INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Université d'Ottawa • University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

SUMNER, Sarah

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.Sc. (Mathematics)

GRADE - DEGREE

Department of Mathematics and Statistics

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

A Probabalistic and Statistical Method for Analysis of MPEG Encoded Video

André Dabrowski

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

A. Bose

D. McDonald

F. Théberge

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

A PROBABALISTIC AND STATISTICAL METHOD FOR ANALYSIS OF MPEG ENCODED VIDEO

By
Sarah Sumner
July 2001

A M.Sc. Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Science in Mathematics[1]

University of Ottawa
Ottawa, Ontario
Canada

[1]The M.Sc. Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

To achieve high rates of compression, the MPEG video compression standard provides methods for reducing both spatial and temporal redundancies in video data. The key step in lowering spatial redundancies is the application of the two-dimensional Discrete Cosine Transformation, while calculation of motion vectors is vital for temporal dependence reduction. Both the DCT coefficients and motion vectors can be used to provide information about the video data itself, without actually viewing or even completely decompressing the original data.

After discussing different models for background noise, statistical hypothesis tests are developed to detect either the presence or motion of an object in a small area of one picture in the video sequence using either the DCT or motion vector data. The hypothesis tests are then combined into a unified cumulative sum procedure, based on the p-values of the hypothesis tests, which will signal an alarm at the point in time when an object appears in the video under investigation.

Acknowledgments

This thesis would not have been completed without the help of others. I would like to thank my supervisor, Dr. André Dabrowski, for answering all of my many questions, and for all of his guidance and assistance. I would also like to thank Dr. David McDonald, for sharing his knowledge of cusum procedures. Gilles Lamothe deserves thanks for all of his technical and mathematical advice. I would also like to thank Serge Mister for helpful advice and many useful discussions.

This research was made possible through the generous support of the National Sciences and Engineering Council of Canada and the University of Ottawa.

Dedication

To S. M.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the years ultrasound imaging has become a technique popular with medical health professionals for performing safe and non-invasive diagnoses and monitoring. For example, ultrasound examinations of a foetus during gestation, echocardiograms of the heart, ultrasound prostate examinations as well as general abdominal imaging have all become standard medical procedures.

The application motivating this work is the use of cardiac ultrasound videos, often called echocardiograms in the medical community, during a type of knee surgery. During this procedure, a tourniquet is placed around the patient's leg above the knee and the surgery is performed. When the tourniquet is removed, pieces of debris enter the bloodstream and can be seen traveling through the heart in the echocardiogram image. Monitoring the presence of this debris is vital, as after moving through the heart, it may enter the coronary arteries, potentially causing blockages and even cardiac arrest. The treatment team must be prepared to handle these possibilities.

Now suppose, for example, a new drug is developed which is believed to reduce the amount of debris that enters the bloodstream. A statistical analysis would be appropriate to determine if this is actually true. A standard approach would be to develop a clinical trial that would measure the effectiveness of the drug by estimating the proportion of patients who experience cardiac arrest. This has two disadvantages.

First, the sample size required to detect a shift in probabilities may be large. For example, a 95% confidence interval for a population proportion of total length 0.03 would require 4269 subjects. Second, the data in such a trial ignores the actual amount of debris observed in the ultrasound images and consequently may be rather insensitive to departures from the null hypothesis. It would seem reasonable to believe that more debris in the heart implies a greater risk of cardiac arrest. Thus we are led to consider using the ultrasound videos themselves in a test of efficacy.

Examining ultrasound recordings directly may initially seem very time consuming and unreasonable but new technology has made this proposition feasible. Traditionally, ultrasound recordings have been captured on magnetic video cassettes. Recently, however, systems for storing ultrasound videos on computers in MPEG compressed format have become available. This change in storage medium should be expected to occur in other areas of medical and industrial imaging. Details about the mechanics of MPEG technology are discussed in Chapter 2.

In proposing an alternate analysis method of MPEG video data, it is desired to avoid approaches that demand that each case be individually modelled and separately examined. Such methods are time-consuming and costly, and would be inadequate for analysing the quantity of data that is generated.

Ideally, a statistical analysis procedure using ultrasound recordings directly would be automated and employ a combination of image processing methods including pattern recognition and image analysis, to complement the statistical analysis procedures. One of the important steps in such a process would be to determine the point at which debris enters the heart. This is actually a very challenging problem for several reasons. There is significant error in the image, which is already of somewhat low resolution. Moreover, it is necessary to be able to distinguish between the moving heart and the actual objects of interest, the debris.

The goal of this thesis is only to solve part of the above problem. Specifically, we will answer the question:

> Is it possible to determine when an object is moving across the field of view when the image is corrupted with noise, simply by examining the

corresponding compressed MPEG data[1]?

Multiple models of background noise will be considered, including independent normally distributed random noise, and several, temporally and spatially correlated, block-averaged noise models.

1.2 Outline

Chapter 2 describes the mechanics of the MPEG compression algorithm. A basic description of ultrasound technology and known facts about errors in ultrasound displays are the topics of Chapter 3. Simulation and the modelling of noise are discussed in Chapter 4. Chapter 5 provides a more detailed examination of the Discrete Cosine Transformation used in MPEG compression and explains hypothesis testing within I-frames. Hypothesis testing in P-frames is the topic of Chapter 6. Finally, in Chapter 7, the hypothesis testing methods are combined into a cumulative sum procedure for detecting the movement of an object in an MPEG video. Conclusions and suggestions for further research may be found in Chapter 8. An introduction to entropy is contained in Appendix A, and selected computer programs used in simulations may be found in Appendix B (Matlab code) and in Appendix C (C++ code).

1.3 Contributions of This Thesis

With the proliferation of digital video and faster hardware, it is plausible that in the near future real time analysis of video data will be possible. This thesis provides a first step in this direction, by developing a procedure to detect the presence and motion of an object in what is, effectively, compressed video data. Simulation is used to gauge the effectiveness of the proposed method.

[1]Here, the term "MPEG data" must be explained. As is outlined in Chapter 2, the final step in any MPEG encoding algorithm is lossless variable length coding, and is not appropriate for analysis. Therefore, we must either reverse this encoding (which is possible since it is lossless) or perform the analysis before it is done.

Chapter 2

An Introduction to MPEG

Officially, the acronym MPEG stands for **M**otion **P**ictures **E**xperts **G**roup, a joint committee of the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC)[15]. Colloquially, MPEG refers to a compression standard for synchronized video and audio data. In fact, one needs to be more specific. MPEG-1, officially known as ISO Standard 11172 "Coding of moving pictures and associated audio—for digital storage media at up to about 1.5 Mbits/s," is the standard used in video CDs and MP3 players. MPEG-2, ISO Standard 13818 "Generic coding of moving pictures and associated audio," was developed for use with digital television and DVD players, and MPEG-4 is the standard for Internet multimedia. Currently, work is in progress on MPEG-7, described officially as a "Multimedia Content Description Interface" and its purpose is to enable searches of image archives. Only the video aspect of MPEG-1, hereafter referred to simply as MPEG, will be discussed in this paper.

MPEG is a lossy compression scheme[1] that takes advantage of temporal and spatial redundancies in video, as well as the nature of the human audio-visual system, to greatly reduce the bandwidth required to transmit video and the size of a stored

[1] A *lossy* compression scheme is one in which data is thrown away during the compression process. Thus, the original data can not be exactly reconstructed from the compressed bit stream. Conversely, *lossless* compression algorithms are completely reversible. However, MPEG's lossy nature does not necessarily imply that there is a perceptible difference between the original video and the MPEG encoded one. Attempts are made to only discard information not processed by the human audio-visual system.

video file. Video is a sequence of still frames and between successive frames there is a high probability that much of the picture will remain essentially the same. Within a given picture, the chances that adjacent pixels are the same colour is also very high. Thus, within video, there is much redundant information. A video compressed using MPEG achieves high levels of compression by attempting to discard the maximum amount of data, while maintaining the ability to reconstruct the video with a minimal amount of error.

It is important to note that MPEG standards do not actually describe how to encode video. Only an MPEG compliant bit stream is described, and the steps that must be taken to decode it. However, it is more intuitive to first understand the process by examining how video is encoded. A good introductory work on the MPEG process is Symes [22], while more detailed analysis may be found in Watkinson [24] and Haskell, Puri and Netravali [7] as well as Bhaskaran and Konstantinides [1].

In Section 2.1, general techniques of compression that are applied in the MPEG algorithm are outlined. The representation of raw digital video data is described in Section 2.2. The details of MPEG spatial compression are covered in Section 2.3, while motion compensation specifics are found in Section 2.4.

2.1 Techniques of Compression

There are four general techniques that are employed by the MPEG standard to achieve data compression: data decorrelation, differential predictive encoding, quantization, and variable length lossless encoding.

2.1.1 Data Decorrelation: The Discrete Cosine Transformation

The concept of data decorrelation is most easily explained by first considering a data set that consists of simply a one-dimensional series of numbers. Sometimes, information that would take many data points to represent in one domain can be represented with significantly fewer data points in another domain. For example,

suppose it is desirable to transmit a sine wave. Consider using a method whereby every second a number corresponding to the value of the curve at that point in time is transmitted. With only this information, it will take many data points to accurately describe this curve. However, since it is known that a sine wave is being transmitted, if the period, amplitude, and phase shift are known, these can be used to deduce the characteristics of the complete signal. This transformation of the data from one domain to another allows a much more compact representation of the signal. The original series of values of the curve at particular points in time, that is, the set of data points in the time domain, was highly correlated, and required many points to give an accurate description. The transformed data points, that is, the set of data points in the frequency domain, give values that are completely independent and thus permit the representation of the data with a much smaller amount of information. This is the essence of data decorrelation: removing dependencies from data so that all of the data may be represented using a minimum number of points.

The best known method to shift from the time domain to the frequency domain is the Fourier transform. This same concept may be applied to images, using a two-dimensional Fourier-like transform, the Discrete Cosine Transformation (DCT). MPEG (as well as JPEG[2]) uses the most common of the four forms of the two-dimensional DCT,[3] a transform which is applied to square matrices.

Definition 2.1 (Two-Dimensional Discrete Cosine Transformation) *Let x be an $N \times N$ matrix of input values, indexed from 0 to $N-1$, where x_{ij} represents the (i,j)th entry. Let G be the $N \times N$ matrix of discrete cosine transformed values, indexed from 0 to $N-1$, where G_{kl} represents the (k,l)th entry. Then the two-dimensional discrete cosine transformation values are given by*

$$G_{kl} = \frac{1}{N} c_k c_l \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x_{ij} \cos\left(\frac{\pi(2i+1)k}{2N}\right) \cos\left(\frac{\pi(2j+1)l}{2N}\right),$$

[2]Officially, JPEG stands for Joint Photographic Experts Group, but the acronym has come to refer to the standard for encoding still pictures developed by this committee.

