default matrix in Table 10.1 are listed in Table 10.10. After zigzag scanning the quantized DCT coefficients using the pattern in Table 10.4, we get the following run/level pairs as shown below.

Run/level pairs obtained from zigzag scanning the quantized DCT coefficients of Table 10.10: DC, (0, −3), (0, −5), (0, −2), (0, −7), (0, 4), (0, −8), (0, −1), (0, 1), (0,1), (0, −1), (0, 1), (0, 3), (1, −2), (0, −1), (0, 1), (1, 1), (0, 4), (1, −3), (0, 2), (0, −3), (0, 3), (1, −1), (1, 1), (0, 1), (0, 1), (0, −1), (3, 1), (0, 2), (0, 1), (1, −1), Table 10.7 8 × 8 pixel block of an intensity image

130	115	137	137	125	113	109	113
229	100	50	54	51	66	95	91
179	220	108	97	65	51	64	61
59	179	220	153	115	89	67	55
105	65	185	222	126	126	108	77
101	92	61	190	216	115	97	87
129	135	99	64	196	215	133	124
131	130	126	97	62	194	211	129

(2, −1), (0, 2), (0, −1), (1, −1), (0, −1), (0, −1), (3, −1), (0, 1), EOB. Then the codes for the run/level pairs of the AC coefficients are obtained from Table 10.6 and are **001011 001001101 01001 00000010101 00001100** ... **10**. Note that the sign bit is a “0” for positive coefficient value and a “1” for negative coefficient value.

P- and B-Picture Coding Predictive and bidirectionally predictive pictures are non-intracoded using DCT. It is a known fact that temporal prediction removes correlation between temporally adjacent pictures. The predicted pictures generally have zero mean value and the amplitudes are distributed in a Laplacian-like manner. Therefore, applying 2D DCT on a block of pixels in a predicted picture does not anymore decorrelate the block. However, quantizing the DCT coefficients of a predictive block is more efficient from a compression point of view than quantizing the individual predictive pixel.

Unlike in the I-picture coding, the quantized DC and AC coefficients of a predictive block are coded using the same Huffman codebook (Table 10.6—column Table A). That is, there is no separate codebook for DC coefficients. As mentioned earlier, the quantization matrix for the P- and B-pictures has the same value of 16 for all its elements [6].

In P- and B-picture coding, there is a finite probability that a whole macroblock is zero after quantization. In case when a completely zero macroblock is encountered, it is coded by an address increment so that the decoder simply skips that particular macroblock. If all blocks in a macroblock are not zero, then the blocks within the macroblock that are zero are coded using the *code block pattern* [6]. The code block

Table 10.8 8 × 8 pixel block after subtracting 128

2	−13	9	9	−3	−15	−19	−15
101	−28	−78	−74	−77	−62	−33	−37
51	92	−20	−31	−63	−77	−64	−67
−69	51	92	25	−13	−39	−61	−73
−23	−63	57	94	−2	−2	−20	−51
−27	−36	−67	62	88	−13	−31	−41
1	7	−29	−64	68	87	5	−4
3	2	−2	−31	−66	66	83	1

Table 10.9 DCT of the pixel block in Table 10.8

7	−119	−155	5	39	5	3	5
−46	61	−27	−72	−82	−23	7	−4
−97	21	108	86	50	−19	−44	−10
−46	−27	27	7	16	70	49	8
−17	−4	−27	−27	−20	−43	−14	9
14	14	−3	−16	0	8	−7	−17
70	−17	30	16	−19	−8	3	4
−2	−66	46	29	−31	−14	4	6

pattern is a 6-bit VLC in which each bit says whether a particular block is completely zero or not. If there is a nonzero block within a macroblock, the DCT coefficients of that block are entropy coded using the Table 10.6 (Table A) with all entries included.

The block diagram in Figure 10.4 is suitable for I- and P-picture coding. When coding involves B-pictures, there is a need for reordering of pictures. Remember that B-picture coding involves both previous and future pictures. Thus, if the GOP follows the pattern IBBPBBP. . ., then the first picture is intra or I-picture, fourth picture is P-picture, second and third pictures are B-pictures, and so on. This requires frame store and reordering of the pictures at the encoder. Similarly, while displaying the decompressed video, frame store and reordering are necessary. Figure 10.7 is a block diagram illustration of B-picture coding.

Rate Control The output bit rate of the VLC in the MPEG coder is variable because the activities in the macroblocks vary within a picture and as a result the quantized DCT coefficients generate different bit rates. There are applications such as digital television (DTV) that require a constant bit rate for real-time transmission. Therefore, it is necessary to control the output bit rate of the MPEG coder so as to produce a constant bit rate into the transmission system [10–13]. This in turn requires an output buffer, which acts like a capacitor to smooth out the bit rate variability. In applications such as DVD that records rather than transmits a program, the coder output is necessarily variable and the objective is to vary the instantaneous bit rate to achieve a given picture quality and minimize storage space requirement. There is a

Table 10.10 Quantized DCT using the matrix in Table 10.1

1	−7	−8	0	2	0	0	0
−3	4	−1	−3	−3	−1	0	0
−5	1	4	3	2	−1	−1	0
−2	−1	1	0	1	2	1	0
−1	0	−1	−1	−1	−1	0	0
1	1	0	0	0	0	0	0
3	−1	1	0	−1	0	0	0
0	−2	1	1	−1	0	0	0

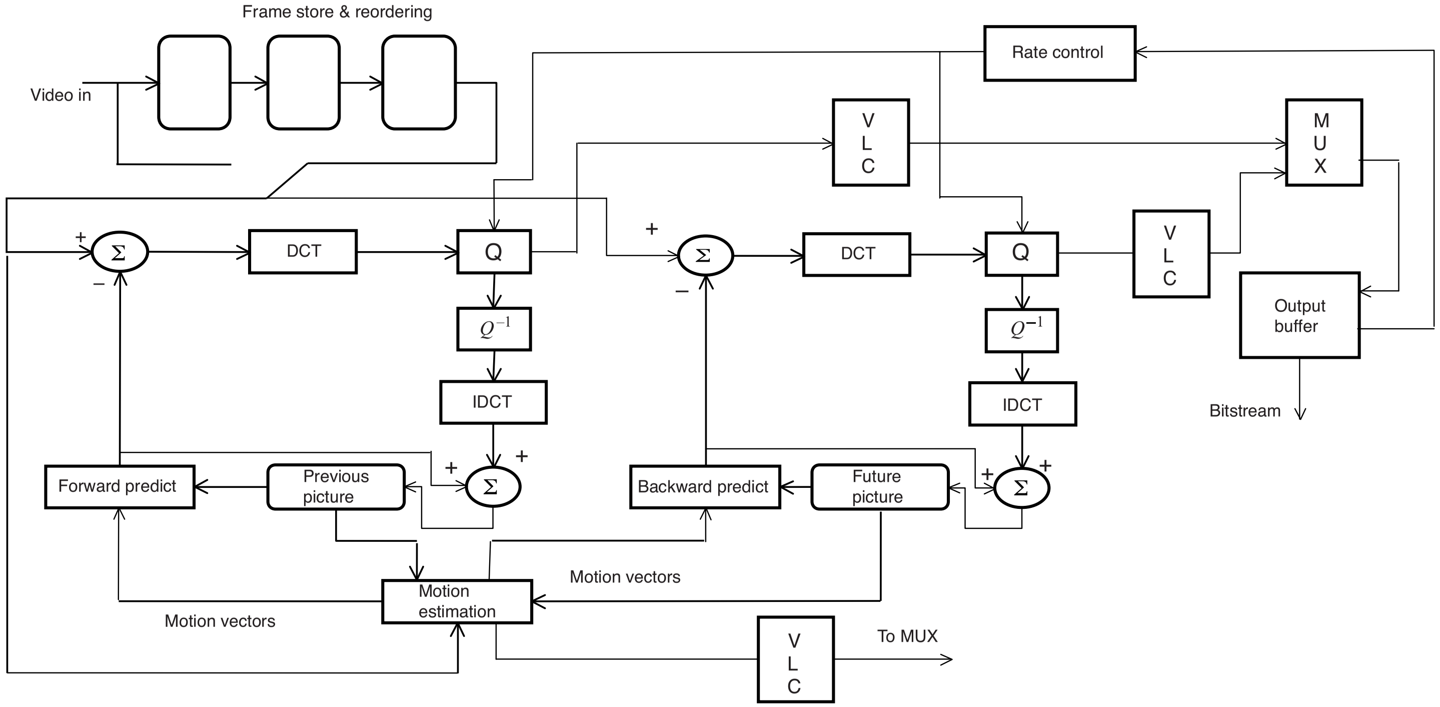


Figure 10.7 Block diagram illustrating B-picture coding in MPEG standards.

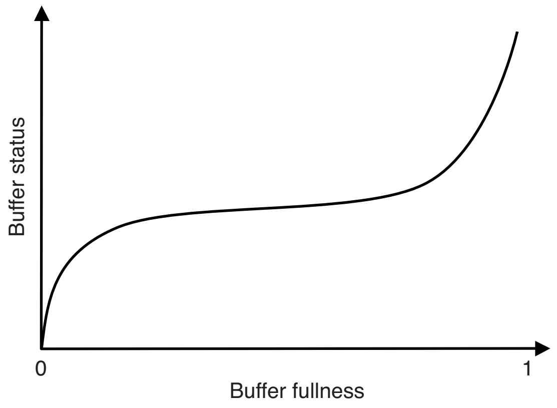


Figure 10.8 A typical S-shaped relationship between buffer fullness and buffer status in an MPEG rate controller.

conflicting situation. If the decoder buffer size is large, then there is more freedom for the encoder to allocate varying bits to the pictures to maintain a higher visual quality. However, a larger buffer at the decoder is more expensive especially when hundreds of thousands of decoders are in use. Thus, there is a tradeoff between decoder buffer size and the flexibility at the coder. In fact, the rate control strategy is much more complicated than meets the eye.

In the heart of the rate control mechanism lies the quantizer scale parameter. Increasing the quantizer scale increases the quantization step sizes, which in turn reduces the bit rate. Conversely, decreasing the quantizer scale increases the bit rate [14]. MPEG-2 allows changing the quantizer scale parameter at macroblock, slice, and picture levels. The coder senses the buffer status based on the fullness of the output buffer and accordingly adjusts the quantizer scale parameter at the appropriate level in the picture. If the buffer is nearly full, the quantizer scale is increased sharply to quickly bring down the rate. When the buffer is nearly empty, the quantizer scale is decreased rather sharply to increase the bit rate. A typical relationship between the buffer fullness and the buffer status is an S-shaped curve, as shown in Figure 10.8.

Spatial Masking

Variable quantization may be effected on the basis of spatial and temporal activities at macroblock level. The human visual perception is such that it is less sensitive to quantization noise in active regions. Sensitivity is the inverse of threshold. A large quantization noise level is required in active regions for it to be noticed. Here, the quantization noise is the stimulus and the spatially activity is the mask. Thus, spatial active regions can be quantized heavily without the possibility of noticing such distortions. Larger threshold implies larger quantization steps, which in turn implies larger quantizer scale. In order to adjust the quantizer scale, one must first determine the activity of the region being considered and then determine the quantizer scale [15–20].

There are various ways to determine the activity of a given region. One method is to use the variance of the region. Another method is to use the local contrast relative to its neighborhood [21, 22]. Local contrast C_{local} may be defined as

$$C_{local} = \frac{|\mu_O - \mu_B|}{\mu_O + \mu_B} \tag{10.2}$$

where μ_O and μ_B are the mean intensities of the macroblock in question and its neighborhood, respectively. The neighborhood size is variable.

Temporal Masking

In coding a video sequence one encounters temporal masking phenomenon [23–27]. It is hard to notice the distortions due to high compression if the motion between frames is large. Thus, the quantization noise that is a temporal stimulus is masked by large motion in a video sequence. This phenomenon allows us to increase the quantizer scale value in high motion frames, thereby decreasing the bit rate and vice versa.

10.2.3 MPEG-2 Levels, Profiles, and Scalability

Levels and Profiles Because there are numerous possible applications ranging from streaming video to DVD to HDTV, MPEG-2 is divided into *profiles* and *levels*. It may not be economically viable nor may be necessary to implement all possible coding options. Hence, each encoder may be designed to perform a portion of the available options [28]. Table 10.11 lists the various profiles and levels for MPEG-2.

Table 10.11 Profiles and levels in MPEG-2

	Profiles				
Levels	Simple	Main	SNR	Spatial	High
High		4:2:0 1920 × 1152 90 Mb/s			4:2:0 or 4:2:2 1920 × 1152 100 Mb/s
High-1440		4:2:0 1440 × 1152 60 Mb/s		4:2:0 1440 × 1152 60 Mb/s	4:2:0 or 4:2:2 1440 × 1152 80 Mb/s
Main	4:2:0 720 × 576 15 Mb/s No B-pictures	4:2:0 720 × 576 15 Mb/s	4:2:0 720 × 576 15 Mb/s		4:2:0 or 4:2:2 720 × 576 20 Mb/s
Low		4:2:0 352 × 288 4 Mb/s	4:2:0 352 × 288 4 Mb/s		

MPEG-2 has five profiles, which are simple profile (SP), main profile (MP), SNR scalable profile (SNR), spatially scalable profile (Spt), and high profile (HP). The SP has only I- and P-picture coding, while the MP supports I-, P-, and B-picture coding. The intracoding precision for the quantized DC coefficient can be 8, 9, or 10 bits, while MPEG-1 has only 8-bit precision.

MPEG-2 defines four levels, namely, low level (LL), main level (ML), high-1440 (H-1440), and high level (HL). The simple profile at main level (SP@ML) has 720 samples per line with 576 total lines; 30 frames per second. With 480 active lines, this profile generates samples at the rate of 10,369,000 per second. The encoder bit rate is 15 Mb/s and the VBV (variable buffer verifier) buffer size is 1,835,008 bits.

Scalability A single program may be broadcast in different formats, such as SDTV and HDTV. These two end users have different quality and resolution. There must be an encoding/decoding mechanism, whereby both end users can receive the same broadcast and deliver the respective quality/resolution video. The idea is to transport layers of compressed data consisting of a *base* layer followed by one or more *enhancement* layers. Decoding the base layer will deliver lower quality/resolution video, and decoding additional layers will produce the higher quality/resolution video. Because the base layer carries a larger part of the compressed data and the enhancement layers carry only incremental data, the encoding becomes much more efficient. Otherwise we must employ separate coder for each of the intended application. The ability to encode/decode a video at varying quality/resolution is termed *scalability*. MPEG-2 allows three types of scalability, namely, SNR, spatial, and high (refer to Table 10.11). We will briefly describe each of the scalable schemes as follows.