[3]For an excellent in-depth analysis of all four variants of the DCT transformations, see Rao and Yip [19].

where

$$c_n = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } n = 0 \\ 1 & \text{otherwise.} \end{cases}$$

A more compact representation of the DCT uses matrix notation. MPEG applies the DCT on 8×8 matrices, so let $N = 8$. Let G be the 8×8 matrix of DCT coefficients, x be the 8×8 matrix of input values, and T be the following 8×8 matrix:

$$\begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\left(\frac{\pi}{16}\right) & \cos\left(\frac{3\pi}{16}\right) & \cos\left(\frac{5\pi}{16}\right) & \cos\left(\frac{7\pi}{16}\right) & \cos\left(\frac{9\pi}{16}\right) & \cos\left(\frac{11\pi}{16}\right) & \cos\left(\frac{13\pi}{16}\right) & \cos\left(\frac{15\pi}{16}\right) \\
\cos\left(\frac{\pi}{8}\right) & \cos\left(\frac{3\pi}{8}\right) & \cos\left(\frac{5\pi}{8}\right) & \cos\left(\frac{7\pi}{8}\right) & \cos\left(\frac{9\pi}{8}\right) & \cos\left(\frac{11\pi}{8}\right) & \cos\left(\frac{13\pi}{8}\right) & \cos\left(\frac{15\pi}{8}\right) \\
\cos\left(\frac{3\pi}{16}\right) & \cos\left(\frac{9\pi}{16}\right) & \cos\left(\frac{15\pi}{16}\right) & \cos\left(\frac{21\pi}{16}\right) & \cos\left(\frac{27\pi}{16}\right) & \cos\left(\frac{33\pi}{16}\right) & \cos\left(\frac{39\pi}{16}\right) & \cos\left(\frac{45\pi}{16}\right) \\
\cos\left(\frac{\pi}{4}\right) & \cos\left(\frac{3\pi}{4}\right) & \cos\left(\frac{5\pi}{4}\right) & \cos\left(\frac{7\pi}{4}\right) & \cos\left(\frac{9\pi}{4}\right) & \cos\left(\frac{11\pi}{4}\right) & \cos\left(\frac{13\pi}{4}\right) & \cos\left(\frac{15\pi}{4}\right) \\
\cos\left(\frac{5\pi}{16}\right) & \cos\left(\frac{15\pi}{16}\right) & \cos\left(\frac{25\pi}{16}\right) & \cos\left(\frac{35\pi}{16}\right) & \cos\left(\frac{45\pi}{16}\right) & \cos\left(\frac{55\pi}{16}\right) & \cos\left(\frac{65\pi}{16}\right) & \cos\left(\frac{75\pi}{16}\right) \\
\cos\left(\frac{3\pi}{8}\right) & \cos\left(\frac{9\pi}{8}\right) & \cos\left(\frac{15\pi}{8}\right) & \cos\left(\frac{21\pi}{8}\right) & \cos\left(\frac{27\pi}{8}\right) & \cos\left(\frac{33\pi}{8}\right) & \cos\left(\frac{39\pi}{8}\right) & \cos\left(\frac{45\pi}{8}\right) \\
\cos\left(\frac{7\pi}{16}\right) & \cos\left(\frac{21\pi}{16}\right) & \cos\left(\frac{35\pi}{16}\right) & \cos\left(\frac{49\pi}{16}\right) & \cos\left(\frac{63\pi}{16}\right) & \cos\left(\frac{77\pi}{16}\right) & \cos\left(\frac{91\pi}{16}\right) & \cos\left(\frac{115\pi}{16}\right)
\end{bmatrix}.$$

Then, the DCT transformation is given by

$$G = TxT^t.$$

The "DCT coefficients," G_{kl}, can be thought of as the proportions in which the DCT basis functions, shown in Figure 2.1, need to be added to yield the original picture. Figure 2.2 provides images of several 8×8 matrices, and the corresponding transformed matrices.

In general, transforms of the form TMT^t are easily seen to be linear, and thus can be represented by a single matrix multiplication, the standard notation for linear transforms. For the two-dimensional DCT, this change from one representation to another is most compactly written through the use of the Kronecker product of matrices, often denoted by $A \otimes B$.

Definition 2.2 (Kronecker Product) *Let A be an $m \times n$ matrix and B be a $p \times q$*

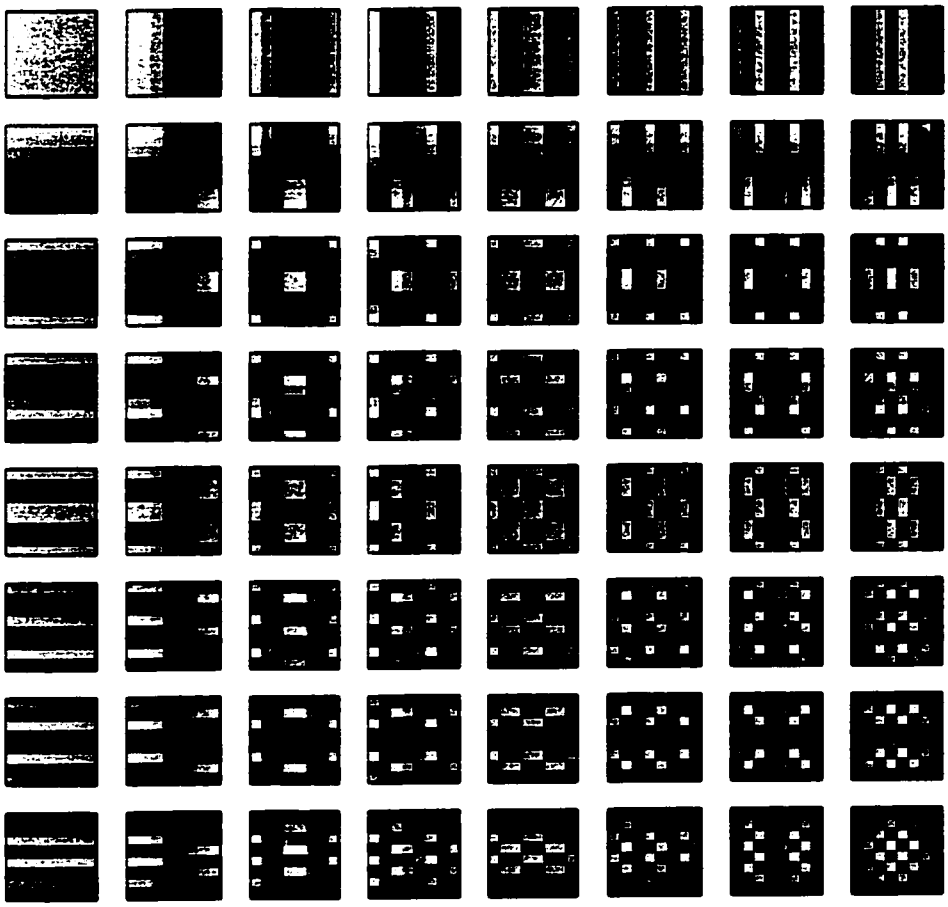


Figure 2.1: The two-dimensional DCT basis functions.

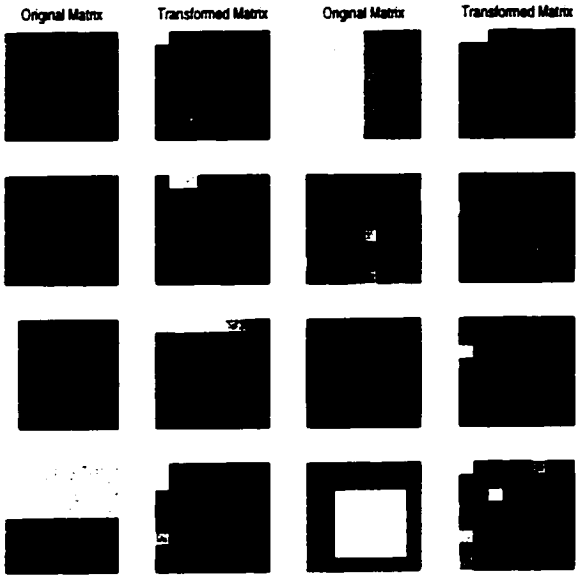


Figure 2.2: The effect of the DCT on some 8×8 matrices.
Lighter areas correspond to larger matrix entries while black represents smaller entries.

matrix. Then $A \otimes B$ *is given by the* $mp \times nq$ *matrix*

$$\begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}.$$

Now to rewrite the DCT in the form of a single matrix multiplication, we need to first represent our data in column vector format. Define $\tilde{G}$ to be the transpose of the concatenation of the rows of G, and define $\tilde{x}$ in the same way. It is not too difficult to see that the single matrix required to represent the DCT in this case, call it $\tilde{T}$, is just the Kronecker product of T with itself, that is,

$$\tilde{T} = T \otimes T.$$

So now, the DCT can be written in the form $\tilde{G} = \tilde{T}\tilde{x}$, where $\tilde{G}$ is a column vector of the DCT coefficients of length 64, $\tilde{T}$ is a 64×64 matrix, and $\tilde{x}$ is a column vector of the input values of length 64.

The DCT is reasonably effective at decorrelating data because, in a certain sense, it is asymptotically equivalent to the Karhunen-Loeve Transform, the transform which completely decorrelates Markov-1[4] data.[5] In particular, in the one-dimensional case, if the signal is given by a mean-zero, Markov-1 process, which consequently has covariance matrix $[\Sigma]_{ij} = \rho^{|i-j|}$, then the DCT is asymptotically equivalent to the Karhunen-Loeve Transform as $\rho \to 1$, (see Rao and Yip, pp. 36-38 [19]).

2.1.2 Predictive-Difference Coding

The purpose of predictive-difference coding is to transform the data into a domain on which lossless compression techniques are more powerful. Entropy is a measure of the amount of uncertainty in an observation and the key to achieving high lossless

[4]Here Markov-1 means a Markov chain in which dependencies go back only one step in time.

[5]The reason the Karhunen-Loeve Transform is not used is because it is completely dependent on the data; its transformation matrix consists of eigenvectors of the covariance matrix of the original data.

compression lies in reducing data entropy.[6] The goal of predictive-difference coding is to make some symbols much more likely than others, and thus reduce the source entropy.

The method of predictive-difference coding is quite general and can be applied in several situations where numbers are being encoded. The basic idea is to make an educated guess of what the next value to be encoded should be, and then just encode the difference between this predicted value and the actual value. The process begins by specifying the starting value, that is, the first element in the sequence to be encoded. Then, a deterministic method is used to make a prediction for the next value in the sequence using this initial value.[7] This prediction is then subtracted from the actual value of the next number. If the prediction algorithm is effective, this difference value should be somewhere near zero.

It is entirely possible for the prediction to be very wrong. But, assuming the prediction algorithm is reasonable, the probability of this happening will be reasonably low, and the quantity of numbers near zero in the encoded sequence will have increased. Thus, the entropy of this new sequence will be lower than that of the original.

Furthermore, the original sequence is recovered by simply reversing the process. Given the starting value, the next number in the sequence is predicted using the prediction algorithm. But this prediction is known to be incorrect, by exactly the amount that was transmitted in the encoded sequence of error values. So the error value is added to the predicted value to give back the original number.

Predictive-difference coding is easily illustrated with a simple example. Suppose the sequence $\{3, 10, 99, 9801, 96059602\}$ is to be encoded. The starting value is 3 and a reasonable prediction algorithm would be that the next number in the sequence is the square of the number preceding it. So the predicted value for the second element is 9, which is subtracted from the actual value 10, to give the encoded value 1. The

[6] One of the more significant results in information theory is that the average length of a uniquely decodable code can be no lower than the source entropy. For a proof of this fact, and details about entropy, see Appendix A. A good text on information theory and Shannon's theorems is Cover and Thomas[4].

[7] This procedure may be generalized to use several values in the prediction.

prediction for the third sequence element is 100, which is subtracted from the actual value of 99 to give -1. The fourth predicted value is 9801, which when subtracted from the actual value gives 0. The final predicted value is 96059601, and subtracting this value from the actual value yields 1. So the encoded sequence is

$$\{3, 1, -1, 0, 1\}.$$

Decoding is equally simple. $3^2 = 9, 9 + 1 = 10$. Next, $10^2 = 100, 100 + (-1) = 99$, $99^2 = 9801, 9801 + 0 = 9801$, and finally, $9801^2 = 96059601, 96059601 + 1 = 96059602$.

2.1.3 Variable Length Lossless Coding

Often is it possible to achieve some data compression through coding schemes that are completely reversible. Predictive-difference codes, and the DCT are both reversible, but do not actually guarantee data reduction. The final step of any MPEG encoding process involves using lossless codes to achieve additional compression. Two main types are used: modified Huffman codes and run-length codes.

A modified Huffman code is used to assign shorter bit length code words to sequences that occur with high frequency. The original Huffman coding algorithm is best explained with an example. Suppose the random variable X has outputs $\{1, 2, 3, 4, 5, 6\}$ with probabilities $\{0.5, 0.2, 0.15, 0.1, 0.025, 0.025\}$ respectively, and we wish to develop a binary code for data compression with the shortest possible average codeword length. Such a code should assign the longest codewords to 5 and 6. In fact, for an optimal code, these codewords must be the same length, and need only differ in the last bit. To actually make the Huffman code, group symbols 5 and 6 together to form a new symbol, with probability $0.025 + 0.025 = 0.05$ of occurring. Repeat this process of grouping the symbols with the smallest probabilities together to construct the binary tree shown in Figure 2.3. The codewords can then be read off directly (from right to left) from this diagram.

Run-length codes take long sequences of the same symbol x of length n, and replace them with a shorter codeword that means "insert n x's here." For example, the sequence

$$2, 2, 2, 2, 2, 2, 4, 5, 3, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3$$

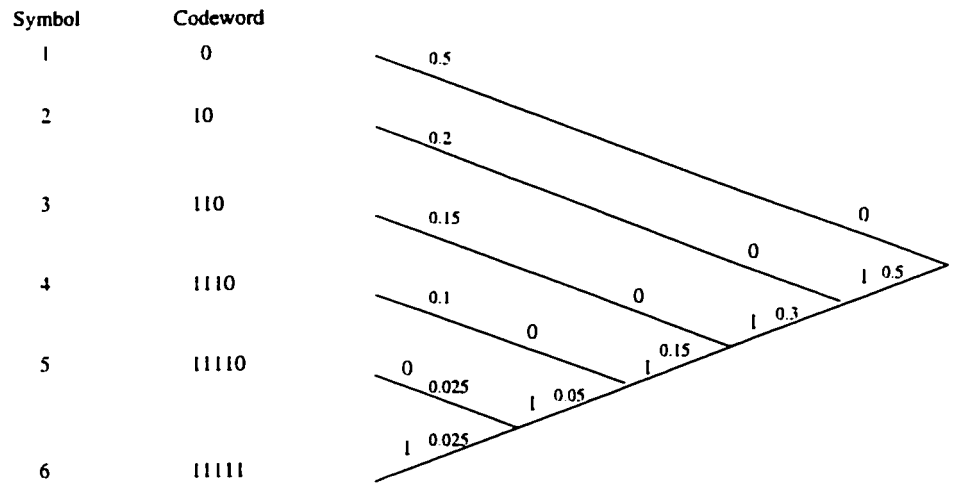


Figure 2.3: Binary tree used in creating a Huffman code.

could become

$$R2, 6, 4, 5, 3, 4, R3, 10$$

where the first symbol after the R is the one to be repeated and the second is the number of times to do the repetition.

Usually, run-length coding will be done before Huffman codes are applied.

2.2 Pixel Representation

Ultimately, video data is a sequence of pixel values, describing how each point should appear on the display screen. For a full appreciation of the MPEG compression scheme, it will be necessary to understand how exactly these pixel values are represented.

Individual colour digital pictures are usually stored in the RGB colour format, where each pixel's colour is represented by three numbers corresponding to the amount of red, green and blue light. Thus, a picture with dimensions 16 × 16 pixels would require one 16 × 16 matrix of R values, and two additional 16 × 16 matrices for the G and B values. Digital video is usually stored in the luminance-crominance system,

YCbCr. Each pixel is still represented by a triplet of values, but each component has a different interpretation. Y represents the grey-scale portion of the image, a black and white version of the original picture, and Cb and Cr contain all of the colour information.

The transformation from one colour space to the other is a simple linear relationship, given by:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

The transform to YCbCr space is not part of MPEG encoding, but is performed on many digital videos to achieve data reduction. Because the human visual system is more sensitive to changes in the brightness of a picture than in its colour, every other value in the crominance portion of the pixel representation may be discarded, without any noticeable changes in picture quality. Thus, in one 16×16 array of pixel values, there are four 8×8 arrays representing the luminance, Y, but only two 8×8 arrays for the crominance Cb, and two 8×8 arrays for the crominance Cr. This is the 4:2:2 standard, and is the most common. The 4:4:4 standard contains all eight crominance arrays.

In this thesis, because the original motivation was black and white ultrasound images, macroblocks will be assumed to contain luminance (Y) blocks only.

2.3 Spatial Compression

Having examined basic data compression techniques, we are now in a position to explore MPEG-specific details. There are two fundamental types of compression in the MPEG algorithm. Compression to reduce redundancies within one picture, *spatial compression*, is the topic of this section. Compression to reduce redundancies between different pictures, *temporal compression*, is the focus of Section 2.4.

As noted above, video is just a sequence of pictures or frames,[8] displayed in rapid succession, usually 30 frames per second, each of size about 250×350 pixels.

[8]The words *picture* and *frame* will be used interchangeably.

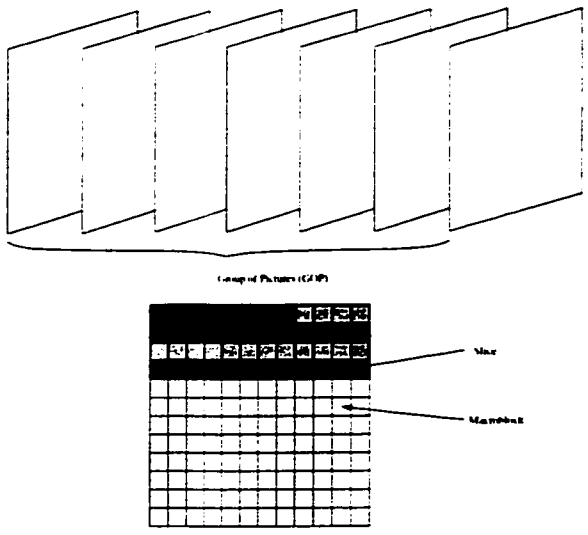


Figure 2.4: The decomposition of video.

Spatial compression is performed on the level of the picture and attempts to remove extraneous information from the picture. Each picture is first divided into a grid of 16×16 pixel *macroblocks*. Each macroblock contains subarrays of 8×8 *blocks*. As outlined in Section 2.2, each macroblock consists of four blocks of luminance values and usually two blocks of each type of crominance values. Adjacent macroblocks (and thus all of their luminance and crominance sub-blocks) are grouped together into *slices*. A slice may be as small as one macroblock, or as large as an entire picture. Several sequential pictures, usually from 10 to 30, are grouped together to form a *Group of Pictures* or GOP. See Figure 2.4 for an illustration of these concepts.

Spatial compression on one frame begins with attempts to decorrelate data. The DCT is applied to each of the eight blocks contained in a macroblock of pixels. The data is taken from the spatial domain, to the spatial frequency domain and is now represented by a set of DCT coefficients which describe in what proportions the DCT basis functions (Figure 2.1) need to be added together to produce the original picture.

These DCT coefficients are now transformed into a linear sequence using a zig-zag scan pattern, as illustrated in Figure 2.5. This pattern was chosen to maximize the probability of long sequences of zeros, and thus run-length encoding will be more

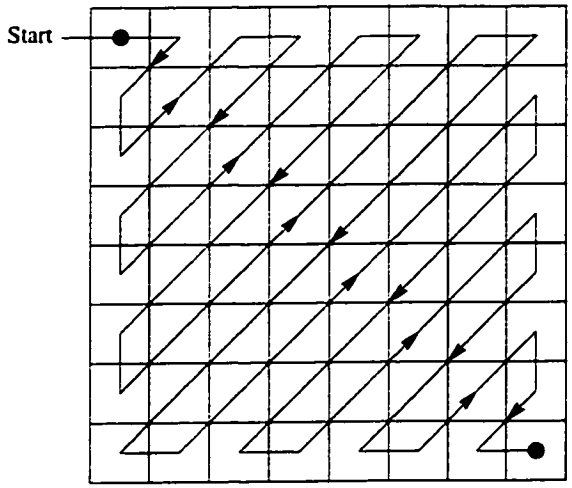


Figure 2.5: The zig-zag scan pattern of the DCT coefficients in a block.

effective on the resultant sequence.

The process of applying the DCT followed by the zig-zag scan is repeated for all macroblocks in a slice. Then, the DCT coefficients of all macroblocks within a slice are considered together. This is because for the purposes of image reconstruction, not all DCT coefficients are of equal importance. The top left DCT coefficient, corresponding to the cosine spatial frequency of zero, is called the DC (direct current) coefficient. The DC coefficient represents the average pixel value across the whole block, multiplied by a constant. If these DC coefficients were to be corrupted, the reconstructed picture quality would suffer dramatically. Thus, DC coefficients must be losslessly encoded.

So, all DC coefficients from a slice are grouped together and losslessly compressed separately from the other coefficients, using predictive difference techniques and variable length codes. Predictive differencing techniques make sense for DC coefficients, because it is expected that the average pixel values for a given macroblock will be highly correlated to the average pixel values of neighbouring macroblocks. Thus, reasonable prediction algorithms can be formed. However, these techniques on the DC coefficients alone are not sufficient to achieve significant data compression.

Applying the DCT does not in itself achieve data reduction. But, the data is now in a form in which some of it can be judiciously discarded, with negligible effect on the reconstructed picture quality. The method of choice for data elimination is quantization, which is in fact the only significant lossy component of MPEG compression.

Quantization is the process whereby the range of all possible values is divided into "bins", and all values that fall into the same bin will be reconstructed with the same value. In essence, quantization is the process of discretizing the possible values of the DCT coefficients. All remaining DCT coefficients in the slice (that is, all the DCT coefficients except the DC coefficients), sometimes called the AC coefficients,[9] are quantized.