SNR Scalability Figure 10.9 shows qualitatively how the compressed bit stream is scalable to achieve pictures of different quality. Here, quality is associated with the

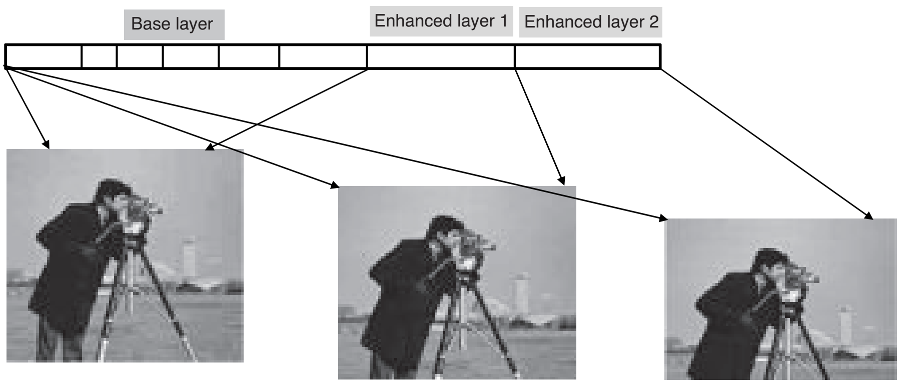


Figure 10.9 A qualitative way of showing SNR scalability from an MPEG-2 bit stream.

signal-to-noise ratio (SNR). By decoding only the base layer, a lower quality image is obtained. By decoding the base layer and an additional enhancement layer, we get the same spatial resolution picture but with a better quality. By decoding all the data, we obtain the highest quality picture [28]. In the context of MPEG-2, a typical SNR scalable encoder using DCT is shown in Figure 10.10a. The base layer consists of the bit stream generated by quantizing the DCT coefficients of the prediction errors with quantization Q_1. The difference between the actual and quantized/dequantized DCT coefficients at the encoder is quantized by quantization Q_2 and transmitted as the enhancement layer. The SNR scalable decoder shown in Figure 10.10b decodes the base layer with inverse quantization Q_1^{-1} and delivers a base quality video. If both layers are decoded in the manner shown in Figure 10.10b, one obtains the higher quality video.

Example 10.1 In this example, let us demonstrate the idea of SNR scalability by using an actual video data. Specifically, we read the video named “rhinos.avi.” This video is MATLAB supplied and is in AVI file format. The MATLAB function “aviread (‘rhinos.avi’)” will read the entire video into a structure. The pictures are RGB and of size 352 × 240 pixels. This example uses one base layer and one enhancement layer. Also, this example codes the luma component only.

The first picture is intracoded and stored in the buffer. Subsequent pictures are predictive coded with a quantization step size of 16 and a quantizer scale of 4 for all the differential DCT coefficients. This forms the base layer. The difference between the actual DCT and the quantized/dequantized DCT in the base layer is then quantized with a quantization step of 4 for all the coefficients. This additional data is the enhancement layer. Both base and enhancement layers are variable length coded and transmitted or stored. For lower quality video, only the base layer is decoded. To obtain a higher quality video, both layers are decoded.

The base quality and higher quality pictures from the decoded video are shown in Figures 10.11a,b, respectively. As can be seen from the figures, the base quality picture has a lot of quantization distortions especially on the automobile, while the enhanced layer picture has no noticeable distortions. This fact is also evident from the SNR and peak signal-to-noise ratio (PSNR) values, which are shown in Figures 10.12a,b, respectively. These values are calculated only for nine frames for want of central processing unit (CPU) time and storage. Note that the quantizer scale of 4 is chosen to generate a large disparity between the base and enhancement layers. In practice, it depends on the end users.

The same trend in the quality of the base and enhancement layers is seen for other video sequences, as well. Figures 10.13a and b show the SNR and PSNR values for the Table Tennis sequence using frames 41 through 49.

The following is a listing of the MATLAB code to perform SNR scalable coding. It does not calculate the bit rate nor does it generate the VLC codes. The aim is to illustrate the idea of SNR scalability. However, the codes follow the P-picture coding and quantization rules of MPEG-2.

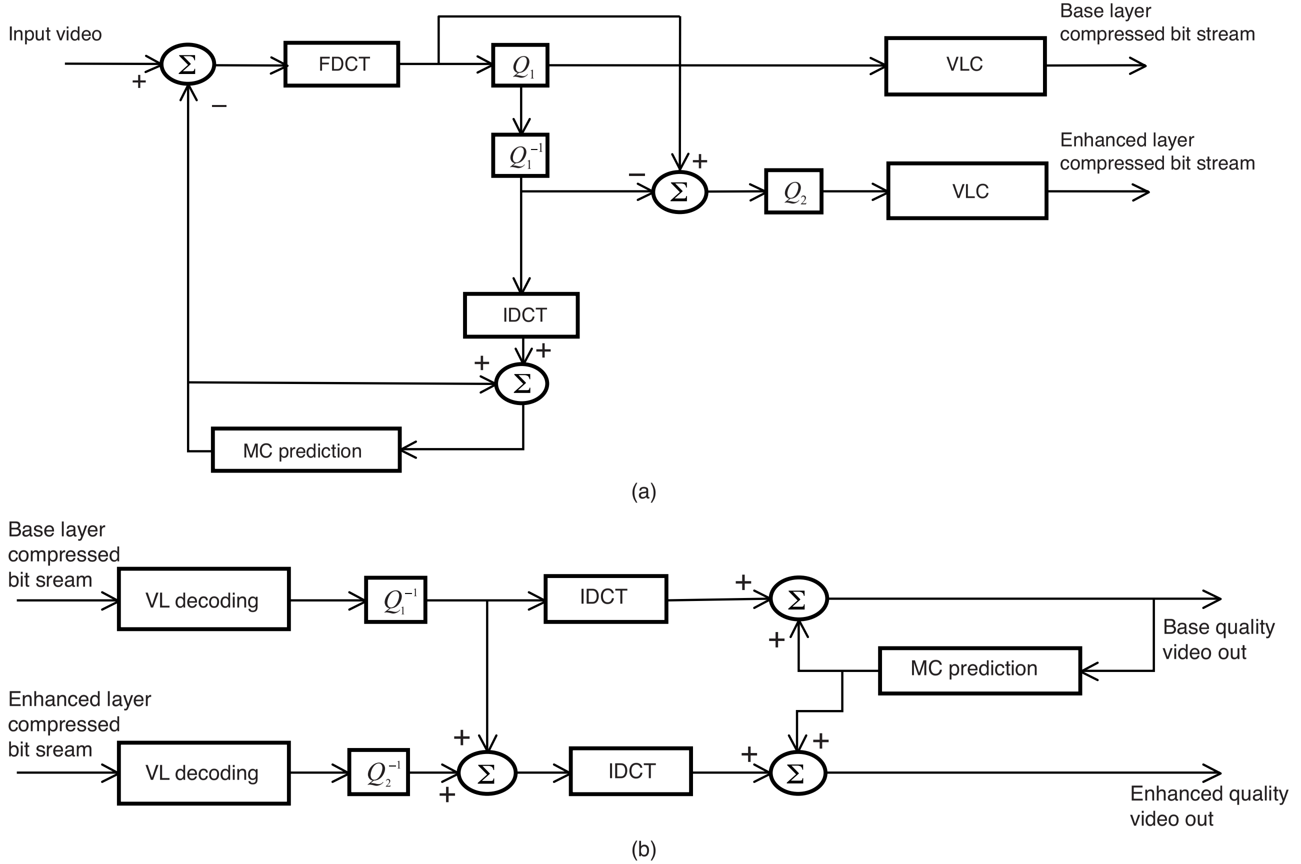
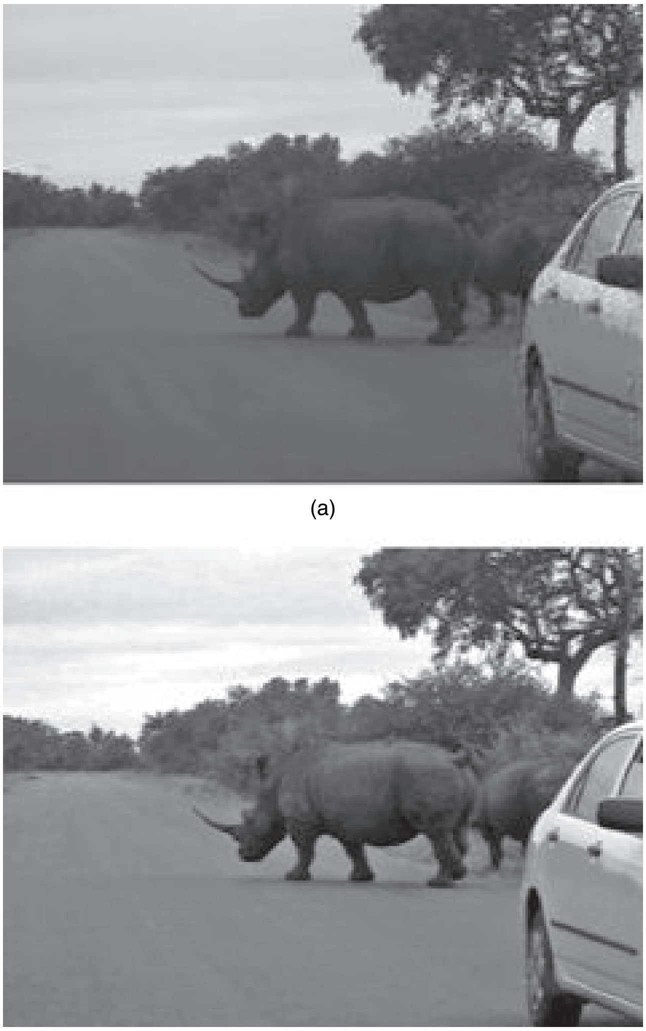

Input video
FDCT
Q_1
Q_1^{-1}
Q_2
VLC
VLC
Base layer compressed bit stream
Enhanced layer compressed bit stream
IDCT
MC prediction
(a)
Base layer compressed bit sream
Enhanced layer compressed bit sream
VL decoding
VL decoding
Q_1^{-1}
Q_2^{-1}
IDCT
IDCT
MC prediction
Base quality video out
Enhanced quality video out
(b)

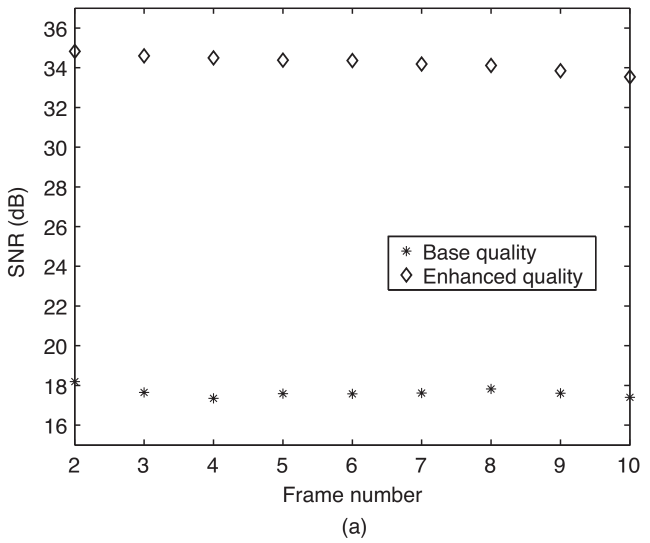
Figure 10.10 A block diagram showing SNR scalable DCT-based MPEG video coding: (a) encoder and (b) decoder.



(a)

(b)

Figure 10.11 Decoded pictures using SNR scalable MPEG video coding: (a) base quality picture and (b) enhanced quality picture. Input video is the rhinoceros AVI clip. The pictures (a) and (b) correspond to second and third frames in the video sequence.



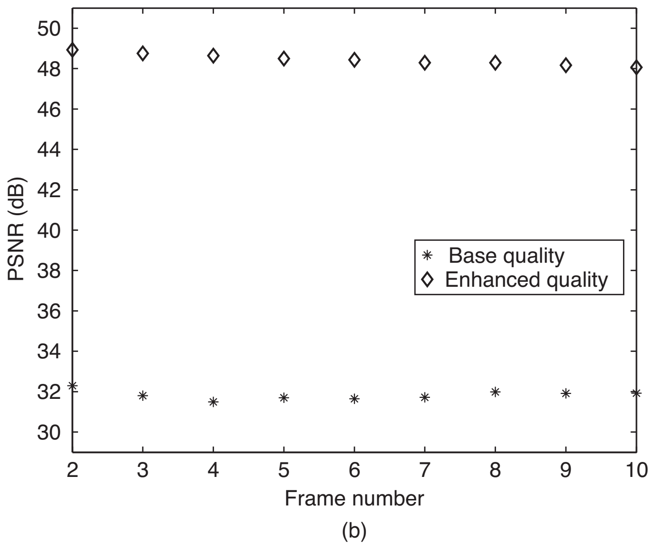
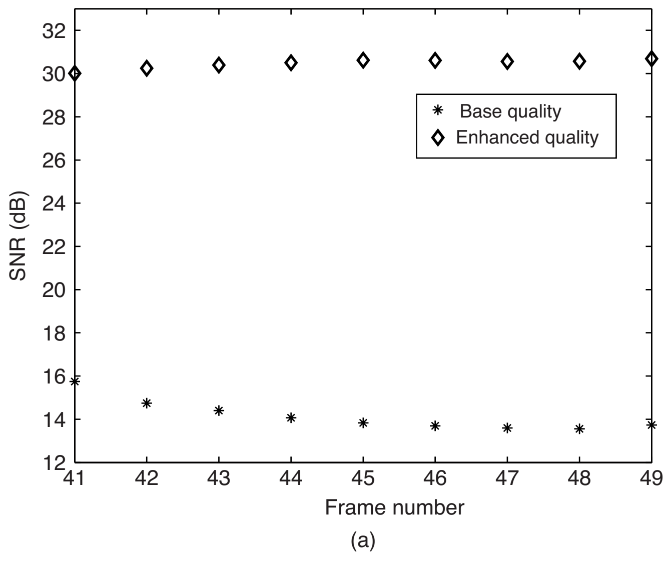


Figure 10.12 SNR and PSNR values for the base and enhanced quality pictures of Example 10.1 for SNR scalability. The values are for frames 2 through 10, inclusive. (a) SNR values in dB, (b) PSNR values in dB.