Now, the quantization bins into which the AC coefficient values are divided need not be uniformly spaced. The DCT coefficients in the bottom right hand corner, corresponding to higher spatial frequencies and thus the finer details of the picture, can be more coarsely quantized without a substantial degradation in picture quality. So, each remaining element in the array of DCT coefficients is first multiplied by a quantization factor, and then the resulting values are quantized into equally spaced bins. The quantization factors need not remain constant throughout the encoding session.[10] Finally, the resulting bit stream of quantized AC coefficients is losslessly compressed.

In summary, to accomplish spatial compression, the DCT is applied to all blocks within a macroblock, for all macroblocks in a slice. The DC coefficients are then grouped together, and losslessly encoded. AC coefficients are first quantized, and then losslessly encoded.

2.4 Motion Compensation

To achieve high compression ratios, the MPEG standard must also exploit temporal redundancies, the redundancies between pictures. For example, in many cases, the

[9] For alternating current.

[10] In fact, it is almost certain they will not, as it is through changes in levels of quantization that MPEG controls the rate at which data is output from the encoder. This rate control is necessary to prevent buffer overflow or underflow.

background from one picture to the next changes very little. The basic idea behind motion compensation is that for a given macroblock in a picture, the closest matching macroblock in a temporally close picture will be found, and the difference between the two macroblocks will be spatially compressed.

Motion vectors for a particular frame are always calculated relative to another frame. So not all frames can be encoded using motion compensation; the process must begin somewhere. These frames that start the process off are encoded using only spatial compression techniques, and are called I-pictures, and are discussed in Section 2.4.1. Frames that use motion compensation methods with respect to reference frames that occur earlier in time are called P-pictures, and are covered in Section 2.4.2. Finally, a frame may use motion compensation methods but with respect to frames that are both ahead and behind it in time. Such frames are called B-pictures and are discussed in Section 2.4.3.

2.4.1 I-pictures

Not all pictures can be coded using motion compensation techniques, for example, the very first frame in a video sequence. The process must begin with a base reference picture. Thus, some pictures will be compressed by only removing spatial redundancies, in the manner described in Section 2.3. Such pictures are coded only with reference to themselves, and are called **I-pictures** where the I stands for *intra-coded.*

2.4.2 P-pictures

Finally, we are in a position to discuss motion compensation. Pictures that are coded with reference to another picture that occurred earlier in time are called *predictively encoded* or P-pictures. We can only encode P-pictures if we already have an anchor picture already encoded, such as an I-picture or even another P-picture.

The first step when encoding a P-picture is to decode the anchor picture, that is, the picture to be used as a reference. This is desirable because of the lossy nature of the quantization; the decompressed picture will not be identical to the

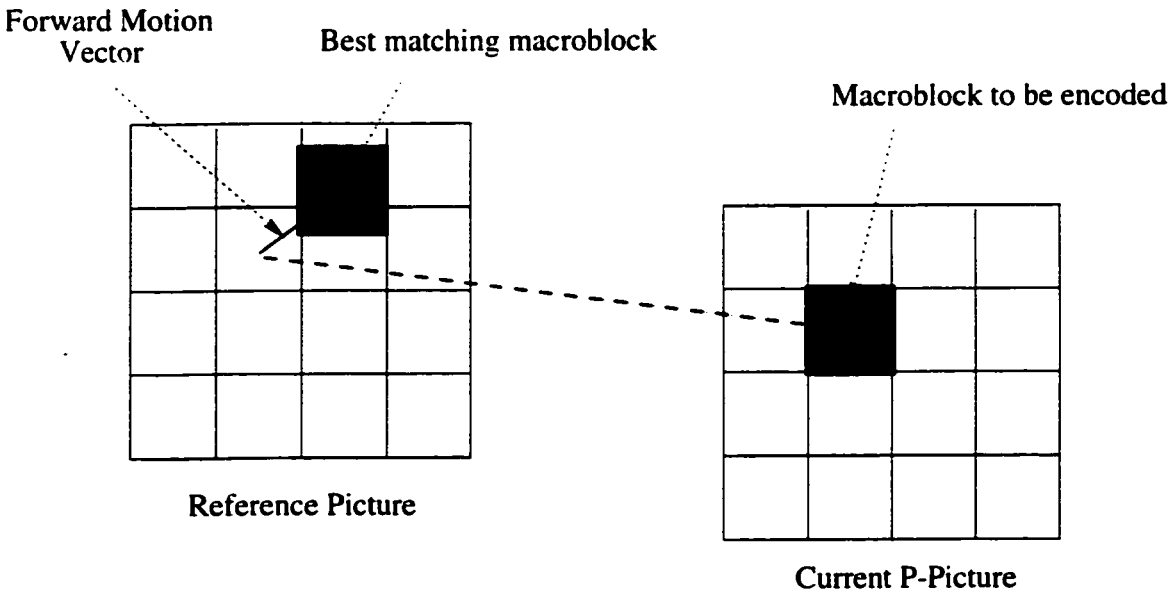


Figure 2.6: Forward motion vector.

original.[11] Consider the macroblock to be encoded, the *target macroblock*. A search of the reference picture is performed to find the macroblock that matches this target macroblock most closely.[12] If a close enough match is found, a *motion vector* is then calculated which describes the vertical and horizontal displacement of the reference macroblock from the target macroblock. This is illustrated in Figure 2.6. If no suitable match is found, the macroblock is intra-coded.

Next, the pixel by pixel difference between the reference macroblock and the target macroblock is calculated. If this difference is sufficiently small, then a "skip macroblock" message is sent. Otherwise, the resulting matrix of prediction errors has the DCT applied to it. The matrix of DCT coefficients is then zig-zag scanned, run-length, and variable-length coded, as described in Section 2.3, except that the DC coefficients are not treated separately. No special treatment of DC coefficients

[11] However, there do exist MPEG encoders that use the original frame as a reference. They may be software encoders that are processing multiple video frames in parallel. The MPEG standard does not specify which picture must be used.

[12] For a discussion of motion vector searching algorithms, see Section 2.4.5.

is required, because it has been determined empirically that it is not important to maintain an accurate average error value for reasonable reconstruction.

2.4.3 B-pictures

Searching for closely matching macroblocks in pictures that occurred earlier in time can achieve good data reduction. However, even more efficient compression can be achieved by also searching in a picture that occurs later in time, relative to the current picture, because the probability of finding a closely matching macroblock increases. Pictures that are coded with reference to two such pictures are called *bidirectionally* encoded pictures or B-pictures.

The process to encode B-pictures is very similar to that for encoding P-pictures. The major difference is that two reference pictures are used, one that occurred earlier in time, and one that will actually appear later in time. Note that this implies that MPEG does not encode pictures in a time sequential manner. So, for B-pictures, there may be motion vectors that point both forwards or backwards in time,[13] as shown in Figure 2.7. To achieve more data compression, B-pictures may be encoded with more errors than I- or P-pictures, but the additional error is tolerated because B-pictures are never used as reference frames.

[13]In fact, in some cases, a weighted average of two macroblocks, one from each time direction, can be used as an estimate of the current macroblock. So a macroblock could have two motion vectors.

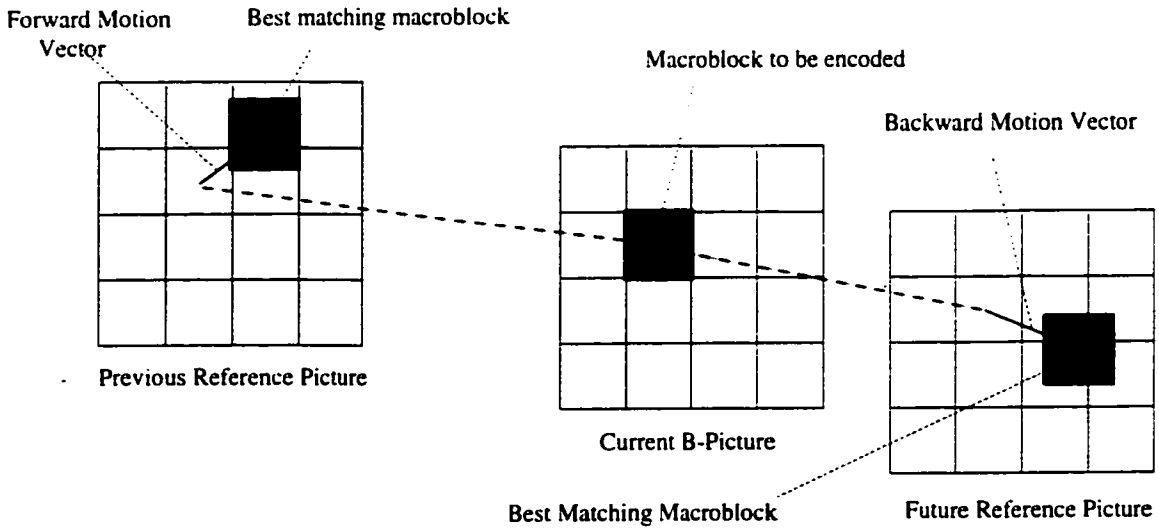


Figure 2.7: Forwards and backwards motion vectors.

2.4.4 MPEG from Start to Finish

Whether a picture will be encoded as an I-picture, P-picture or B-picture may be given by a predetermined coding pattern, or more sophisticated MPEG encoders may determine this pattern at run time. The only rule is that every slice must begin with an I-picture.

A schematic of the MPEG encoding process can be found in Figure 2.8, and the decoding process is outlined in Figure 2.9.

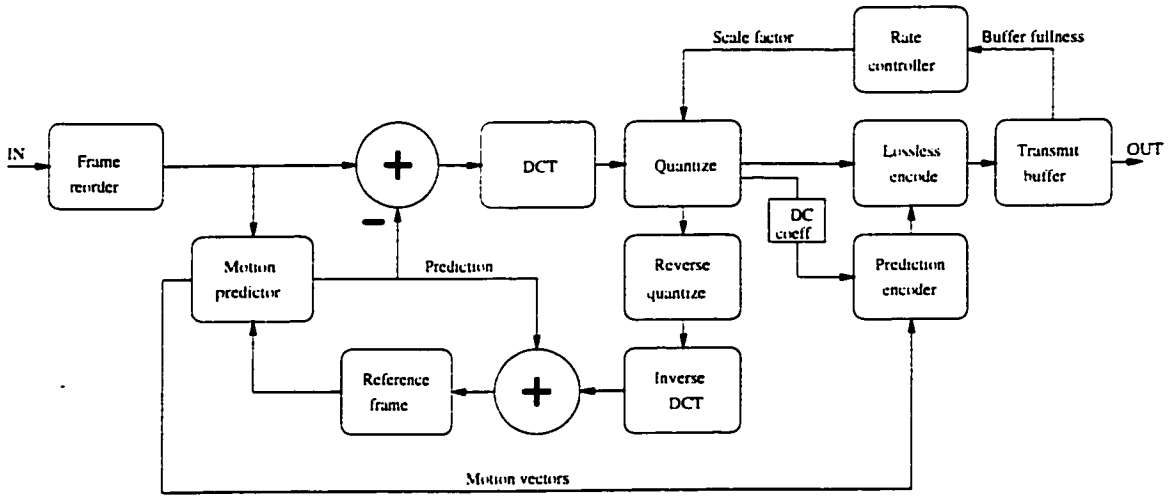


Figure 2.8: A simplified MPEG encoder.