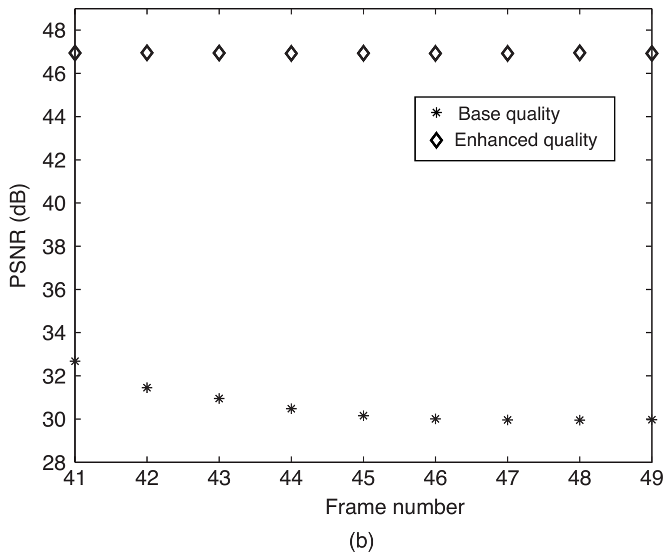


Figure 10.13 Same SNR and PSNR values as in Figure 10.12 for the Table Tennis sequence: The values are for frames 41 through 49, inclusive: (a) SNR in dB and (b) PSNR in dB.

```
% Example10_1.m
% SNR scalable video coding using motion compensated prediction
% Block motion is estimated to an integer pel accuracy
% using full search and SAD matching metric.
% Block size is 8 x 8 pixels.
% Search window size is 16 x 16 pixels.
% The differential block is 2D DCT transformed, quantized
% using uniform quantizer with constant quantization step of 16.
% The reconstructed block becomes the reference block for
% the next input frame.
% Base layer carries the above quantized data.
% The enhanced layer is created by coding the difference
% between the unquantized DCT coefficients and the base layer
% quantized DCT coefficients and then quantizing this differential
% DCT with a different quantization step.
% Decoding the base layer yields a lower quality video and
% adding the enhanced layer to the base layer results in a
% higher quality video.
% Only intensity (Luma) image sequence is accepted.
% Note: Both Table Tennis and Trevor sequences have ".ras"
% extension while the Claire sequence has ".yuv" extension.


clear
N = 8;% block size is N x N pixels
N2 = 2*N;
W = 16; % search window size is W x W pixels
quantizer_scale = 4; % used only for the base layer
% Four different sequences are tested.
% "rhinos.avi" exists in MATLAB and the others do not.
%inFile = 'tt040.ras';% Table Tennis sequence
inFile = 'twy040.ras'; % Trevor sequence
%inFile = 'clairey040.yuv'; % Claire sequence
%inFile = 'rhinos.avi'; % AVI file from MATLAB
strIndx = regexp(inFile,'\.');
if strcmpi('avi',inFile(strIndx+1:strIndx+3))
    M = aviread(inFile); % There are 114 320x240x3 frames
    F = int16(1:10);% frames to encode
else
    F = int16(40:49);% frames to encode
    M.cdata = cell(1,length(F));
    for k = 1:length(F)
        strIndx1 = regexp(inFile,'0');
        inFile1 = strcat(inFile(1:strIndx1(1)),num2str(F(k)),...
            inFile(strIndx:end));
        M(k).cdata = imread(inFile1);
    end
end
```

```
% Make image size divisible by 8
[X,Y,Z] = size(M(1).cdata);
if mod(X,8)~=0
    Height = floor(X/8)*8;
else
    Height = X;
end
if mod(Y,8)~=0
    Width = floor(Y/8)*8;
else
    Width = Y;
end
Depth = Z;
clear X Y Z
%
if Depth == 3
    A = rgb2ycbcr(M(1).cdata);% Convert RGB to YCbCr & retain only Y
    y_ref = A(:,:,1);
else
    A = M(1).cdata;
    y_ref = A;
end
% pad the reference frame left & right and top & bottom
y_ref = double(padarray(y_ref,[W/2 W/2],'replicate'));
% arrays to store SNR and PSNR values
Base_snr = zeros(1,length(F)-1); Enhanced_snr = zeros(1,length(F)-1);
Base_psnr = zeros(1,length(F)-1); Enhanced_psnr = zeros(1,length(F)-1);
% Encode the monochrome video using MPC
for f = 2:length(F)
    if Depth == 3
        B = rgb2ycbcr(M(f).cdata);
        y_current = B(:,:,1);
    else
        y_current = M(f).cdata;
    end
    y_current = double(padarray(y_current,[W/2 W/2],'replicate'));
    for r = N:N:Height
        rblk = floor(r/N);
        for c = N:N:Width
            cblk = floor(c/N);
            D = 1.0e+10;% initial city block distance
            for u = -N:N
                for v = -N:N
                    d = y_current(r+1:r+N,c+1:c+N)-y_ref(r+u+1:
                        r+u+N,c+v+1:c+v+N);
                    d = sum(abs(d(:)));% city block distance
                        between pixels
                    if d < D
                        D = d;
                        x1 = v; y1 = u; % motion vector
                    end
                end
            end
            % MC compensated difference coding
            temp = y_current(r+1:r+N,c+1:c+N)...
                -y_ref(r+1+y1:r+y1+N,c+1+x1:c+x1+N);
            TemP = dct2(temp); % DCT of difference
            s = sign(TemP); % extract the coefficient sign
            TemP1 = s .* round(abs(TemP)/(16*quantizer_scale))...
                *(16*quantizer_scale); % quantize/dequantize DCT
            temp = idct2(TemP1); % IDCT
            Base(r-N+1:r,c-N+1:c) = y_ref(r+1+y1:r+y1+N,c+1+x1:c+x1+N)
                                    +...
                temp; % reconstructed block - base quality
            delta_DCT = TemP - TemP1; % incremental DCT
            s1 = sign(delta_DCT); % extract the sign of
                                    incremental DCT
            delta_DCT = s1 .* round(abs(delta_DCT)/...
                4)*4;
            temp1 = idct2(TemP1 + delta_DCT);
            Enhanced(r-N+1:r,c-N+1:c) = y_ref(r+1+y1:r+y1+N,c+1+x1:
                                        c+x1+N) +...
                temp1;
        end
    end
    % Calculate the respective SNRs and PSNRs
    Base_snr(f-1) = 20*log10(std2(y_current(N+1:Height+N,N+1:Width+N))
                    /...
        std2(y_current(N+1:Height+N,N+1:Width+N)-Base));
    Enhanced_snr(f-1) = 20*log10(std2(y_current(N+1:Height+N,N+1:
                        Width+N))/...
        std2(y_current(N+1:Height+N,N+1:Width+N)-Enhanced));
    Base_psnr(f-1) = 20*log10(255/std2(y_current(N+1:Height+N,N+1:
                     Width+N)-Base));
    Enhanced_psnr(f-1) = 20*log10(255/std2(y_current(N+1:Height+N,N+1:
                         Width+N)...
        -Enhanced));
    % replace previous frames by the currently reconstructed frames
    y_ref = Base;
    y_ref = double(padarray(y_ref,[W/2 W/2],'replicate'));
end
figure,plot(F(2:end),Base_snr,'k*','LineWidth',1), hold on
plot(F(2:end),Enhanced_snr,'kd','LineWidth',2), title('SNR (dB)')
axis([F(2) F(end) min(Base_snr)-2 max(Enhanced_snr)+2]) % for
Rhinos sequence
legend('Base Quality','Enhanced Quality',0)
xlabel('Frame #'), ylabel('SNR (dB)'), hold off
figure,plot(F(2:end),Base_psnr,'k*','LineWidth',1), hold on
```

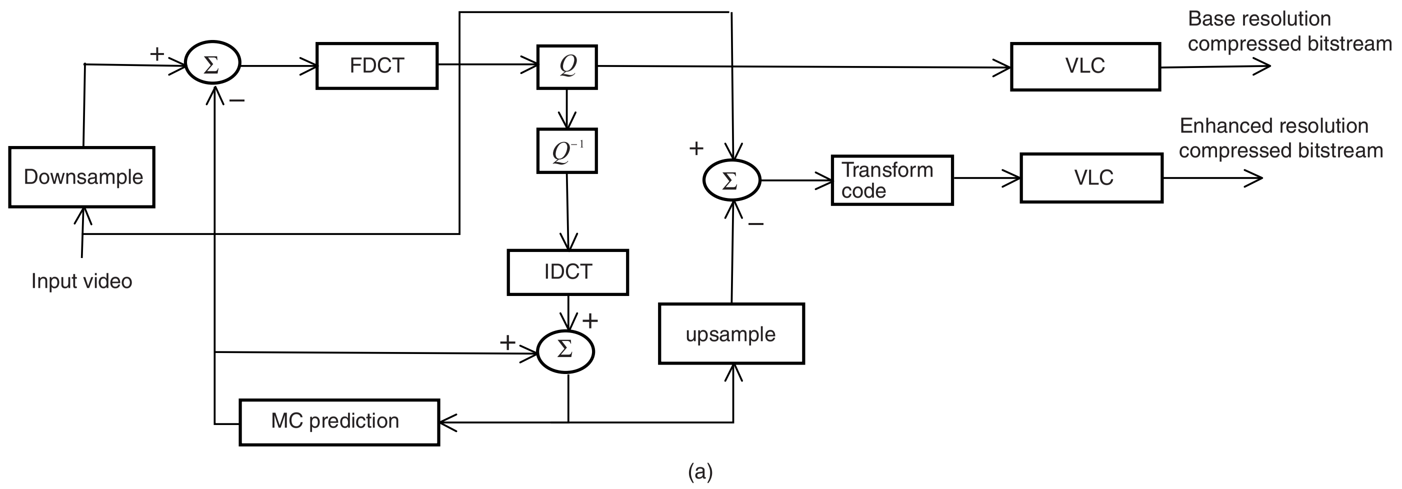
```
plot(F(2:end),Enhanced_psnr,'kd','LineWidth',2), title('PSNR (dB)')
axis([F(2) F(end) min(Base_psnr)-2 max(Enhanced_psnr)+2]) % for
Rhinos sequence
legend('Base Quality','Enhanced Quality',0)
xlabel('Frame #'), ylabel('PSNR (dB)'), hold off
```

Spatial Scalability The idea behind spatial scalability is similar to that of SNR counterpart, namely, that the base layer carries a compressed lower spatial resolution video and the enhancement layers carry incremental data [29]. When only the base layer bit stream is decoded, one obtains a base resolution video. When the base layer and enhancement layers are together decoded, the highest resolution video is obtained. However, the quality of both the base and enhanced resolution videos may be about the same because the intent is to distribute the same video material in a single bit stream which carries information about different resolutions of the videos.

The spatial scalability is illustrated in block diagram form in Figure 10.14. Figure 10.14a is the spatially scalable encoder. The input video is in full resolution. It is lowpass filtered and down sampled to the base resolution and then encoded using MPEG2 scheme. The difference between the full resolution input video and the base layer decoded (locally) video is coded using the DCT transform. The two bit streams are transmitted or stored. On the decoder side (Figure 10.14b), only the base layer is decoded to obtain the base resolution video. To obtain the full resolution video, both base and enhanced layers are decoded, the decoded base resolution pictures are upsampled and filtered, and the two are added. The quantizer scale may be the same in both layers and as a result the decompressed videos may have essentially the same quality but at different resolutions. It must be pointed out that spatial scalability can also be accomplished in the wavelet domain.

Example 10.2 We consider the same video as in Example 10.1, which has the innate resolution of 352×240 pixels in the red, green, and blue components. For the sake of illustration, let us use a base resolution of size 184×136 pixels, where both width and height are divisible by 8. We will only use the luma component in the encoding and decoding. The quantization matrix for the prediction error DCT has the same constant value of 16 for all the coefficients, as explained earlier. The quantizer scale to be used is 2. The same quantization matrix and quantizer scale values are used for the enhancement layer. The interpolation scheme used for both up and down sampling is "bicubic." Other possible interpolation methods are "nearest" and "bilinear."

The base and full resolution decoded pictures are shown in Figures 10.15a,b, respectively. These correspond to the second frame in the video. As can be seen from the Figures, both have similar visual quality except for the size. We next compute the SNR and PSNR for the 10 frames (frames 2–11) and show them as plots in Figures 10.16a,b, respectively. As expected, the SNR and PSNR values are nearly the



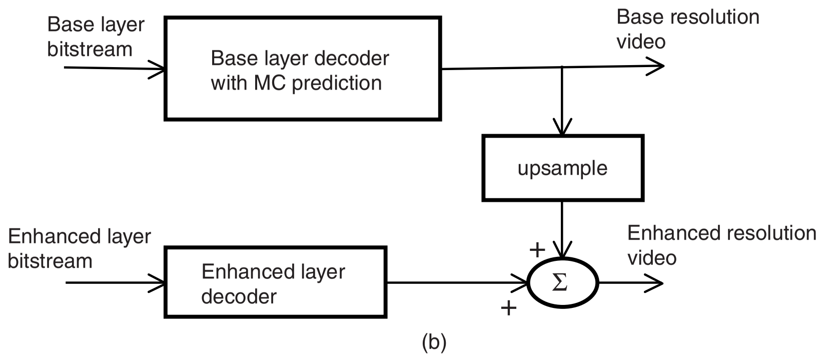
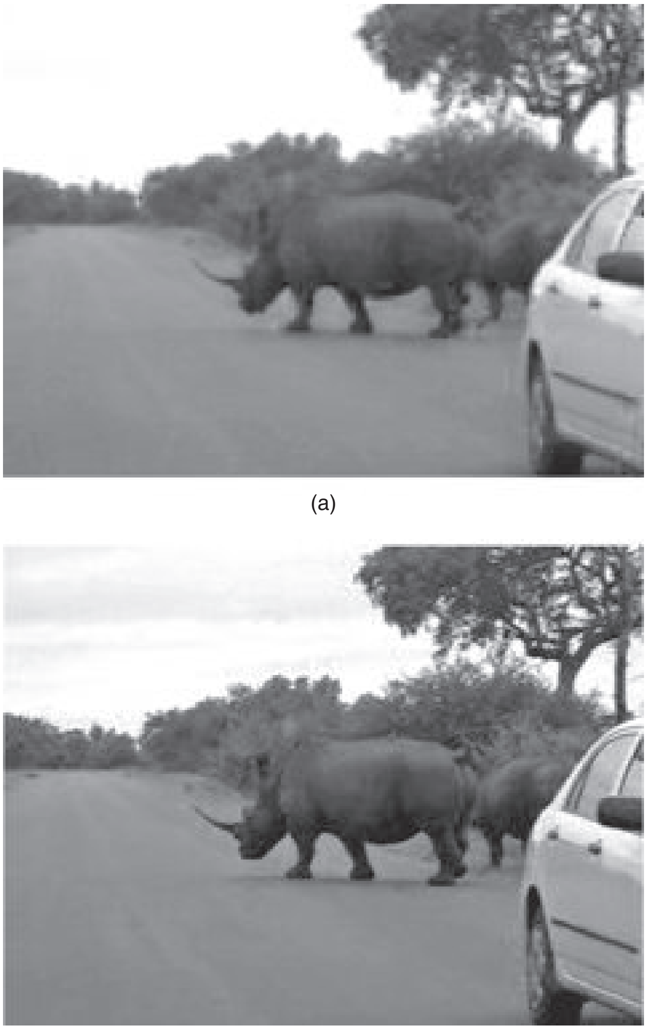


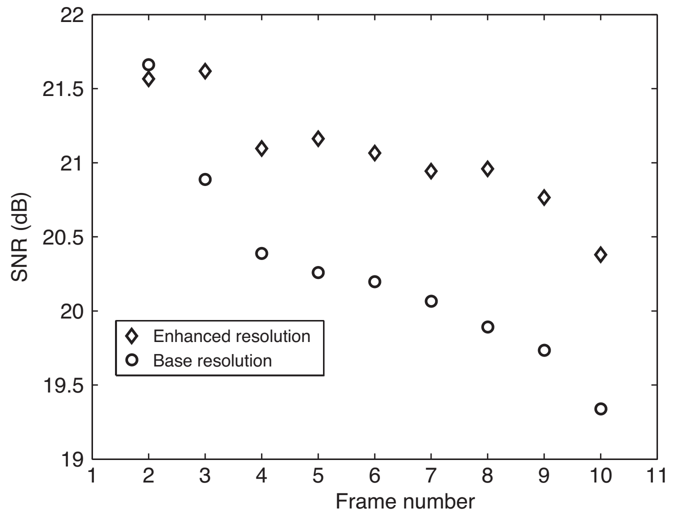
Figure 10.14 A block diagram illustrating a spatially scalable MPEG video coding system: (a) encoder and (b) decoder.