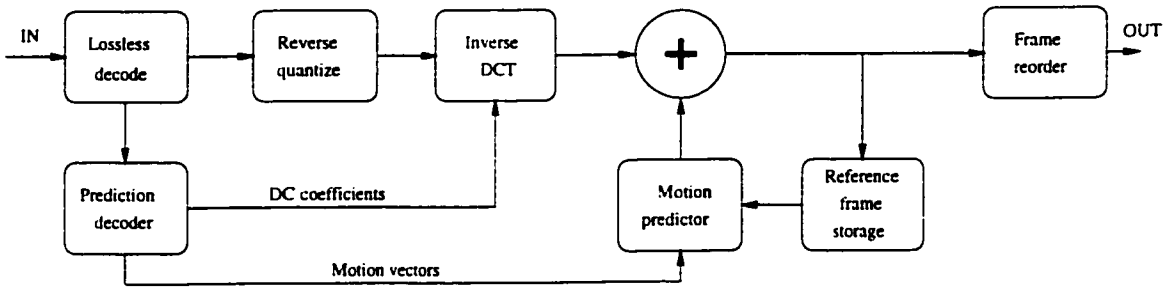


Figure 2.9: A simplified MPEG decoder.

2.4.5 Motion Vector Searching Algorithms

Because the MPEG standard only describes an MPEG compliant bit-stream, no particular motion vector searching algorithm is prescribed in the standard. However, a few examples of possible algorithms are discussed within the standard itself [8].

There are two basic parts to any motion vector searching algorithm: the macroblock matching criterion and the method of deciding which macroblocks in the reference picture should be searched for a match.

Two obvious macroblock matching criteria are the mean square error of the difference and the mean absolute difference of the two macroblocks.

Many searching techniques are possible. One is the simple exhaustive search of the reference picture. This method guarantees that the best matching macroblock will be found, but is also very computationally expensive. Although it has not been emphasized in the current discussion, the MPEG standard allows motion vector accuracy up to the nearest half-pixel. This additional accuracy is achieved by simple linear interpolation of pixel values. Thus, a modification on the full-search to half-pixel accuracy, is a full-search on full-pixel accuracy, with a final step of checking the 8 closest half-pixel shifts from the full-pixel motion vector to calculate the final motion vector to half-pixel accuracy. Not only are the number of comparisons in such a search reduced by one-fourth, but the need for interpolation is removed.

A second type of search is the logarithmic search. Here it will be explained using specific numbers, but the concept can be extended to fit different restrictions. The search begins with a search grid with a spacing of 4 pixels. Comparisons to find the closest matching macroblock at the nine points that are shifts of $-4, 0$ or 4 from the position of the current macroblock are performed. The search grid is then reduced to have a spacing of 2 pixels, and is centered around the location of the previous closest matching macroblock. Comparisons are made to find the closest matching macroblock at the nine points that are shifts of -2, 0 or 2 away from the previous closest matching macroblock. Next, the search grid is reduced to spacings of 1 pixel. The closest matching macroblock from the nine that are shifts of -1, 0 or 1 away is found. Finally, the procedure is repeated once more to find the best matching macroblock up to half-pixel shifts. This final position determines the motion vector. See Figure 2.10 for a pictorial representation.

A third type of search, that could reduce computation time even further is the telescopic search. Again, a concrete example will be used to explain it. Consider the following frame sequence shown in Figure 2.11.

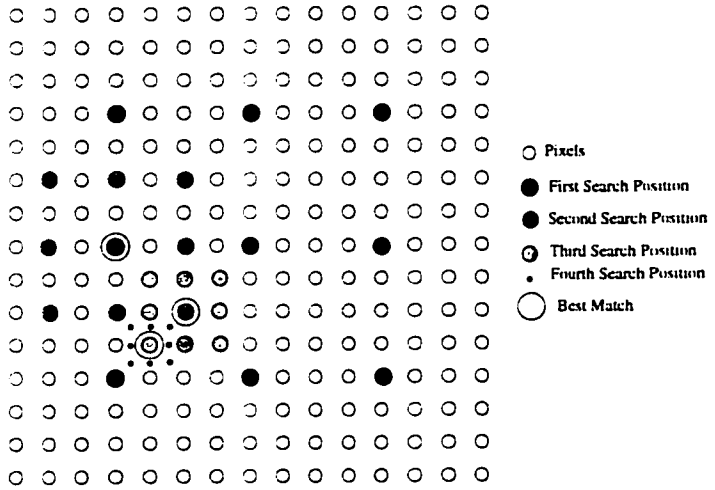


Figure 2.10: A representation of the logarithmic motion vector search.

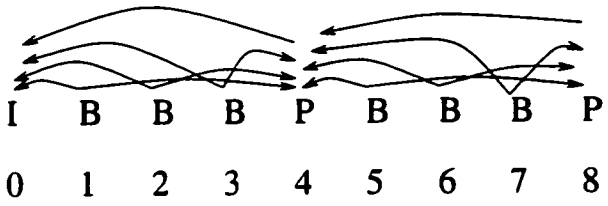


Figure 2.11: An MPEG frame sequence.
Arrows denote the frames with respect to which motion vectors are found.

Because the motion vectors of frames 1, 2 and 3 are found with respect to frames 0 and 4, the motion vectors for these frames are found as a group. First, all of the forward motion vectors for frame 1 with respect to frame 0 are calculated using some two-dimensional search (e.g. logarithmic) with zero displacement. Next the forward motion vectors for frame 2 with respect to frame 0 are found, using the same two-dimensional search, but this time centered around the displacement given by the corresponding motion vector from frame 1. The same procedure is repeated for the forward motion vectors of frames 3 and 4.

The backward motion vectors for frames 1, 2 and 3 are found in a similar manner.

All the backward motion vectors for frame 3 with respect to frame 4 are found, with the search centered with a zero displacement. Then the backward motion vectors for frame 2 with respect to frame 4 are found, with the search centered at the displacements given by the previously calculated backward motion vectors, and similarly for the backward motion vectors for frame 1.

Chapter 3

Modelling Ultrasound

As the motivation of this research is ultrasound images of the heart, a discussion of the mechanics of how an ultrasound device works and how its output can be modelled mathematically is appropriate. In Section 3.1 a general overview of ultrasound technology and image types is given. Section 3.2 describes the most common model of background noise in ultrasound images, the Rayleigh distribution. Section 3.3 examines the second order statistics of Rayleigh distributed speckle, and Section 3.4 outlines alternate proposals for interference models.

3.1 Ultrasound Technology

In the literature, the terms ultrasound and ultrasonics are used interchangeably. Both refer to the use of very high frequency sound waves for the purposes of obtaining images of the interior of usually solid objects. Over the years, ultrasound imaging technology has improved dramatically, such that for many medical applications, ultrasound image quality rivals that of MRI and CT scans and has the advantage of significantly decreased patient risk and discomfort. A good introduction to ultrasound technology is found in [17]. The most complete, but somewhat technical, description of medical ultrasound technology from its beginning to the present is Papadakis [18].

It is possible to use acoustic waves to "see" inside an object because the differences in acoustic impedance of various types of tissues cause the incident sound waves to

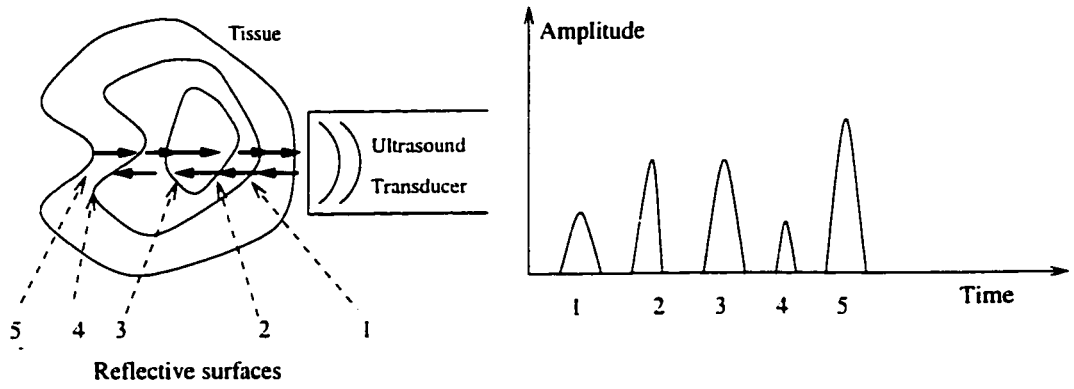


Figure 3.1: Schematic diagram of A-scan generation.
The amplitude of the reflected waves are detected and plotted versus time.

be partially reflected back, in the same way that light traveling from air to water is partially reflected[1]. A piezoelectric crystal, typically a synthetic ceramic, is used to generate sound waves in the frequency range of 1 to 20 MHz. The same crystal is then used to detect the reflected waves. The apparatus that houses the piezoelectric crystal, or sometimes just the crystal itself, is called the ultrasonic transducer.

Originally, ultrasound transducers consisted of only one piezoelectric crystal. They were well suited for the original and simplest type of ultrasound image, the A-scan, or amplitude scan. For this type of display, a short pulse of ultrasonic radiation is transmitted through the object, and the reflected echos are detected. A graph of the amplitude[2] of the reflected wave versus time is plotted on an oscilloscope. Different tissue structures can then be identified with the different peaks on the plot. See the illustration in Figure 3.1.

One major limitation of A-scans is that only one image line may be displayed at a time. Furthermore, the scan requires considerable skill to interpret accurately.

A more familiar type of ultrasound image is the B-scan; here the B stands for brightness. Modern ultrasound machines allow real-time processing of B-scans such

[1]We have all experienced this phenomena when looking at our reflection in a pool of water.

[2]It should be noted that sound waves are in fact longitudinal waves with areas of rarefication and compression, although for convenience they are usually pictured as transverse waves with peaks and troughs.

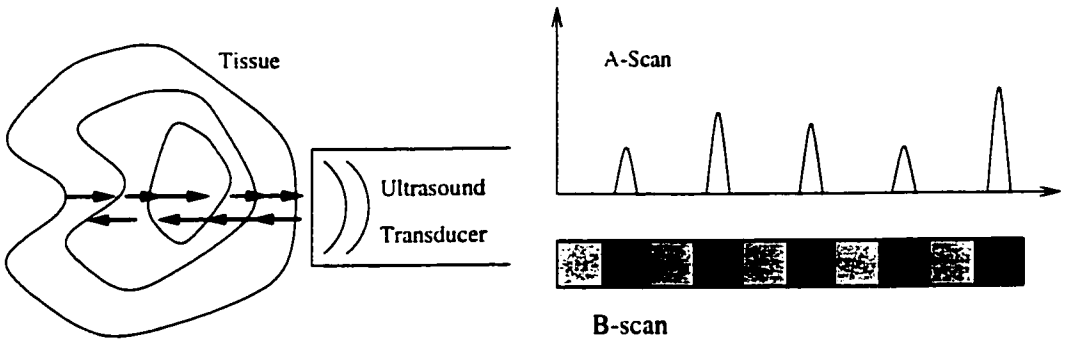


Figure 3.2: Schematic diagram of B-scan generation.

that a motion picture may be obtained. However, the first B-scans were still pictures. For this type of display, the magnitude of the amplitude of the reflected wave is used to determine the brightness of the corresponding position on the output picture, as illustrated in Figure 3.2.