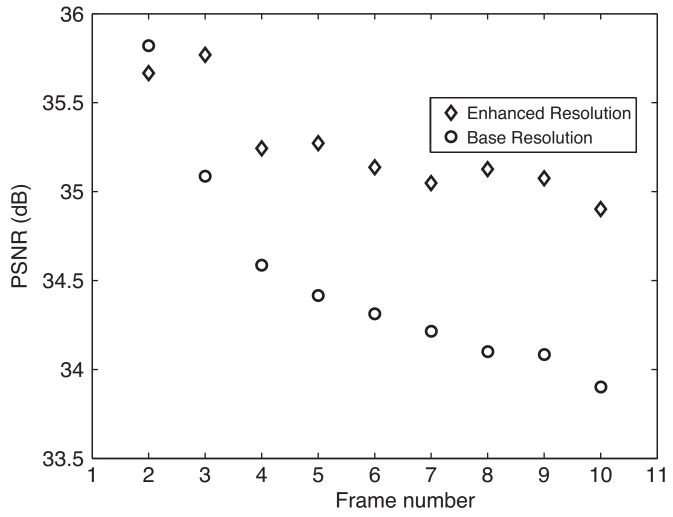
(a)

(b)

Figure 10.15 Decoded pictures using spatial scalable MPEG codec: (a) base resolution picture, (b) higher resolution picture. Input video is the rhinoceros AVI clip. The pictures (a) and (b) correspond to second and third frames in the video sequence. The picture in (a) is downsampled to 184 $\times$ 136 pixels and the picture in (b) is in full resolution of 352 $\times$ 240 pixels.



(a)



(b)

Figure 10.16 SNR and PSNR values for the rhinoceros video sequence using spatial scalable MPEG coding in Example 10.2: (a) SNR in dB and (b) PSNR in dB. The frames used are 2 through 10, inclusive.

same for both the lower and full resolution pictures. The MATLAB code for spatially scalable coding is listed below.

```
% Example10_2.m
% Spatially scalable video coding using motion compensated prediction
% Block motion is estimated to an integer pel accuracy
% using full search and SAD matching metric.
% Block size is 8 x 8 pixels.
% Search window size is 16 x 16 pixels.
% The differential block is 2D DCT transformed, quantized
% using uniform quantizer with constant quantization step of 16.
% The reconstructed block becomes the reference block for
% the next input frame.
% Base layer carries the above quantized data.
% The enhanced layer is created by upsampling the base layer
% reference image and subtracting it from the current full
% resolution picture, taking the block DCT of the difference image,
% quantizing and VLC coding.
% Decoding the base layer yields a lower spatial resolution video and
% adding the enhanced layer to the upsampled base layer results in a
% higher resolution video.
% Only intensity (Luma) image sequence is accepted.
% Note: Both Table Tennis and Trevor sequences have ".ras"
% extension while the Claire sequence has ".yuv" extension.


clear
N = 8;% block size is N x N pixels
N2 = 2*N;
W = 16; % search window size is W x W pixels
quantizer_scale = 2; % used for both the base and enhanced layers
Interp_type = 'bicubic'; % type of interpolation used for upsampling
% Four different sequences are tested.
% "rhinos.avi" exists in MATLAB and the others do not.
%inFile = 'tt040.ras';% Table Tennis sequence
%inFile = 'twy040.ras'; % Trevor sequence
%inFile = 'clairey040.yuv'; % Claire sequence
inFile = 'rhinos.avi'; % AVI file from MATLAB
strIndx = regexp(inFile,'\.');
if strcmpi('avi',inFile(strIndx+1:strIndx+3))
    M = aviread(inFile); % There are 114 320x240x3 frames
    F = int16(1:10);% frames to encode
else
    F = int16(40:49);% frames to encode
    M.cdata = cell(1,length(F));
    for k = 1:length(F)
        strIndx1 = regexp(inFile,'0');
        inFile1 = strcat(inFile(1:strIndx1(1)),num2str(F(k)),...
            inFile(strIndx:end));
        M(k).cdata = imread(inFile1);
    end
end
% Make image size divisible by 8
[X,Y,Z] = size(M(1).cdata);
if mod(X,8)~=0
    Height = floor(X/8)*8;
else
    Height = X;
end
if mod(Y,8)~=0
    Width = floor(Y/8)*8;
else
    Width = Y;
end
Depth = Z;
clear X Y Z
base_Height = 136; % height of lower resolution image
base_Width = 184;  % width of lower resolution image
%
if Depth == 3
    A = rgb2ycbcr(M(1).cdata);% Convert RGB to YCbCr & retain only Y
    y_ref_enhance = double(A(:,:,1));
    y_ref_base = imresize(A(:,:,1),[base_Height base_Width],Interp_type);
else
    A = M(1).cdata;
    y_ref_enhance = double(A);
    y_ref_base = imresize(A,[base_Height base_Width],Interp_type);
end
% pad the reference frame left & right and top & bottom
y_ref_base = double(padarray(y_ref_base,[W/2 W/2],'replicate'));
% arrays to store SNR and PSNR values
Base_snr = zeros(1,length(F)-1); Enhanced_snr = zeros(1,length(F)-1);
Base_psnr = zeros(1,length(F)-1); Enhanced_psnr = zeros(1,length(F)-1);
% Encode the monochrome video using MPC
for f = 2:length(F)
    if Depth == 3
        B = rgb2ycbcr(M(f).cdata);
        y_current_enhance = double(B(:,:,1));
        y_current_base = imresize(B(:,:,1),...
            [base_Height base_Width],Interp_type);
    else
        y_current_enhance = double(M(f).cdata);
        y_current_base = imresize(M(f).cdata,...
            [base_Height base_Width],Interp_type);
    end
    y_current_base = double(padarray(...
        y_current_base,[W/2 W/2],'replicate'));
```

```
for r = N:N:base_Height
    for c = N:N:base_Width
        D = 1.0e+10;% initial city block distance
        for u = -N:N
            for v = -N:N
                d = y_current_base(r+1:r+N,c+1:c+N)-...
                    y_ref_base(r+u+1:r+u+N,c+v+1:c+v+N);
                d = sum(abs(d(:)));% city block distance
                    between pixels
                if d < D
                    D = d;
                    x1 = v; y1 = u; % motion vector
                end
            end
        end
        % MC compensated difference coding
        temp = y_current_base(r+1:r+N,c+1:c+N)...
            -y_ref_base(r+1+y1:r+y1+N,c+1+x1:c+x1+N);
        TemP = dct2(temp); % DCT of difference
        s = sign(TemP); % extract the coefficient sign
        TemP = s .* round(abs(TemP)/(16*quantizer_scale))...
            *(16*quantizer_scale); % quantize/dequantize DCT
        temp = idct2(TemP); % IDCT
        Base(r-N+1:r,c-N+1:c) =...
            y_ref_base(r+1+y1:r+y1+N,c+1+x1:c+x1+N) +...
            temp; % reconstructed block - base quality
    end
end
% Generate enhancement
Delta = y_current_enhance - imresize(Base,[Height Width],
 Interp_type);
for r = 1:N:Height
    for c = 1:N:Width
        temp = Delta(r:r+N-1,c:c+N-1);
        temp_DCT = dct2(temp);
        s = sign(temp_DCT);
        temp_DCT = s .* round(abs(temp_DCT)/(16*quantizer_scale))...
            * (16*quantizer_scale);
        E(r:r+N-1,c:c+N-1) = idct2(temp_DCT);
    end
end
Enhanced = E + imresize(Base,[Height Width],Interp_type);
% replace previous frame by the currently reconstructed frame
y_ref_base = Base;
% pad the reference frame left & right and top & bottom
y_ref_base = double(padarray(y_ref_base,[W/2 W/2],'replicate'));
% Calculate the respective SNRs and PSNRs
Base_snr(f-1) = 20*log10(std2(...
    y_current_base(N+1:base_Height+N,N+1:base_Width+N))/...
```

```
        std2(y_current_base(N+1:base_Height+N,N+1:base_Width+N)-Base));
    Enhanced_snr(f-1) = 20*log10(std2(y_current_enhance)/...
        std2(y_current_enhance-Enhanced));
    Base_psnr(f-1) = 20*log10(255/std2(...
        y_current_base(N+1:base_Height+N,N+1:base_Width+N)-Base));
    Enhanced_psnr(f-1) = 20*log10(255/std2(y_current_enhance -
      Enhanced));
end
%

figure,plot(F(2:end),Enhanced_snr,'kd',F(2:end),Base_snr,'ko',
 'LineWidth',2)
xlim([F(2)-1 F(end)+1])
title('SNR (dB)')
legend('Enhanced Resolution','Base Resolution',0)
xlabel('Frame #'), ylabel('SNR (dB)')
%
figure,plot(F(2:end),Enhanced_psnr,'kd',F(2:end),Base_psnr,'ko',
 'LineWidth',2)
xlim([F(2)-1 F(end)+1])
title('PSNR (dB)')
legend('Enhanced Resolution', 'Base Resolution', 0)
xlabel('Frame #'), ylabel('PSNR (dB)')
```

Temporal Scalability As the name implies, temporal scalability provides different frame rate videos. The base layer yields the basic lower frame rate video. Higher rate video is obtained by decoding additional pictures in the data stream using temporal prediction with reference to the base layer pictures.

10.3 MPEG-4

As pointed out at the beginning of this chapter, MPEG-4 is targeted for low bit rate video coding. This implicates the necessity of achieving much higher compression efficiency. There are some basic differences between MPEG-1 and MPEG-2 and MPEG-4 schemes. In MPEG-1 and MPEG-2, motion estimation and compensation are performed on rectangular block boundaries. Even if motion is estimated to sub-pixel accuracies, still some residuals exist because objects, in general, are nonrectangular in shape. This results in compression inefficiency. On the other hand, MPEG-4 estimates the motion of actual object boundaries—not rectangular blocks, which results in much reduced residuals and higher compression efficiency thereof. MPEG-4 encodes objects and backgrounds independently. If the background is stationary over a period of time, then there is no need to transmit the background for each frame and this increases the compression efficiency. Another feature of MPEG-4 not available in MPEG-1 and MPEG-2 standards is that perspective motion of objects can be

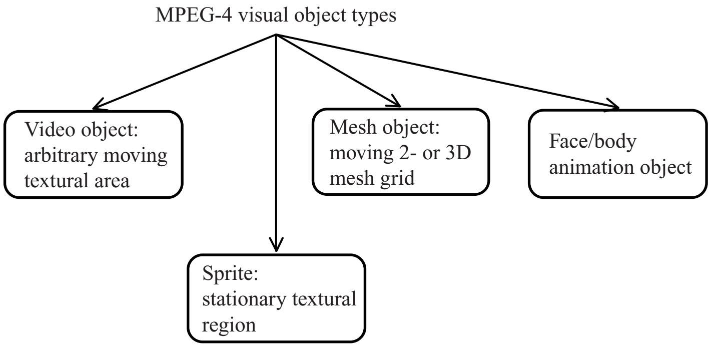

MPEG-4 visual object types
Video object: arbitrary moving textural area
Mesh object: moving 2- or 3D mesh grid
Face/body animation object
Sprite: stationary textural region

Figure 10.17 Types of visual objects supported in MPEG-4/H.263.

estimated by warping the reference frame and then determining the motion. This is done using what is known as *mesh coding*. In MPEG-4, computer animation and user interaction are permitted as well.

10.3.1 MPEG-4 Part 2

MPEG-4 Part 2 is also known as H.263. Here, the visual objects are classified into four groups, namely, video objects (VOs), sprite objects, mesh objects, and animation objects, as shown in Figure 10.17. A *video object* is a rectangular region of pixels that defines an object in a picture. The video object can be moving. A *sprite* is a stationary region of *texture*. What we mean by texture is the actual picture with variations in intensity and color. A mesh object is a 2D/3D shape as a set of grid points that can be moving. To create motion, texture can be mapped on to the mesh. Figure 10.18 shows a warped 2D mesh with the texture mapped on to it. Coding only the mesh will

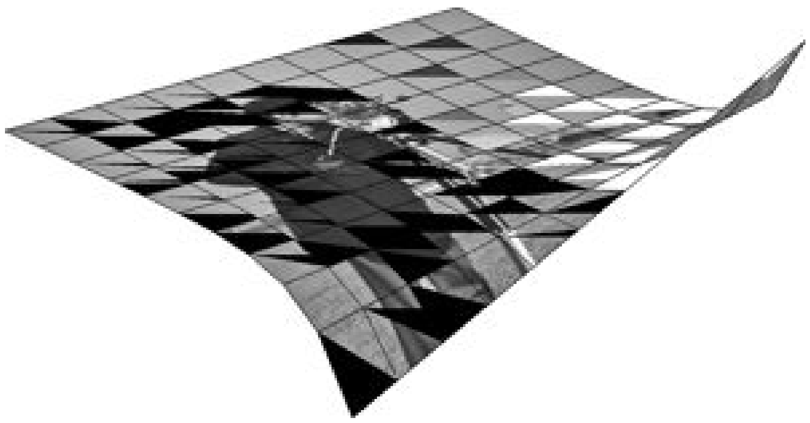


Figure 10.18 Example of a warped 2D mesh with texture mapped on to it.

result in very high compression efficiency. The fourth type of visual object is the face and body animation object. Using 3D mesh and warping techniques, face or body animation can be achieved from a single still image. Again, very high compression efficiency can be achieved by coding a single face or body and the motion vectors.

Example 10.3 To understand what the object and background are, let us take frame 40 of the Claire picture and dissect it to create shape boundary, textural object, and background. Figure 10.19 shows the various items of interest. The top left is the original picture, which consists of the upper body against a uniform background. The boundary of the shape of the object is shown in the top right image. The textural object is shown in the bottom left picture where the background is blacked out. The background is shown in the bottom right image where the object is blacked out. Note that the background in the bottom right image appears brighter than that in the original picture (top left). This is due to the contrast difference between the two pictures: the contrast is higher in the bottom right image. Using 3D warping technique, it is possible to create face and body animation from this single image. Since the

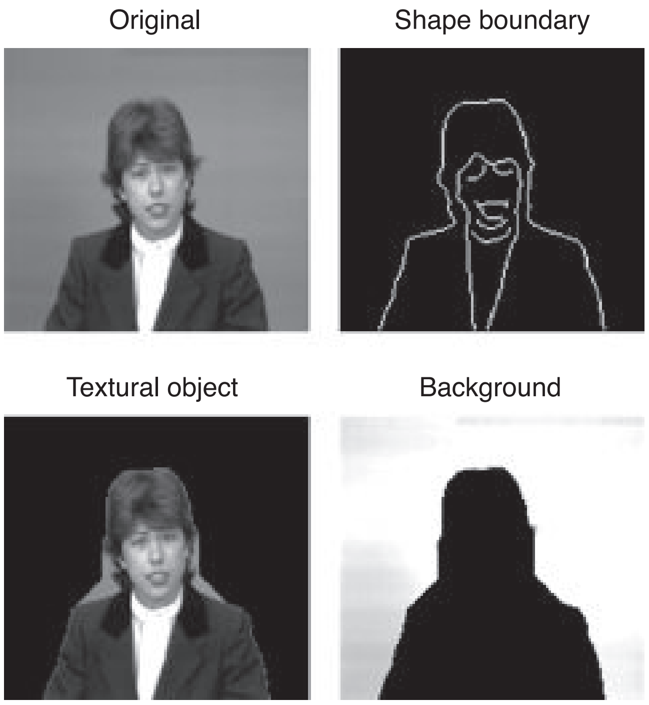


Figure 10.19 Object shape, texture, and background. Top left: frame 40 of the original Claire picture consisting of head and shoulder and uniform background, top right: object boundary, bottom left: textural object with background blacked out, and bottom right: background with object blacked out. Note that the background in the original image has a different visual appearance from that in the bottom right background, which is due to the difference in contrast.

background or sprite is fixed, motion vectors corresponding to the warping can be incorporated to create face and/or body motion.

In this example, we have used the MATLAB function "edge" to draw the boundary of the object. Although there are many edge detection algorithms, "canny" algorithm seems to work better. The output of the edge detection function is a mask of the same size as the image where the edges have values "1" and nonedges have values "0." However, to create the textural object, we need to fill the region within the object boundary. This is done by the morphological functions "imclose" followed by "imopen." Both morphological functions use a structuring mask, which may be "disk," or "square," or any other user specified 2D masking function. After having filled the object region, it is a simple task to create the object texture and background.

```
% Example10_3.m
% Program to create video object shape, texture
% and background from a single picture using
% Canny's edge detection followed by the
% morphological operations of closing and opening.

clear
A = imread('clairey040.yuv');
% The Claire image has two stripes at the left and right ends
% and so we want to remove them and keep the rest.
A1 = A(:,11:165);
% Canny's method has two parameters: a) Threshold to
  block out spurious
% edges, and b) standard deviation which corresponds to
  the width of the
% Gaussian filter used.
E = edge(A1,'canny',.25,1.25);
E1 = E; % Retain the edge image as a mask
% Use a disk of radius 28 as the structural element
  for the "imclose"
% function and the edge mask as the structural element for the
% "imopen" function to fill the region within the object
  boundary.
E = imopen(imclose(E,strel('disk',28)),E1);
A2 = uint8(zeros(size(A1)));
% To extract the object from the background, assign "0" to the
% region where the edge image is "0" and assign the
  actual pixel
% values to the region where the edge image is "1".
% The background is obtained by subtracting the object texture
% from the original picture.
A2(logical(E == 1)) = A1(logical(E == 1));
% Display the various images as a collage.
figure,subplot(2,2,1),imshow(A1),title('Original')
```

```
subplot(2,2,2),imshow(E1),title('Shape Boundary')
subplot(2,2,3),imshow(A2),title('Textural Object')
subplot(2,2,4),imshow(double(A1)-double(A2),[]),
title('Background')
```

Video Object Plane A video object plane (VOP) is a plane in which the video object shape and texture are displayed. Similar to MPEG-1 and MPEG-2, a VOP can be coded as intra or inter. If it is intercoding, then it can be P-VOP or B-VOP. If it is intracoding, then it is I-VOP.

Shape and texture data are coded separately and transmitted. The decoder then creates animation by compositing the shape and texture data. The shape data is decoded to produce a keying signal, which tells the decoder where to place the object texture in a VOP.

Figure 10.20 shows the format of a video object within a VOP. It consists of a shape and texture within a bounding rectangle. The rectangular region consists of rectangular macroblocks. Blocks outside the shape are transparent and have no texture and result in very high compression. Blocks within the shape region are textures and are coded using DCT.

Coding Textural Regions Textural regions in I-VOP are coded using 8 × 8 2D DCT. MPEG-4 achieves higher coding efficiency by coding not only the DC difference but also a row or column of DCT coefficient difference [28,30]. Typically, there is high correlation in the average luminance and chrominance components between adjacent blocks. Since the DC coefficient is a measure of the block average, it is more efficient to code the DC difference rather than coding the quantized DC coefficient itself. This is also the case with MPEG-1 and MPEG-2. However, in MPEG-4, the block that has a smaller gradient in DC value is used as the prediction of the DC value of the current block to be coded. Smaller gradient in the DC value implies that

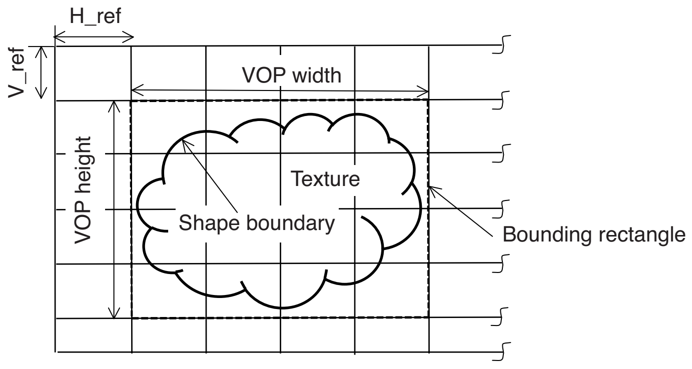


Figure 10.20 A bounding rectangle in a VOP. A video object is bounded by a rectangle consisting of different types of macroblocks such as opaque, transparent, and so on.

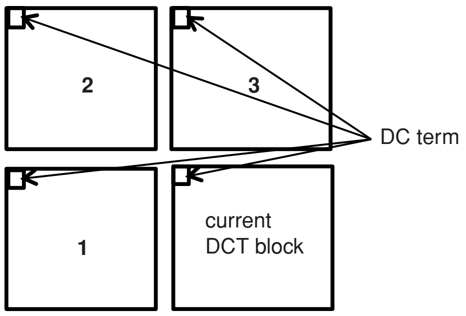


Figure 10.21 Arrangement of four DCT blocks for I-VOP DCT prediction. The three blocks adjacent to the current block being coded are numbered 1, 2, and 3 in a counterclockwise direction. The DC coefficient of each neighboring block is shown by a small square at the top left corner. The 3 DC coefficients are used in the calculation of the vertical and horizontal gradients.

the corresponding block correlation is higher. Figure 10.21 shows four DCT blocks. The three blocks adjacent to the current block being coded are numbered 1, 2, and 3. There are two gradients, vertical and horizontal. The vertical gradient G_V is defined as

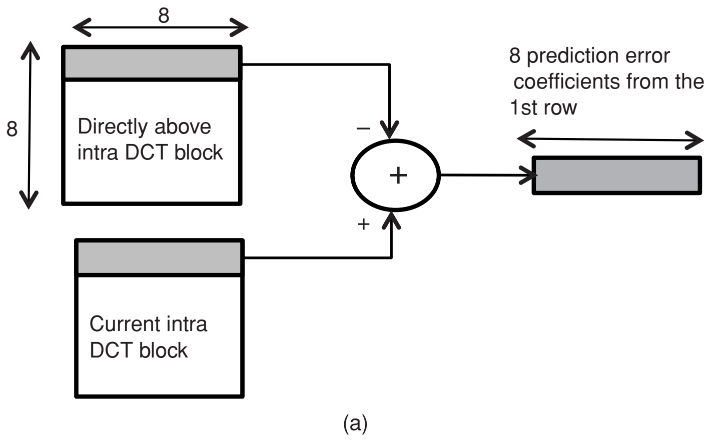
$$G_V = |\mathrm{DC}_3 - \mathrm{DC}_2| \tag{10.3}$$

and the horizontal gradient G_H is defined as

$$G_H = |\mathrm{DC}_1 - \mathrm{DC}_2| \tag{10.4}$$

In equations (10.3) and (10.4), DC_i, $1 \leq i \leq 3$ are the DC coefficients of the blocks 1, 2, and 3, respectively. In MPEG-4, an entire row or column of DCT coefficients of the current block is predicted from the row or column of the adjacent block that has the lower gradient. That is to say that if $G_V \prec G_H$, then the entire first row of the DCT coefficients of the current block is predicted from the first row of the DCT of block 3 that is directly above the current block. If not, then the entire first column of the DCT coefficients of the current block is predicted from the first column of the DCT of block 1 that is directly to the left of the current block. The vertical and horizontal predictions are shown diagrammatically in Figures 10.22a,b, respectively. The rest of the quantized AC coefficients are coded as they are, after zigzag scanning. MPEG-4 allows two additional coefficient scanning patterns, which are shown in Figures 10.23a,b. Vertical prediction generates more nonzero coefficients in the horizontal direction. Therefore, a horizontal scan pattern (Figure 10.23a) will produce larger zero runs. Similarly, when using horizontal prediction, vertical scan pattern (Figure 10.23b) will yield larger zero runs.

Example 10.4 The intent of this example is to show that in intracoding, prediction of the DCT coefficients reduces the range of values not only in the DC but also in the


8
8
Directly above intra DCT block
−
+
+
8 prediction error coefficients from the 1st row
Current intra DCT block
(a)

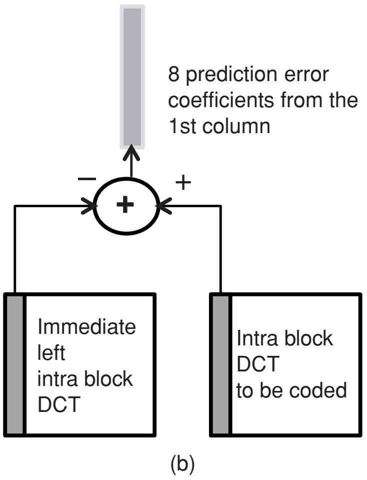

8 prediction error coefficients from the 1st column
−
+
+
Immediate left intra block DCT
Intra block DCT to be coded
(b)

Figure 10.22 Diagrammatic representation of DCT prediction used in MPEG-4/H.263. (a) Vertical prediction that involves the block directly above the current block. The first row of DCT coefficients of the block directly above the current block is subtracted from that of the current block to obtain the prediction residuals. (b) Horizontal prediction where the first column of DCT coefficients of the block immediately to the left of the current block is subtracted from that of the current block to obtain the prediction residuals.

AC coefficients in the first row or column of an 8×8 block. We read the “Aerial” picture, which has large areas of fine details (see Figure 10.24). Vertical or horizontal prediction of the row or column of 8×8 DCT blocks is carried out according to equations (10.3) and (10.4). Figure 10.25 shows the histograms of the DC coefficient for the vertical and horizontal predictions. The top and bottom left figures show the histograms of the unquantized DC coefficients in the vertical and horizontal directions,

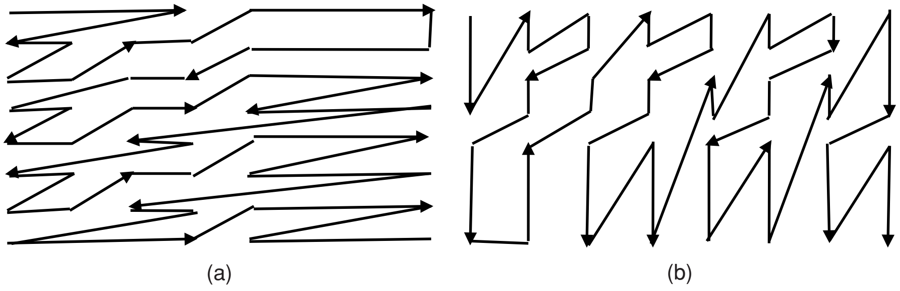


Figure 10.23 Additional zigzag scanning supported in MPEG-4/H.263: (a) horizontal scanning pattern which is used when vertical prediction of the DCT coefficients is used and (b) vertical scanning pattern which is used when horizontal prediction of the DCT coefficients is used.

Figure 10.24 "Aerial" picture that is an intensity picture used in Example 10.4 for the intra-DCT prediction.

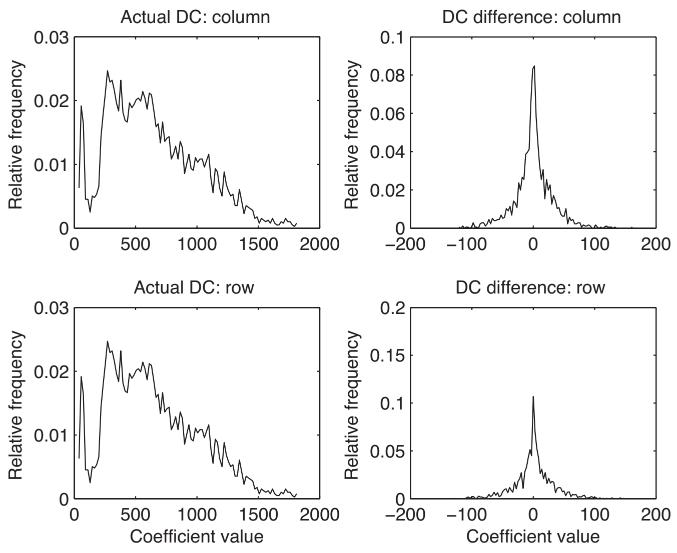


Figure 10.25 Histograms of the prediction residuals of the DC coefficients of the "Aerial" picture. Top left: histogram of the unquantized DC coefficients for the vertical prediction, top right: histogram of the DC coefficients for the vertical prediction after quantization, bottom left: histogram of the unquantized DC coefficients for the horizontal prediction, and bottom right: histogram of the DC coefficients for the horizontal prediction after quantization. A quantization value of 8 is used to quantize all the DCT coefficients.