Only one line of B-mode display is produced with one pulse of ultrasound radiation, but several adjacent scan lines may be combined together to form a complete two-dimensional image. Originally, two-dimensional B-scans were generated by mechanical rocking of the ultrasound transducer, or by physically scanning with the transducer across the area of interest. Such techniques were necessary when the transducer contained only one piezoelectric crystal. For well over a decade now, transducers have consisted of arrays of micro-sized piezoelectric crystals. Non-mechanical methods are now implemented to direct beams through a sector scan. In all cases, non-trivial data processing is required to construct a meaningful image.

3.2 Mathematical Model of Ultrasound Output

In this section, a model of hypothetical tissue is described and used to determine the expected pattern of speckle or background texture that is associated with most B-scans. Knowing the expected properties of background speckle is of particular importance when it is desirable to detect low contrast lesions in an image.

In general, tissue is modelled as a collection of tiny "scatterers" suspended in an otherwise uniform medium. The speckled effect associated with ultrasound images is hypothesized to result from the constructive and destructive interference of the backscattered waves from these scatterers [3]. Whether or not this observed speckle is simply undesirable background noise or actually contains some information about the tissue involved seems to depend on the conditions under which the scanning was performed, and the type of tissue being analysed [3] [23] [13] [21]. Scanning of an object under *identical* conditions will result in identical speckle patterns. However, if parameters such as transducer aperture, pulse length or transducer angulation are modified the resulting speckle patterns will be different [3].

A resolution cell is the smallest unit of resolution in an ultrasound scan. The echo detected for a particular resolution cell is represented as

$$s(t) = \Re\{ae^{i\omega t}\},$$

where ω is the incident wave frequency, t is time, a is the complex amplitude,[3] $a = Ae^{i\phi} = a_{re} + ia_i$, and $\Re(z)$ is the real part of z. Now a is the summation of all of the backscattered echos. Supposing there are n scatterers in a resolution cell,

$$a = \sum_{k=1}^{n} A_k e^{i\phi_k}.$$

If n is sufficiently large, the Central Limit Theorem may be applied to yield that a_{re} and a_{im} are both normally distributed. They are also assumed independent. It is usually assumed that the $\{\phi_k\}$ are uniformly distributed on $[0, 2\pi]$, and thus a_{re} and a_{im} have mean zero. In engineering parlance, they have the circular Gaussian distribution with probability density function (pdf),

$$f(x, y) = \frac{1}{2\pi\sigma^2} \exp\left\{-\frac{x^2 + y^2}{2\sigma^2}\right\}.$$

However, for a B-mode scan, it is the magnitude, V, of the complex amplitude a that is used to create the grey-scale picture. So $V = \sqrt{a_{re}^2 + a_{im}^2}$, and V has the

[3]The complex amplitude a my also be referred to as the *complex phasor*.

Rayleigh distribution, with pdf given by,

$$f(v) = \frac{v}{\sigma^2} \exp\left\{-\frac{v^2}{2\sigma^2}\right\}.$$

The transformation from the joint normal to the Rayleigh distribution is achieved by first transforming to polar coordinates, and then integrating out θ from 0 to 2π.

To examine the spatial dependence of speckle, we will need to examine the second order statistics of this system.

3.3 Spatial Correlation of Speckle

The material in this section is drawn largely from the original and ground-breaking 1983 paper by Wagner *et al.* [23]. In this section, the Rayleigh pdf will be parameterized as follows:

$$f(v) = \frac{v}{\Psi} \exp\left\{-\frac{v^2}{2\Psi}\right\},$$

to be consistent with notation used by Wagner and other specialists. We will proceed by first developing some general results for linear systems. Ultimately, we are interested in the statistics on the magnitude of the complex phasor that results from the detection of the various reflected backscattered echos. However, first we will need to develop the autocorrelation function for the complex phasor itself, which will then be incorporated into the autocorrelation function for the phasor magnitude. It will be argued later that the Rayleigh speckle statistics are independent in the lateral and longitudinal directions. Thus, initial results will be for one dimension only.

Since the complex summation at the transducer face is assumed linear, the output of the system will be given by the convolution of the input function with the point spread function, say $g(x)$. Let the input of our system be $a(x)$, which are the complex amplitudes from the scatterers detected by the transducer. Let the output be $\mathcal{A}(x)$. So,

$$\begin{aligned} \mathcal{A}(x) &= a(x) * g(x) \\ &= \int_{-\infty}^{\infty} a(x-y)g(y)\,dy \end{aligned}$$

It should be noted that although we will be following Wagner's notation, $a(x) * g(x)$ for convolution, the usual mathematical notation is $(a * g)(x)$.

Denote the autocorrelation of a process X at the two points x_1 and x_2 by $R_X(x_1, x_2)$. In the current situation, x_1 and x_2 represent two positions along a line. Recall,

$$R_X(x_1, x_2) = E[X(x_1)X^*(x_2)],$$

where X^* denotes complex conjugation. Related to the autocorrelation function is the autocovariance function, denoted $C_X(x_1, x_2)$, and defined by,

$$C_X(x_1, x_2) = E[\{X(x_1) - E[X(x_1)]\}\{X^*(x_2) - E[X^*(x_2)]\}].$$

A little bit of algebra shows that

$$C_X(x_1, x_2) = R_X(x_1, x_2) - E[X(x_1)]E[X^*(x_2)]. \tag{3.1}$$

The autocorrelation of the output of our system is

$$\begin{aligned}
R_{\mathcal{A}}(x_1, x_2) &= E[\mathcal{A}(x_1)\mathcal{A}^*(x_2)] \\
&= E[a(x_1) * g(x_1) \cdot a^*(x_2) * g^*)(x_2)] \\
&= E\left[\int_{-\infty}^{\infty} a(y)g(x_1 - y)\,dy \cdot \int_{-\infty}^{\infty} a^*(z)g^*(x_2 - z)\,dz\right] \\
&= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} E[a(y)a^*(z)]g(x_1 - y)g^*(x_2 - z)\,dy\,dz \\
&= \int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} E\left[a(g)a^*(z)\right] g(x_1 - y)\,dy\right) g^*(x_2 - z)\,dz \\
&= \int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} R_a(y, z)g(x_1 - y)\,dy\right) g^*(x_2 - z)\,dz \\
&= \left(\int_{-\infty}^{\infty} R_a(y, x_2)g(x_1 - y)\,dy\right) * g^*(x_2) \\
&= R_a(x_1, x_2) * g(x_1) * g^*(x_2)
\end{aligned} \tag{3.2}$$

It is possible to make some mathematical simplifications by considering the physics of our model. We will assume that the object under investigation has resolvable components that are slowly varying compared to the finer unresolvable microstructure

(the minute scatterers responsible for the speckle). Then standard results in physics permit the simplification,

$$R_a(x_1, x_2) = |a(x)|^2 \mu_a(\Delta x), \tag{3.3}$$

where $x = x_1 (\simeq x_2)$, $\Delta x = x_2 - x_1$, and $\mu_a(\Delta x)$ is an (unknown) function that describes the microscopic correlation of the tissue. Wagner [23] now substitutes (3.3) into (3.2), and performs some manipulation to claim:

$$R_{\mathcal{A}}(x, x + \Delta x) = [g(-\Delta x)|a(x + \Delta x)|^2] * [\mu_a(\Delta x) * g^*(\Delta x)]. \tag{3.4}$$

If a further assumption is made that the scatterers are uniformly distributed in the tissue (reasonable for many real-life applications) then it may be assumed that the input $a(x)$ is a constant, so

$$a(x) = a_o.$$

Substituting into (3.4) yields,

$$R_{\mathcal{A}}(x, x + \Delta x) = a_o^2 g(-\Delta x) * \mu_a(\Delta x) * g^*(\Delta x) = R_{\mathcal{A}}(\Delta x). \tag{3.5}$$

It is now clear that this process is second-order stationary, because the auto-correlation function only depends on Δx. If it is further assumed that the tissue microstructure is uncorrelated, that is it consists of randomly uniformly distributed fine particles, then $\mu_a(\Delta x) = \delta(\Delta x)$, the Dirac delta function. Then because

$$\begin{aligned} \delta(z) * f(z) &= \int \delta(z - y) f(y)\, dy \\ &= f(0) \end{aligned}$$

equation (3.5) becomes

$$R_{\mathcal{A}}(\Delta x) = a_o^2 g(-\Delta x) * g^*(\Delta x). \tag{3.6}$$

Denote the autocovariance function of $\mathcal{A}$ by $C_{\mathcal{A}}(\Delta x)$. Since the output $\mathcal{A}$ has mean zero (its distribution is Normal$(0, \Psi)$), we see from (3.1) that

$$C_{\mathcal{A}}(\Delta x) = R_{\mathcal{A}}(\Delta x).$$

The normalized autocovariance function $k(\Delta x)$ will also be needed. Define

$$k(\Delta x) = \frac{C_{\mathcal{A}}(\Delta x)}{C_{\mathcal{A}}(0)}. \tag{3.7}$$

Because B-scans depend on the magnitude of the complex phasor output, the object of real interest is $V = (\mathcal{A}_{re}^2 + \mathcal{A}_{im}^2)^{1/2}$. We would like to know the autocorrelation function for V,

$$R_V(x, \Delta x) = E[V(x)V(x + \Delta x)].$$

Here we will apply general results obtained by Middleton about joint moments of Rayleigh distributed random variables (see p. 402, [12]). This requires that we assume $g(x)$ is an even function.

$$\begin{aligned} &E[V^m(x)V^n(x + \Delta x)] = \\ &\quad (2\Psi)^{(m+n)/2}\Gamma(m/2 + 1)\Gamma(n/2 + 1) \\ &\quad \times {}_2F_1(-m/2, -n/2; 1; |k(\Delta x)|^2), \end{aligned}$$

where

$$\begin{aligned} {}_2F_1(\alpha, \beta; \gamma; x) &= 1 + \frac{\alpha\beta}{\gamma}\frac{x}{1!} + \frac{\alpha(\alpha+1)\beta(\beta+1)}{\gamma(\gamma+1)}\frac{x^2}{2!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{(\alpha)_n(\beta)_n x^n}{(\gamma)_n n!}, \quad |x| < 1, \end{aligned}$$

a hypergeometric function.