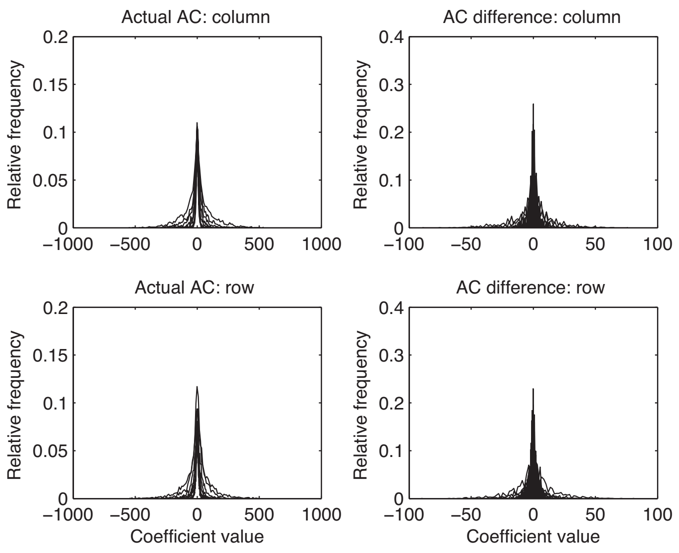


Figure 10.26 Histograms of the prediction residuals of the AC coefficients of the "Aerial" picture. Top left: histograms of the unquantized AC coefficients for vertical prediction (first row), top right: histograms of the corresponding quantized AC coefficients, bottom left: histograms of the unquantized AC coefficients for horizontal prediction (first column), and bottom right: histograms of the corresponding quantized AC coefficients.

Table 10.12 Standard deviation of DC and AC coefficients

DCT Coefficients	Actual Vertical	Predicted Vertical	Actual Horizontal	Predicted Horizontal
0	363	31	363	32
1	116	17	106	16
2	73	11	66	10
3	49	8	44	7
4	33	5	32	5
5	23	4	23	4
6	16	2	16	3
7	11	2	12	2

respectively. The top and bottom right figures portray the histograms of the quantized DC coefficient prediction errors in the vertical and horizontal directions, respectively. As expected, the predicted DC values are centered at zero with a much reduced dynamic range. Hence, compression is achieved. In this example, both DC and AC coefficients are quantized by the same value of 8 just to emphasize the fact that AC prediction also contributes to coding efficiency.

The histograms of the actual AC coefficient values and the prediction errors of the quantized AC coefficients along the first row or column as determined by equations (10.3) and (10.4) are shown in Figure 10.26. The interesting thing is that the region near zero value is completely filled in the histogram of the predicted AC coefficients, whereas it is hollow in the histogram of the actual AC coefficients. This indicates that AC prediction does result in more zero-valued coefficients and hence higher coding efficiency. The standard deviations of the DC and AC coefficients for the vertical and horizontal predictions are listed in Table 10.12. As can be seen from the table, there is quite a bit of reduction in the standard deviations in both the DC and the AC coefficients. Note that coefficient 0 is the DC and the rest are AC coefficients.

```
% Example10_4.m
% Program to calculate vertical and horizontal
% predictions of DC and AC coefficients as per MPEG-4 rule.
% With reference to Fig. 10--21, vertical prediction is used
% if the horizontal gradient is smaller, otherwise, horizontal
% prediction is used. The program calculates the statistics
% only for the luma component. The block size is 8 × 8 pixels.
% A constant quantization value of 8 is used as an example.

clear
%A1 = imread('cameraman.tif');
A1 = imread('aerial.ras');
% Make image size divisible by 8
[X,Y,Z] = size(A1);
```

```
if mod(X,8) ~ = 0
   Height = floor(X/8)*8;
else
   Height = X;
end
if mod(Y,8) ~ = 0
   Width = floor(Y/8)*8;
else
   Width = Y;
end
Depth = Z;
clear X Y Z
% if the input image is RGB, then convert it to YCbCr
  & retain only the
% Y component.
if Depth = = 3
   A1 = rgb2ycbcr(A1);
   A = A1(:,:,1);
else
   A = A1;
end
% Compute the 8 × 8 DCT of the input image
BLKdct = blkproc(double(A),[8 8], @dct2);
% Quantize the DCT coefficients by a constant value of 8.
BLKdctQ = round(BLKdct/8);
% Calculate vertical & horizontal predictions of the
% first column or row of the quantized coefficient block.
% If the horizontal gradient is smaller, then perform
% vertical prediction. Otherwise perform horizontal prediction.
Vcount = 1; Hcount = 1;
for m = 9:8:Height
   for n = 9:8:Width
      gradV = abs(BLKdctQ(m-8,n-8) -- BLKdctQ(m-8,n));
      gradH = abs(BLKdctQ(m-8,n-8) -- BLKdctQ(m,n-8));
      if gradV > gradH
         Vpredict(Vcount,:) = BLKdctQ(m:m+7,n) -- BLKdctQ(m:m+7,n-8);
         Vcount = Vcount + 1;
      else
         Hpredict(:,Hcount) = BLKdctQ(m,n:n+7) -- BLKdctQ(m-8,n:n+7);
         Hcount = Hcount + 1;
      end
   end
end
% Compute the histograms of the actual and predicted differentials
Cnt = 1;
for m = 9:8:Height
   for n = 9:8:Width
      Vdct(Cnt,:) = BLKdct(m:m+7,n);
      Hdct(:,Cnt) = BLKdct(m,n:n+7);
      Cnt = Cnt + 1;
   end
end
%
for m = 1:8
   [HorigV(m,:),BorigV(m,:)] = hist(Vdct(:,m),100);
   HorigV(m,:) = HorigV(m,:)/sum(HorigV(m,:));
   [HorigH(m,:),BorigH(m,:)] = hist(Hdct(m,:),100);
   HorigH(m,:) = HorigH(m,:)/sum(HorigH(m,:));
   [Hv(m,:),Bv(m,:)] = hist(Vpredict(:,m),100);
   Hv(m,:) = Hv(m,:)/sum(Hv(m,:));
   [Hh(m,:),Bh(m,:)] = hist(Hpredict(m,:),100);
   Hh(m,:) = Hh(m,:)/sum(Hh(m,:));
end
% Plot the histograms
figure, subplot(2,2,1),plot(BorigV(1,:),HorigV(1,:),'k')
title('Actual DC: Column'),ylabel('Relative Frequency')
subplot(2,2,2),plot(Bv(1,:),Hv(1,:),'k')
title('DC difference: Column'), ylabel('Relative Frequency')
subplot(2,2,3),plot(BorigH(1,:),HorigH(1,:),'k')
title('Actual DC: Row')
xlabel('Coefficient value'), ylabel('Relative Frequency')
subplot(2,2,4),plot(Bh(1,:),Hh(1,:),'k')
title('DC difference: Row')
xlabel('Coefficient value'),ylabel('Relative Frequency')
%
figure, subplot(2,2,1),plot(BorigV(2,:),HorigV(2,:),'k')
title('Actual AC: Column'), ylabel('Relative Frequency')
hold on
for k = 3:8
   plot(BorigV(k,:),HorigV(k,:),'k')
end
hold off
%
subplot(2,2,2),plot(Bv(2,:),Hv(2,:),'k')
title('AC difference: Column'), ylabel('Relative Frequency')
hold on
for k = 3:8
   plot(Bv(k,:),Hv(k,:),'k')
end
hold off
%
subplot(2,2,3),plot(BorigH(2,:),HorigH(2,:),'k')
title('Actual AC: Row')
xlabel('Coefficient value'), ylabel('Relative Frequency')
hold on
for k = 3:8
   plot(BorigH(k,:),HorigH(k,:),'k')
end
```

```
hold off
%
subplot(2,2,4),plot(Bh(2,:),Hh(2,:),'k')
title('AC difference: Row')
xlabel('Coefficient value'), ylabel('Relative Frequency')
hold on
for k = 3:8
   plot(Bh(k,:),Hh(k,:),'k')
end
hold off
%
% calculate the correlation of the DCT coefficient row and column
CorrV = sum(Vdct.* Vdct)/length(Vdct);
for k = 1:8
   CorrH(k) = sum(Hdct(k,:).* Hdct(k,:))/length(Hdct);
end
figure, subplot(2,1,1), stem(CorrV,'k')
title('Correlation of DCT Column')
ylabel('Correlation')
subplot(2,1,2), stem(CorrH,'k')
title('Correlation of DCT Row')
xlabel('Coefficient #')
ylabel('Correlation')
% Calculate the standard deviations
sprintf('std. dev. vertical prediction:%6.2f\n',std(Vpredict,0,1))
sprintf('std. dev. vertical actual:%6.2f\n',std(Vdct,0,1))
sprintf('std. dev. horizontal prediction:%6.2f\n',std(Hpredict,0,2))
sprintf('std. dev. horizontal actual:%6.2f\n',std(Hdct,0,2))
sprintf('Vertical Corre.:%6.5f\n',CorrV/max(CorrV))
sprintf('Horizontal Corre.:%6.5f\n',CorrH/max(CorrH))
```

Video Object Coding As pointed out earlier, MPEG-4 uses object-based coding scheme. Therefore, the audio/video material to be compressed is divided into audio and VOs. We are concerned only with the VOs here. The first task in video object coding is to segment each picture into VOs. In Example 10.3 was shown how to segment a scene into object and background (see Figure 10.19). The shape consists of a binary mask whose values are either 0 or 255. All pixels within the shape boundary are assigned 255 and pixels outside the shape boundary are assigned 0. Thus, corresponding to an input video sequence, there is a sequence of VOs. Each VO consists of shape and texture. These are coded by different procedures. After coding all VOs (both objects and backgrounds), the data is multiplexed and transmitted.

A high-level VO-based MPEG-4 video coder is shown in Figure 10.27. The video segmentation may be carried out using semiautomatic or fully automatic procedure. As we experienced in Example 10.4, segmentation involves edge detection, which in turn uses a threshold to determine whether a pixel belongs to an edge or

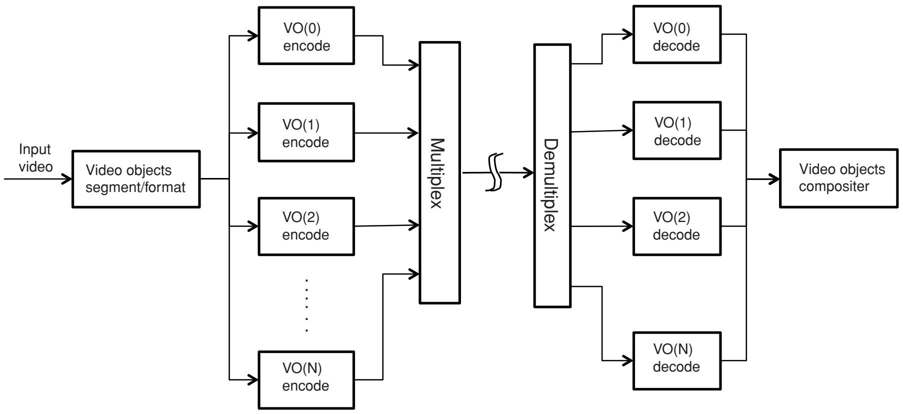


Figure 10.27 A high-level MPEG-4/H.263 video object encoder.

not. Because of variations in luminance from frame to frame and from video to video, it is hard to find a universal threshold for the edge detection scheme. Therefore, it may be necessary to employ semiautomatic procedure at least some of the times during the coding process. Each segmented VO is coded individually and the compressed data is multiplexed for transmission or storage. The decoder demultiplexes the compressed data, decodes the VOs, and delivers it to the compositor for rendering.

Figure 10.28 shows MPEG-4 VO coding in detail. Texture (a region of luma and chroma values) is coded using I-VOP, P-VOP, and B-VOP coding scheme similar to MPEG-1 and MPEG-2. However, as noted earlier, enhancements in compression such as DC and AC predictions are used to improve coding efficiency. Shapes, which are binary masks, are coded using context prediction [28, 30].

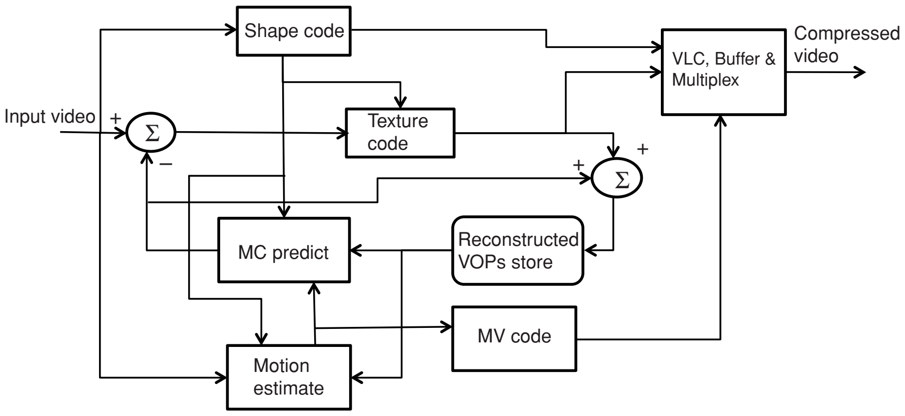


Figure 10.28 Block diagram of an MPEG-4/H.263 VO encoder.

10.4 H.264

MPEG-1 is targeted for bit rates of around 1.5 Mb/s, while MPEG-2 is meant for rates 10 Mb/s or higher. MPEG-2 is widely used for TV broadcasting. With the maturation of mobile phone and wireless network technologies, a need arose for lower bit rate audio-visual coding standard that is resilient to errors. This lower bit rate demands a more efficient compression scheme, hence, the MPEG-4 standard. Now there is a twist to this situation. HDTV is beginning to gain widespread usage, and as a result, it is creating the need for much higher compression efficiency. In addition to satellite broadcast, there are other transmission media such as cable modem, DSL, or UMTS that offer programs at much lower data rates than that of broadcast channels. Therefore, a higher compression-efficient standard will be able to offer more video channels or higher quality video over the existing low data rate media. This has motivated the emergence of the so-called AVC or H.264 video compression standard [31].

10.4.1 Highlights of AVC Features

Some of the solutions offered by H.264/AVC standard are (1) broadcast over satellite, cable modem, DSL, and so on, (2) interactive or serial storage on optical or magnetic recording devices such as Blue Ray DVD, (3) conversational services over LAN, Ethernet, wireless and mobile networks, and so on, (4) video-on-demand or multimedia streaming services over cable modem, DSL, and so on, and (5) multimedia messaging services over DSL, ISDN, mobile and wireless networks, and so on. In order to cater to the needs of a wide variety of service media, the H.264/AVC standard covers two systems, namely, an efficient video coding layer (VCL) for the compression of the video data and a network abstraction layer (NAL), which formats the VCL data in a manner suitable for a variety of service media to carry the compressed material.

As our concern here is with video compression, we will briefly highlight some of the features of H.264/AVC to get a better perspective of how compression efficiency is improved over MPEG-2:

- *Variable block-size motion compensation*: Motion compensation can be performed on blocks as small as 4×4 pixels instead of fixed-size macroblocks. Areas in a picture with high details have a high threshold of noise visibility and so such areas can be quantized heavily without incurring any visible distortions in the reconstructed picture. Higher quantization results in a higher compression.
- *Quarter-pel prediction*: Increasing the accuracy of motion estimation to a quarter pel results in reduced prediction residuals, which in turn results in a higher coding efficiency.
- *Motion vectors over picture boundaries*: Picture boundary extrapolations are used that allow motion vectors to point to areas outside the previously reconstructed reference picture.

- *Motion compensation using multiple reference pictures*: Higher coding efficiency is achieved by allowing motion compensation to be performed with respect to multiple reference pictures that have already been reconstructed. This also applies to B-picture coding.
- *Independent decoding and display order*: H.264/AVC allows the encoder to choose the ordering of pictures for decoding and displaying with a high degree of flexibility. This flexibility eliminates the extra delay associated with B-pictures.
- *Flexibility in B-picture coding*: Unlike the earlier standards, H.264/AVC removes the restriction in coding B-pictures where only previously coded I- or P-pictures may be used. This flexibility allows for referencing the picture that approximates more closely the current picture being coded.
- *Weighted prediction*: Motion compensated prediction residuals can be weighted and offset to improve coding efficiency.
- *Improved intraprediction*: Use of directional prediction improves coding efficiency.
- *In-loop deblocking filter*: As we noted earlier, blocking artifacts arise in DCT-based coding because of the use of rectangular blocks of pixels. Blocking artifacts can be reduced or removed by filtering the reconstructed image as post-processing. However, H.264 uses deblocking filtering in the prediction loop to improve prediction as well as to remove blocking artifacts.
- *4 × 4 transform*: H.264/AVC compression uses 4 × 4 DCT-like transform to compress the prediction residuals in intracoding. The coefficients of the 4 × 4 transform kernel are integers as shown in equation (10.5):

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \tag{10.5}$$

- *Hierarchical block transform*: Uses a hierarchical transform to extend the effective block size to 8 × 8 for low-frequency chroma information. It also uses a special intracoding type to encode 16 × 16 luma blocks.
- *Short word-length for transform*: Unlike the previous standards, which generally require higher arithmetic precision, H.264 requires only 16-bit arithmetic in the computation of the block transform.
- *Exact match in inverse transform*: When the inverse transform at the decoder does not match exactly the standard's specified rule, the decoded video for each decoder design will produce slightly different quality, which will cause a drift between the encoder and decoder. Note that the encoder has a decoder built into its loop and that the decoder loop must follow the same en-

coder loop. But H.264 achieves exact equality of decoded video in all the decoders.

- *Advanced arithmetic coding*: H.264 uses two advanced arithmetic coding methods known as CABAC (context-adaptive binary arithmetic coding), and CAVLC (context-adaptive variable-length coding) to improve performance from MPEG-2.

10.4.2 H.264 Macroblock Coding

As in the other MPEG standards, a macroblock consists of a 16 × 16 luma block and an 8 × 8 block of each of the Cb and Cr chroma components. A slice is made up of several macroblocks. H.264 uses *flexible macroblock ordering* (FMO) wherein slices need not be in raster scan order.

Figure 10.29 depicts the H.264 video encoding process. In intraspatial prediction using 4 × 4 blocks, nine modes of prediction are supported. A 4 × 4 block of pixels labeled "a" through "p" are predicted from a row of eight pixels labeled "A" through "H" above the current block and a column of four pixels labeled "I" through "L" to the left of the current block as well as a corner pixel labeled "M," as shown in Figure 10.30. The nine modes of 4 × 4 intraprediction are mode 0 (vertical prediction), mode 1 (horizontal prediction), mode 2 (DC prediction), mode 3 (diagonal down/left prediction), mode 4 (diagonal down/right prediction), mode 5 (vertical-right prediction), mode 6 (horizontal-down prediction), mode 7 (vertical-left prediction), and

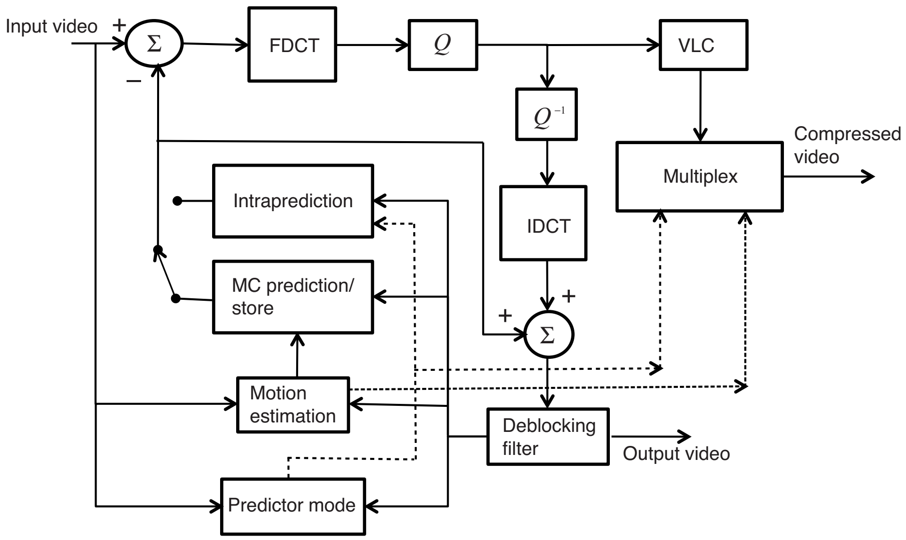


Figure 10.29 H.264/AVC video encoder.

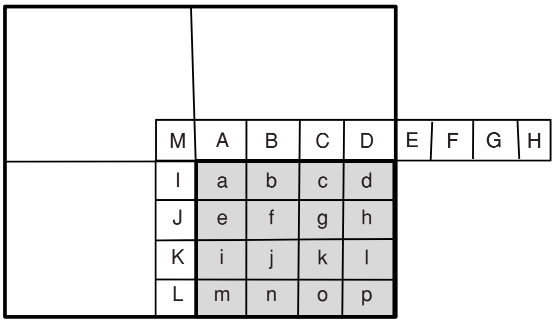


Figure 10.30 Relationship of the 16 pixels labeled "a" through "p" of the current 4 × 4 block to its neighboring row of pixel on the top and column of pixels to the left of the current block.

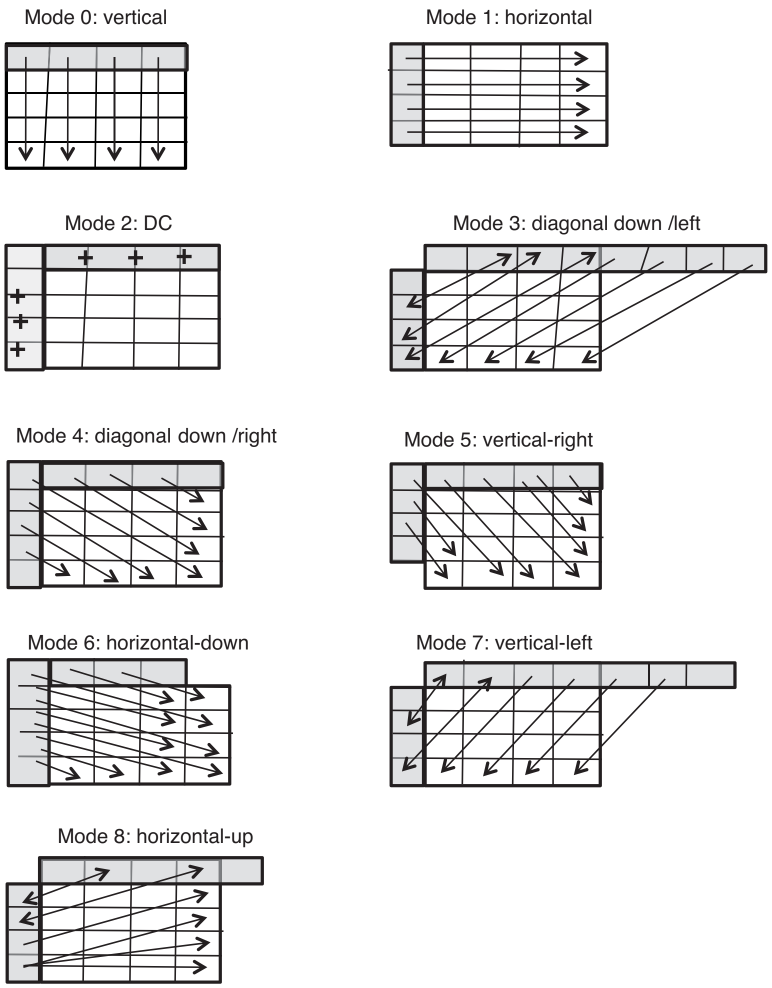


Figure 10.31 Modes of directional intraprediction of 4 × 4 blocks supported by H.264/AVC.

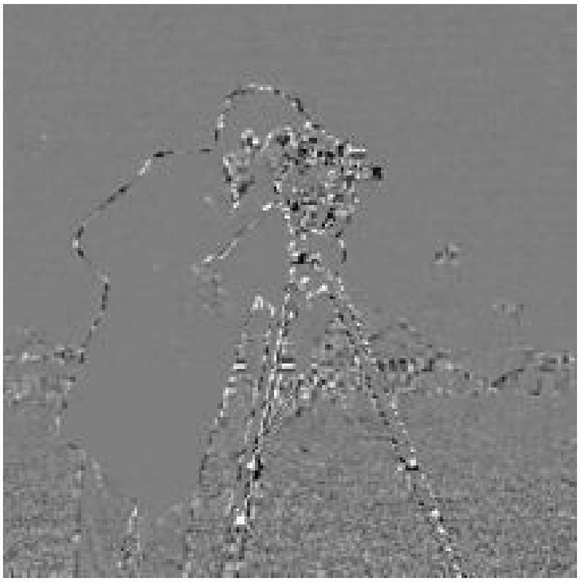


Figure 10.32 Intrapredicted error image of the Cameraman image of Example 10.5. This obtained by predicting 4 × 4 luma blocks using the nine modes of prediction with SAD as the metric.

mode 8 (horizontal-up prediction). These nine modes are shown in Figure 10.31. While 4 × 4 block prediction is used mostly to compress areas of high details, 16 × 16 blocks are selected for intracoding flat areas (areas where there are relatively little spatial variation in pixels). Similar to the 4 × 4 prediction, four modes of intraprediction are used for 16 × 16 blocks. These are mode 0 (vertical prediction), mode 1 (horizontal prediction), mode 2 (DC prediction), and mode 3 (plane prediction). In a 16 × 16 prediction, 16 neighbors on top and left sides are used.

In motion compensated predictive coding of P-pictures, blocks of sizes 4 × 4, 8 × 8, and 16 × 16 are utilized with possible combinations of 16 × 16, 16 × 8, 8 × 16, and 8 × 8. If 8 × 8 block prediction is used, H.264 supports 8 × 4, 4 × 8, and 4 × 4 blocks.

Variable block-size prediction and other refined compression methods used in H.264 provide higher compression efficiency for the same quality. Hence, the name AVC. Finally, we will show how intra 4 × 4 prediction is carried out by way of an example.

Example 10.5 The cameraman picture is used in this example to carry out the intraprediction using 4 × 4 blocks of luma pixels. The example illustrates the idea of directional intraprediction using the nine modes shown in Figure 10.31. Since we are not going to use slice groups, we will pad the picture with an extra row at the top, a column to the left, and four columns to the right. Starting from the top left corner, we will do the prediction of 4 × 4 blocks in a raster scan order and obtain

Table 10.13 Prediction gain for 4 × 4 directional intraprediction

Input Picture	Prediction Gain (dB)
Aerial	**7.27**
Airplane	18.84
Birds	16.26
Cameraman	11.80
Lighthouse	9.62
Masuda1	**22.68**
Peppers	18.14
Yacht	13.92

the prediction residuals. Finally, we will calculate the prediction gain (ratio of the variance of the input picture to that of the prediction residuals, expressed in dB) for the 4 × 4 intraprediction corresponding to the input picture.

Figure 10.32 shows the prediction residual image corresponding to the input cameraman picture. Table 10.13 lists the prediction gain for several pictures of different sizes. These pictures have been used in previous chapters and so we will not display them here. Because the effectiveness of prediction depends on the activities in the given image, we anticipate variations in the prediction gain. The aerial picture has the highest details and so the corresponding prediction gain is the lowest.