So, in the specific case at hand,

$$\begin{aligned} R_V(\Delta x) &= E[V(x)V(x + \Delta x)] \\ &= 2\Psi\Gamma(1 + 1/2)\Gamma(1 + 1/2){}_2F_1(-1/2, -1/2; 1; |k(\Delta x)|^2) \\ &= \frac{\Psi\pi}{2}{}_2F_1(-1/2, -1/2; 1; |k(\Delta x)|^2). \end{aligned} \tag{3.8}$$

We see that V is second order stationary, as $R_V(x, \Delta x)$ depends only on Δx. Note also the dependence of R_V on $R_{\mathcal{A}}$ through the normalized autocovariance function k.

From equation (3.8) and using equation (3.1), C_V can be found.

$$\begin{aligned} C_V(\Delta x) &= R_V(\Delta x) - E[V(x)]E[V^*(x + \Delta x)] \\ &= R_V(\Delta x) - (E[V])^2 \\ &= R_V(\Delta x) - \pi\Psi/2, \end{aligned} \tag{3.9}$$

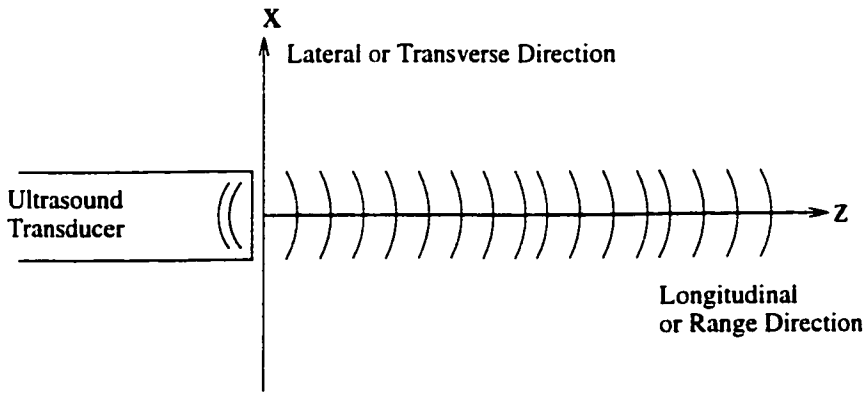


Figure 3.3: Longitudinal and lateral directions in an ultrasound scan.
The longitudinal direction in an ultrasound scan is in the direction of insonification. The lateral direction is the direction of the scan.

which follows by the stationarity of V, the fact that V must be real and because $E[V] = \sqrt{\pi\Psi/2}$.

We are now ready to begin an investigation into possible point spread functions, $g(x, z)$. Again, the physics of the situation will be used to simplify the mathematics. It will be assumed that effects of the point spread function in the transverse direction will be independent of effects in the longitudinal direction. Figure 3.3 gives an illustration of these directions.

The independence of effects in these two orthogonal directions is reasonable, as the physical principles that affect speckle output are distinct in each direction. In the transverse direction, these principles are diffraction and interference effects of coherent pressure (scalar) fields. In the range direction, it is the shape of the radio-frequency pulse that is sent and received (although dispersion and attenuation effects are ignored) that most greatly affects speckle output.

Longitudinal and lateral independence means that the point spread function $g(x, z)$ can be written as a product,

$$g(x, z) = g_x(x) \cdot g_z(z).$$

Consequently, the autocovariance function for $\mathcal{A}$ also factors,

$$C_{\mathcal{A}}(\Delta x, \Delta z) = C_{\mathcal{A}_x}(\Delta x) \cdot C_{\mathcal{A}_z}(\Delta z).$$

Known results about the physical phenomena in the range and transverse directions allow us to write down $g_x(x)$ and $g_z(z)$. In the transverse direction, approximations to the short burst of ultrasound radiation that are actually emitted are made using continuous wave theory. The shape of the transducer face is significant. For a rectangular aperture[4] Wagner [23] states that

$$g_x(x) = \frac{B \sin^2(\pi f_{ox} x)}{(\pi f_{ox} x)^2},$$

where B is a normalizing factor, $f_{ox} = D/(\lambda z_o)$, where D is the transducer aperture dimension in the x direction, λ is the wavelength of the ultrasound radiation, z_o is the distance from the transducer to the focal zone, and x is the distance from the transducer axis in the direction orthogonal to z. For a circular aperture, the function $\text{sinc}(\theta) = \sin(\theta)/\theta$ becomes $\text{jinc}(\theta) = J_1(\theta)/\theta$, ($J_1$ being the Bessel function of the first kind, of order 1) and the normalizing factor is affected.

Substitution $g_x(x)$ into (3.6) yields,

$$C_{\mathcal{A}}(\Delta x) = R_{\mathcal{A}}(\Delta x) = K_x \cdot \text{sinc}^2(\pi f_{ox} \Delta x) * \text{sinc}^2(\pi f_{ox} \Delta x),$$

where K_x is the appropriate constant, and Δx is the separation of the two points whose autocovariance is being calculated.

Fourier methods lead to the exact result (from [23]),

$$C_{\mathcal{A}}(\Delta x) = \left[\frac{K_x}{\pi(\Delta x)^2}\right]\left[1 - \frac{\sin(2\pi f_{ox}\Delta x)}{2\pi f_{ox}\Delta x}\right]. \tag{3.10}$$

In the range direction, the ultrasound pulse envelope shape is assumed to be normal$(0, \sigma_z^2)$, so

$$g_z(z) = \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left\{-\frac{z^2}{2\sigma_z^2}\right\}.$$

[4]Rectangular and circular apertures were common before transducers consisted of arrays of piezoelectric crystals. A different $g_x(x)$ would be expected for modern equipment.

Fourier transform methods then give (from [23])

$$C_{\mathcal{A}}(\Delta z) = K_z \exp\{-(\Delta z)^2/4\sigma_z^2\}. \tag{3.11}$$

It has been shown that this model fits well with experimental data, collected from B-mode scans of simulated tissue with a high concentration of scatterers [23].

The autocovariance functions can be used to determine an average speckle cell size. Denote the average speckle cell size for phasor magnitude by S_C, which is defined as

$$S_C = \int_{-\infty}^{\infty} \frac{C_V(\Delta x)}{C_V(0)}\, d(\Delta x). \tag{3.12}$$

Then, using (3.9) (3.7) (3.10) and (3.12), for phasor magnitude the average speckle cell size in the lateral direction is (from [23])

$$S_{cx} = \frac{0.9396\lambda z_o}{D}$$

and by using (3.11) instead of (3.10) (from [23])

$$S_{cz} = 2.43\sigma_z^2,$$

is the average speckle cell size in the longitudinal direction.

3.4 Non-Rayleigh Statistics of Speckle

Sometimes, the assumptions used to conclude that the magnitude of the backscattered envelope is Rayleigh distributed are not valid. Specifically, the phases of the backscattered echos may not be uniformly distributed on $[0, 2\pi]$, so the complex phasor would not have mean zero. Also, there may not be enough scatterers in the tissue to justify the application of the Central Limit Theorem to conclude that the complex phasor has a joint Gaussian distribution.

Non-uniformly distributed phases of reflected signals result when there are strong, isolated scatterers in the tissue. Then the detected envelope magnitude has a Rician distribution, with pdf

$$f(v) = \frac{v}{\sigma^2} \exp\left\{-\frac{\mu^2 v^2}{2\sigma^2}\right\} I_o\left(\frac{v\mu}{\sigma^2}\right),$$

where I_o is the modified Bessel function of 0th order [13].

For the second case, in which there are not sufficient scatters to justify applying the Central Limit Theorem, it can be shown that the magnitude of the envelope is best modelled as having pdf given by a K-distribution, with pdf,

$$f(v) = \frac{2c}{\Gamma(M)} \left(\frac{cv}{2}\right)^M K_{M-1}(cv), \quad M > 0,$$

where $K_{M-1}(\cdot)$ is the modified Bessel function of order $M-1$, c is a scattering parameter related to σ^2, M is the effective number of scatterers, given by the relation $M = n(1+\gamma)$ where n is the number of scatterers, and γ is a parameter used to describe the lack of homogeneity in the scatterers. As expected, as $n \to \infty$, (especially $n > 10$) this distribution tends to the Rayleigh distribution [13].

Chapter 4

Simulation of Error

Now that the MPEG algorithm and the mechanics of ultrasound imaging are understood, attention can be focused on solving the central problem of this thesis: development of a method of analysis for encoded MPEG video that will signal when an object begins to move through a video otherwise filled with noise. The first step towards this end is determining the types of noise that will be modeled.

In this thesis, the terms error and noise will be used interchangeably. Both are referring to the corruption of the image that is somehow fundamental to the image itself. That is, noise is what would be left if all of the parts of the image that are actually of interest were removed. This error may result either from outside sources such as interfering electromagnetic radiation or due to limitations of the image producing machinery itself.

In this chapter, Section 4.1 will describe the error models that will be pursued. Section 4.2 discusses techniques for simulating correlated random variables.

4.1 Error Models

As was discussed in Chapter 3, it is well known that ultrasound images contain speckle that is Rayleigh distributed, and it is even possible to develop reasonable approximations to the autocorrelation function for such an image, as was done in Section 3.3. Generating independent Rayleigh random variables is simple, but simulating Rayleigh

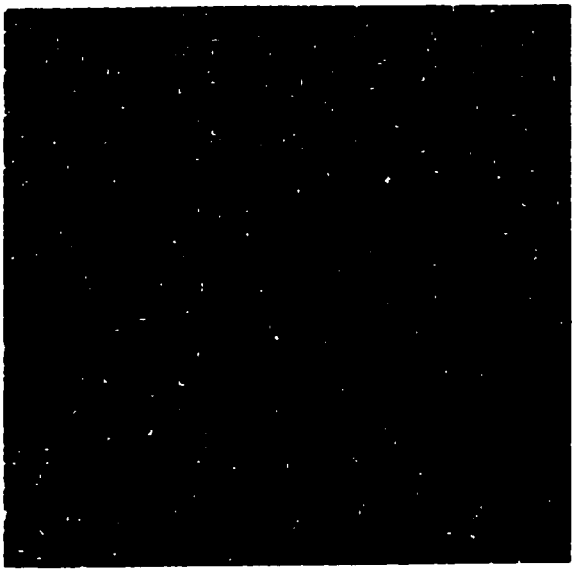


Figure 4.1: A 160 × 160 pixel screen of Rayleigh distributed independent noise.

random variables with the autocorrelation function developed in Section 3.3 is much more complex. Thus, we are led to also consider other plausible error distributions, specifically, normally distributed noise, which is easier to simulate and yet has similar characteristics to the model in Chapter 3. An image of independent Rayleigh distributed noise appears in Figure 4.1.

The simplest error model considered here is to let all pixels in a frame be independent, identically distributed normal random variables, with mean 0 and standard deviation 1. A video of this type of noise appears similar to what is seen on a television screen that is full of "snow." An example of and image of iid normal(0,1) noise is found in Figure 4.2. Note the qualitative similarity to Figure 4.1. It may be noted that there are cases in ultrasound imaging, such as when a quasi-periodic structure is being scanned, that would indeed result in normally distributed noise [23].