The MATLAB code for the 4 × 4 directional intraprediction is listed below.

```
% Example10_5.m
% Program to do 4 x 4 directional intra prediction
% using the 9 modes specified in H.264.
% The 9 modes are as follows:
% Mode 0: Vertical prediction
% Mode 1: Horizontal prediction
% Mode 2: DC prediction
% Mode 3: Diagonal down-left prediction
% Mode 4: Diagonal down-right prediction
% Mode 5: Vertical-right prediction
% Mode 6: Horizontal-down prediction
% Mode 7: Vertical-left prediction
% Mode 8: Horizontal-up prediction
% Only luma prediction is incorporated.
% No FMO is used and the macro blocks are scanned in
% raster scan order.

clear
A1 = imread('cameraman.tif');
% Make image size divisible by 4
```

```
[X,Y,Z] = size(A1);
if mod(X,4)~=0
    Height = floor(X/4)*4;
else
    Height = X;
end
if mod(Y,4)~=0
    Width = floor(Y/4)*4;
else
    Width = Y;
end
Depth = Z;
clear X Y Z
% if the input image is RGB, then convert it to YCbCr & retain
  only the
% Y component.
if Depth == 3
    A1 = rgb2ycbcr(A1(1:Height,1:Width,:));
    A = double(A1(:,:,1));
else
    A = double(A1(1:Height,1:Width));
end
% Input Y padded by 1 row on the top and
% 1 column to the left and 4 columns to the right.
B = zeros(Height+1,Width+5);
E = zeros(Height,Width); % Intra prediction error image
% Fill in the B array
B(2:Height+1,2:Width+1) = A;
B(1,2:Width+1) = A(1,:);
B(2:Height+1,1) = A(:,1);
B(1,1) = A(1,1);
for k = 2:5
    B(2:Height+1,Width+k) = A(:,256);
end
for k = 2:5
    B(1,Width+k) = B(2,Width+k);
end
% Do intra 4 x 4 prediction using 9 orientations.
% by calling the function "IntraBlkPredict".
% The returned values are: "PredictedBlk" is the predicted 4 x 4
% block corresponding to the current 4 x 4 block, and
% "Dist" is the SAD value corresponding to the best matching block.
for m = 2:4:Height+1
    for n = 2:4:Width+1
        CurrentBlk = B(m:m+3,n:n+3);
        [PredictedBlk,Dist] = IntraBlkPredict(B,m,n,CurrentBlk);
        E(m-1:m+2,n-1:n+2) = CurrentBlk - PredictedBlk;
    end
end
```

```
%
figure,imshow(E,[]), title('Intra prediction error')
%
sprintf('Prediction Gain = %5.2f dB\n',20*log10(std2(A)/std2(E)))
% Compute the prediction error histogram
[H,Bin] = hist(E,100);
figure,plot(Bin,H/sum(H),'k'), title('Intra prediction error
histogram')
xlabel('Pixel error')
ylabel('Relative frequency')


function [MatchingBlk,SAD] = IntraBlkPredict(A,Current_row,
Current_col,CurrentBlk)
% [MatchingBlk,SAD] = IntraBlkPredict(A,Row,Col,CurrentBlk)
% Function to perform 4 x 4 intra block prediction
% using the 9 directional prediction schemes used in H.264
% Input:
%   A is the padded input image
%   (Current_row, Current_col) is the coordinate of the top left
%   corner of the current block
%   CurrentBlk is the current block being predicted.
% Output:
%   MatchingBlk is the predicted block
%   SAD is the sum of absolute error (city block distance)
%
% The 9 directional prediction modes are numbered 0 through 8.
% The block that results in the least value of SAD is the
% predicted block.
%
SAD = 99999;
MatchingBlk = zeros(8,8);
for k = 0:8
    switch k
        case 0
            % Vertical Mode
            for n = 1:4
                CurrentBlk(:,n) = A(Current_row-1,Current_col+n-1);
            end
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 1
            % Horizontal Mode
            for m = 1:4
                CurrentBlk(m,:) = A(Current_row + m -1,
                                Current_col - 1);
            end
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 2
            % DC (Average) Mode
            Val = sum(A(Current_row-1,Current_col:Current_col+7))...
                + sum(A(Current_row:Current_row+3,Current_col-1))...
                + A(Current_row-1,Current_col-1);
            CurrentBlk(:,:) = round(Val/13);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 3
            % Diagonal down left Mode
            for m = 1:4
                n = m + 3;
                CurrentBlk(m,1:4) = A(Current_row -
1,Current_col+m:Current_col+n);
            end
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 4
            % Diagonal down right Mode
            for m = 1:4
                CurrentBlk(m,m) = A(Current_row -1,Current_col-1);
            end
            for m = 2:4
                CurrentBlk(m,m-1) = A(Current_row,Current_col-1);
            end
            for m = 3:4
                CurrentBlk(m,m-2) = A(Current_row+1,Current_col-1);
            end
            CurrentBlk(4,1) = A(Current_row+2,Current_col-1);
            for n = 2:4
                CurrentBlk(n-1,n) = A(Current_row-1,Current_col);
            end
            for n = 3:4
                CurrentBlk(n-2,n) = A(Current_row-1,Current_col+1);
            end
            CurrentBlk(1,4) = A(Current_row-1,Current_col+2);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 5
            % Vertical right Mode
            for m = 1:4
                CurrentBlk(m,m) = A(Current_row-1,Current_col);
            end
            for m = 1:3
                CurrentBlk(m,m+1) = A(Current_row-1,Current_col+1);
            end
            for m = 1:2
                CurrentBlk(m,m+2) = A(Current_row-1,Current_col+2);
            end
            CurrentBlk(1,4) = A(Current_row-1,Current_col+3);
            for m = 2:4
                CurrentBlk(m,m-1) = A(Current_row-1,Current_col-1);
            end
            for m = 3:4
                CurrentBlk(m,m-2) = A(Current_row,Current_col-1);
            end
            CurrentBlk(4,1) = A(Current_row+1,Current_col-1);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 6
            % horizontal down Mode
            for m = 1:4
                CurrentBlk(m,m) = A(Current_row,Current_col-1);
            end
            for m = 2:4
                CurrentBlk(m,m-1) = A(Current_row+1,Current_col-1);
            end
            for m = 3:4
                CurrentBlk(m,m-2) = A(Current_row+2,Current_col-1);
            end
            CurrentBlk(4,1) = A(Current_row+3,Current_col-1);
            for m = 1:3
                CurrentBlk(m,m+1) = A(Current_row-1,Current_col-1);
            end
            for m = 1:2
                CurrentBlk(m,m+2) = A(Current_row-1,Current_col);
            end
            CurrentBlk(1,4) = A(Current_row-1,Current_col+1);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 7
            % Vertical left Mode
            CurrentBlk(1,1) = A(Current_row-1,Current_col);
            CurrentBlk(1,2) = A(Current_row-1,Current_col+1);
            CurrentBlk(2,1) = A(Current_row-1,Current_col+1);
            CurrentBlk(3,1) = A(Current_row-1,Current_col+1);
            CurrentBlk(1,3) = A(Current_row-1,Current_col+2);
            CurrentBlk(2,2) = A(Current_row-1,Current_col+2);
            CurrentBlk(3,2) = A(Current_row-1,Current_col+2);
            CurrentBlk(4,1) = A(Current_row-1,Current_col+2);
            CurrentBlk(1,4) = A(Current_row-1,Current_col+3);
            CurrentBlk(2,3) = A(Current_row-1,Current_col+3);
            CurrentBlk(3,3) = A(Current_row-1,Current_col+3);
            CurrentBlk(4,2) = A(Current_row-1,Current_col+3);
            CurrentBlk(2,4) = A(Current_row-1,Current_col+4);
            CurrentBlk(3,4) = A(Current_row-1,Current_col+4);
            CurrentBlk(4,3) = A(Current_row-1,Current_col+4);
            CurrentBlk(4,4) = A(Current_row-1,Current_col+5);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
        case 8
            % horizontal up Mode
            CurrentBlk(1,1) = A(Current_row,Current_col-1);
            CurrentBlk(2,1) = A(Current_row+1,Current_col-1);
            CurrentBlk(1,2) = A(Current_row+1,Current_col-1);
            CurrentBlk(1,3) = A(Current_row+1,Current_col-1);
            CurrentBlk(3,1) = A(Current_row+2,Current_col-1);
            CurrentBlk(2,2) = A(Current_row+2,Current_col-1);
            CurrentBlk(2,3) = A(Current_row+2,Current_col-1);
            CurrentBlk(1,4) = A(Current_row+2,Current_col-1);
            CurrentBlk(4,1) = A(Current_row+3,Current_col-1);
            CurrentBlk(3,2) = A(Current_row+3,Current_col-1);
            CurrentBlk(3,3) = A(Current_row+3,Current_col-1);
            CurrentBlk(2,4) = A(Current_row+3,Current_col-1);
            CurrentBlk(4,2) = A(Current_row+3,Current_col-1);
```

```
            CurrentBlk(4,3) = A(Current_row+3,Current_col-1);
            CurrentBlk(3,4) = A(Current_row+3,Current_col-1);
            CurrentBlk(4,4) = A(Current_row+3,Current_col-1);
            d = sum(sum(abs(CurrentBlk-A(Current_row:Current_row+3,
                Current_col:Current_col+3))));
            if d < SAD
                SAD = d;
                MatchingBlk = CurrentBlk;
            end
    end
end
```

10.5 SUMMARY

In this chapter, we have attempted to describe briefly the motivations behind the establishment of the popular video coding standards such as MPEG. Since video sequences are spatio-temporal data, all of the standards utilize spatial as well as temporal prediction to remove or reduce the respective correlation to achieve data compression. Each standard addresses the need for a set of applications. MPEG-2 popularized TV broadcast for both SDTV and HDTV. With rapid technological developments taking place in all directions, MPEG-2 was found to be inefficient and inadequate for applications such as low-bandwidth mobile and wireless network video, interactive programs, and error resiliency. This gave impetus to the standardization of MPEG-4/H.263 video compression scheme. To improve the coding efficiency from that of the MPEG-2 standard, a new method called object-based coding was introduced. By separating the VOs and backgrounds and by coding them separately, higher coding efficiency is achieved. However, the decoder is assigned the additional task of decoding the various VOs and backgrounds and compositing them to recreate the original video.

More recently, the standard known as H.264 or AVC has been introduced to enable a video coder to distribute compressed video data for destinations using different data rate channels. H.264 is much more efficient compression-wise than MPEG-2 and can, therefore, deliver the same quality video at half the bandwidth of MPEG-2 or deliver a much superior quality video at the same bandwidth. We described some of the essential features of H.264 that are not supported by MPEG-2.

We have not covered error resiliency in MPEG-4. Due to the particular nature of the mobile or wireless networks, packets of data may be lost at random during transmission. If the video decoder has no access to the lost data, block errors will be unavoidable and the reconstructed video will be annoying. One of the important features of MPEG-4 is to make the video decoder more error resilient, thereby providing the decoder with the capability to conceal such errors and deliver a more acceptable video. For more information on error resiliency and concealment methods used in MPEG-4, refer to articles such as [30].

Within the scope of this book, we have shown several examples in this chapter to illustrate the principles involved in the video compression part of the standards.

MATLAB codes for these examples are included so that interested readers can expand on them to suit their particular needs or for academic purposes.

REFERENCES

1. ISO/IEC 11172 International Standard (MPEG-1), Information technology—coding of moving pictures and associated audio for digital storage media at up to 1.5 Mb/s, 1993.
2. ISO/IEC 13818 International Standard (MPEG-2), Information technology—generic coding of moving pictures and associated audio (also ITU-Rec.H.262), 1995.
3. "Video coding for low bit rate communication," ITU-T Recommendation, H.263(1), 1995.
4. "Coding of audio-visual objects—part 2: visual," ISO/IEC 14496-2 (MPEG-4 Visual Version 1), 1999.
5. L. Chiariglione, "MPEG-7 overview," in ISO/IEC JTC1/SC29/WG11, 2004.
6. J. Mitchell et al., *MPEG Video Compression Standard*, Kluwer Academic, 1996.
7. H. G. Musmann, P. Pirsch, and H. J. Grallert, "Advances in picture coding," *Proc. IEEE*, 73 (4), 523–548, 1985.
8. W. B. Pennebaker, "Motion vector search strategies for MPEG-1," Technical Report ESC96–002, Encoding Science Concepts, Inc., 1996.
9. A. Puri, H. M. Hang, and D. L. Shilling, "An efficient block-matching algorithm for motion compensated coding," *Proc. IEEE ICASSP*, 25.4.1–4, 1987.
10. B. G. Haskell, "Buffer and channel sharing by several interframe picture phone coders," *Bell Syst. Technol. J.*, 51 (1), 261–289, 1972.
11. B. G. Haskell, F. W. Mounts, and J. C. Candy, "Interframe coding of videotelephone pictures," *Proc. IEEE*, 60 (7), 792–800, 1972.
12. C. Reader, *Orthogonal Transform Coding of Still and Moving Pictures*, Ph.D. thesis, University of Sussex, 1973.
13. W. H. Chen and W. K. Pratt, "Scene adaptive coder," *IEEE Trans. Comm.*, COM-32(3), 225–232, 1984.
14. K. W. Chun et al., "An adaptive perceptual quantization algorithm for video coding," *IEEE Trans. Consum. Electron.*, 39 (3), 555–558, 1993.
15. A. N. Netravali and B. Prasada, "Adaptive quantization of picture signals using spatial masking," *Proc. IEEE*, 65 (4), 536–548, 1977.
16. J. O. Limb and C. B. Rubinstein, "On the design of quantizers for DPCM coders: a functional relationship between visibility, probability, and masking," *IEEE Trans. Comm.*, COM-26(5), 573–578, 1978.
17. P. Pirsch, "Design of DPCM quantizers for video signals using subjective tests," *IEEE Trans. Comm.*, COM-29(7), 990–1000, 1981.
18. H. Lohscheller, "Subjectively adapted image communication system," *IEEE Trans. Comm.*, COM-32(12), 1316–1322, 1984.
19. W. A. Pearlman, "Adaptive cosine transform image coding with constant block distortion," *IEEE Trans. Comm.*, COM-38(5), 698–703, 1990.
20. K. S. Thyagarajan and S. Morley, "Quality-based image compression," US Patent 6,600,836.
21. K. S. Thyagarajn and M. Merritt, "Contrast sensitive variance-based adaptive block size DCT image compression," US Patent 6,529,634.

22. K. S. Thyagarajan, "Adaptive block size assignment using local neighborhood properties," US Patent 6,996,283.
23. A. Puri and R. Aravind, "Motion-compensated video coding with adaptive perceptual quantization," *IEEE Trans. Circuits Syst. Video Technol.*, 1 (4), 351–361, 1991.
24. M. R. Pickering et al., "VBR rate control with a human visual system-based distortion measure," Australian Broadband Switching and Services Symposium, July 1992.
25. M. R. Pickering and J. F. Arnold, "A perceptually efficient VBR rate control algorithm," *IEEE Trans. Image Proc.*, 3 (5), 527–532, 1994.
26. N. Li et al., "Motion adaptive quantization in transform coding for exploiting motion masking effect," *SPIE Vis. Commun. Image Process.*, 1818, 1116–1123, 1992.
27. J. Lee and B. W. Dickinson, "Temporally adaptive motion interpolation exploiting temporal masking in visual perception," *IEEE Trans. Image Proc.*, 3 (5), 513–526, September 1994.
28. J. Watkinson, *The MPEG Handbook*, Focal Press, Elsevier, New York, 2004.
29. B. G. Haskell, A. Puri, and A. Netravali, *Digital Video: An Introduction to MPEG-2*, Chapman & Hall, New York, 1997.
30. A. Puri and A. Eleftheriadis, "MPEG-4: An object-based multimedia coding standard supporting mobile applications," *Mobile Netw. Appl.*, (3), 5–32, 1998.
31. T. Wiegend et al., "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, 13 (7), 560–576, 2003.

PROBLEMS

10.1. Implement B-picture coding using the MPEG-2 rules. Use the "rhinos.avi" file as the input video.

10.2. Calculate the bit rate produced by encoding the "rhinos.avi" video using MPEG-2 VLC in Tables 10.5 and 10.6. You do not have to generate the bit stream. Just read off the VLC code length to calculate the average bit rate in bits per second.

10.3. Combine both SNR and spatial scalabilities to generate a base layer and an enhanced layer. Decoding the base layer should have the lower resolution pictures at reduced SNR and decoding both layers simultaneously should yield higher resolution pictures with higher SNR values.

10.4. Apply 2D discrete wavelet transform to implement both SNR and spatial scalabilities. For instance, you can create a base layer by coding only the low–low band in the wavelet decomposition and create enhancement layers by coding the other three subbands. Similarly, you can create the base resolution layer by performing multilevel 2D DWT on the original image and coding the LL-band coefficients in the highest level and coding the rest to create enhancement layers.

10.5. Compute the histograms of the vertical (first row) and horizontal (first column) DCT predictions for the Cb and Cr components supported by H.263 and show that the amplitude range for the predicted DC and AC coefficients decreases with Laplacian-like distributions.

10.6. Implement the H.264/AVC 16×16 intrablock prediction using the four directional modes, which are vertical, horizontal, DC, and plane predictions using a real intensity image.

10.7. In H.264 intraprediction, the encoder must decide whether to use a 16×16 or a 4×4 block to code. This decision may be based on the activity measure of the 16×16 region. Use the block variance as a measure of the activity of a 16×16 block and accordingly code as a 16×16 block or four 4×4 blocks. Can you think of a better metric for the block-size selection?

10.8. In P-picture coding in H.264, motion estimation can be performed on a combination of block sizes. Implement the P-picture coding of the H.264 standard.

INDEX