Image processing experts may object to permitting all real numbers as legitimate pixel values. This is because usually in image processing digital images are assumed to have pixel values only in the range between 0 and 1. To avoid additional complexity, we will not enforce this restriction. Instead, built in features of the software used

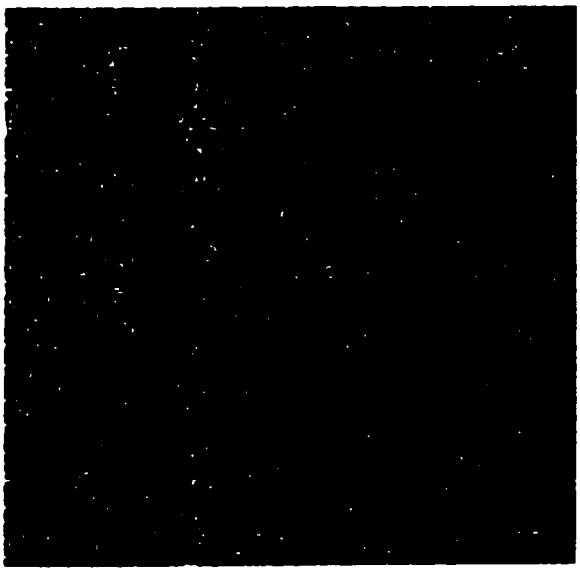


Figure 4.2: A 160 × 160 pixel screen of normal(0,1) independent noise.

to produce images (in this case Matlab) will be used to scale and shift the data to fit within (0,1) before display. Thus, setting each pixel to have mean 0 and standard deviation 1, is reasonable, because it will significantly simplify notation and calculations with no effect on the ability to produce images.

Probably the most questionable assumption in the above model is the independence among nearby pixels. The source of error in one pixel is likely to affect neighbouring pixels. Thus, the second type of error that will be considered is block-averaged iid normal(0,1) noise. This is simply a type of spatial moving average.

Consider the 3 × 3 block-averaged case first. The frame of averaged pixel values is created by averaging from a frame of independent normal(0,1) pixel values. A given averaged pixel value is calculated by taking the sum of the 3 × 3 block of pixels centered at that pixel's position in the frame of independent values, and then dividing by $\sqrt{9} = 3$, to maintain the same level of marginal variability as in the iid case. Figure 4.3 illustrates this concept. The procedure is easily extended to any $m \times n$ block, provided m and n are odd.

How block averaging will be performed along edges must be clarified. This is an

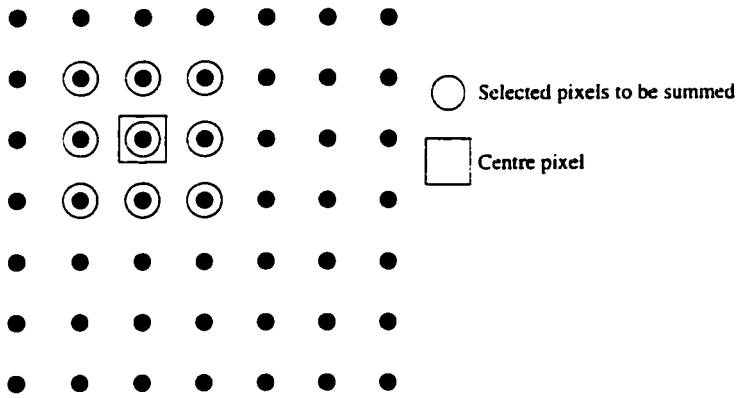


Figure 4.3: Schematic diagram of 3×3 block averaging.
The pixel in the averaged frame that has the same position as the centre pixel in the independent frame is given the value of the sum of the circled pixels divided by 3.

issue because, for example, a corner pixel only has neighbours on two sides. Because the dimensions of a frame will be quite large, and to prevent edge effects, edges will essentially be ignored. That is, to simulate an $N \times N$ frame with $m \times n$ block-averaged noise, we will begin will a $(N+m-1) \times (N+n-1)$ frame of iid normal(0,1) noise, which after averaging will produce the required $N \times N$ frame, with edges that are not a special case. Realizations of 3×3 and 5×5 averaged normal(0,1) noise can be found in Figures 4.4 and 4.5, respectively.

Block averaging also helps to justify the use of the normal instead of the Rayleigh distribution to model pixel values. Instead of modelling Rayleigh noise with the autocorrelation function from Section 3.3, a plausible simplification would be to apply block averaging to the independent Rayleigh noise. However, because of the Central Limit Theorem and the initial semi-bell shape of the Rayleigh pdf, averaging Rayleigh noise results in pixel values that have a distribution very close to being normal. Figure 4.6 shows initially independent Rayleigh noise (with the σ parameter equal to 1) averaged over 7×3 blocks, and Figure 4.7 shows initially normal(0,1) noise averaged over 7×3 blocks, which has been scaled to have the same range as the Rayleigh image. Figures 4.8 and 4.9 show the histograms of the pixel values of each image. It is clear that after scaling (as would be required to display the image) there

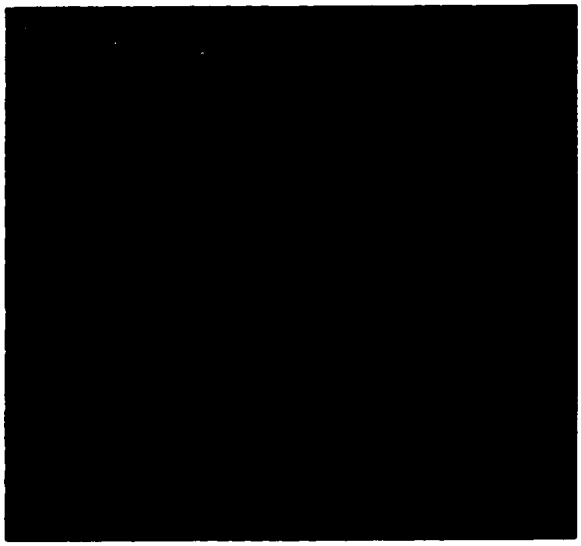


Figure 4.4: A 160 × 160 pixel screen of 3 × 3 averaged normal(0,1) noise.

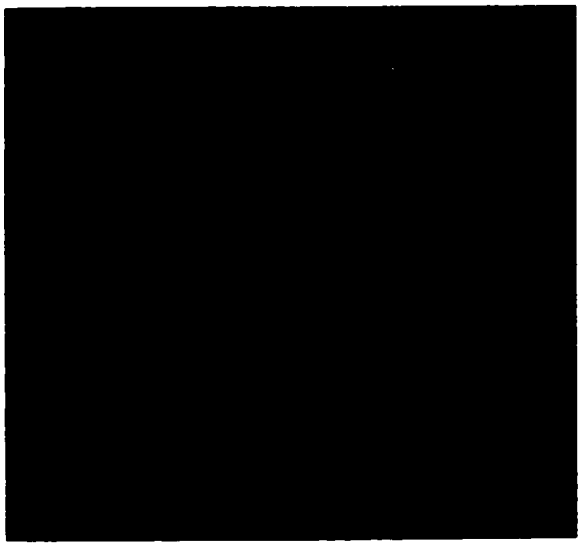


Figure 4.5: A 160 × 160 pixel screen of 5 × 5 averaged normal(0,1) noise.

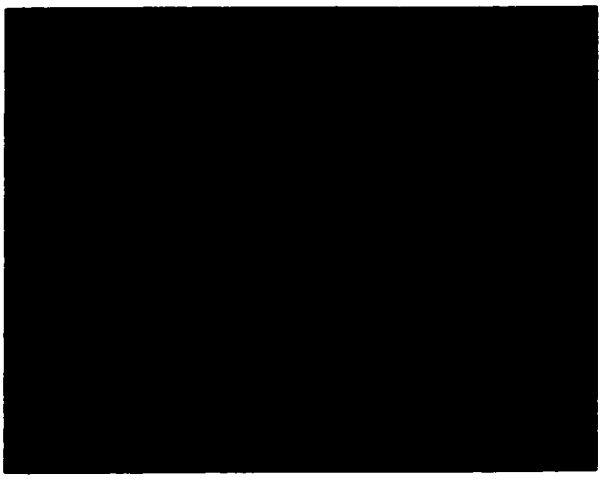


Figure 4.6: A 160 × 160 pixel screen of Rayleigh noise averaged over 7 × 3 blocks.

is little difference between these models.

With the above models, error between frames is still independent, which may also be an unreasonable assumption. Thus, the third type of error that will be modeled is an extension of the block-averaged error to include averaging in the time dimension. Thus, a particular pixel value is determined by taking the sum of at least two temporally adjacent $m \times n$ blocks, and then dividing by the square root of the total number of pixels summed. Edges are handled in the same way as the simple block averaging case.

4.2 Simulating Correlated Normal Random Variables

Simulating independent normal random variables is not difficult; mathematical software packages generally have built in functions for doing so.[1] However, suppose we want to simulate the random variables $Y_1, \ldots, Y_n$ which are normally distributed with

[1]The author would like to thank Gilles Lamothe for useful discussions about simulation, and in particular for the idea of attacking this particular problem in the manner described.

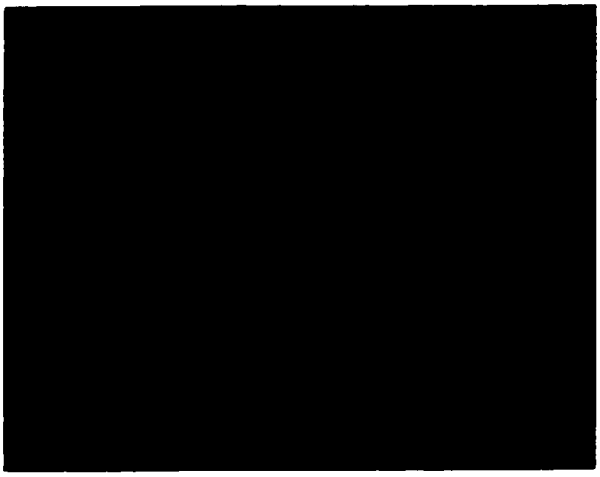


Figure 4.7: A 160 × 160 pixel screen of normal(0,1) noise averaged over 7 × 3 blocks.

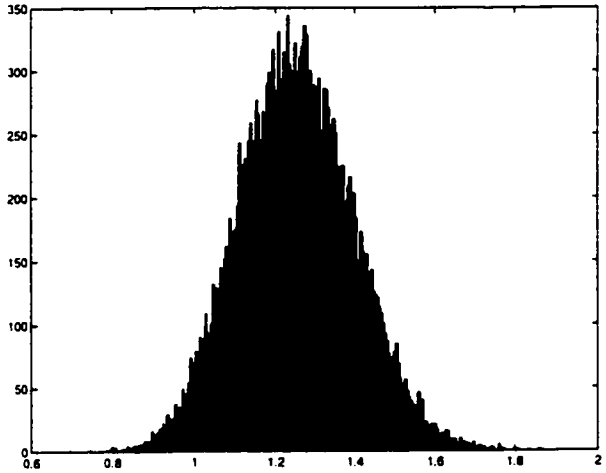


Figure 4.8: Histogram of pixel values of Rayleigh noise averaged over 7 × 3 blocks.

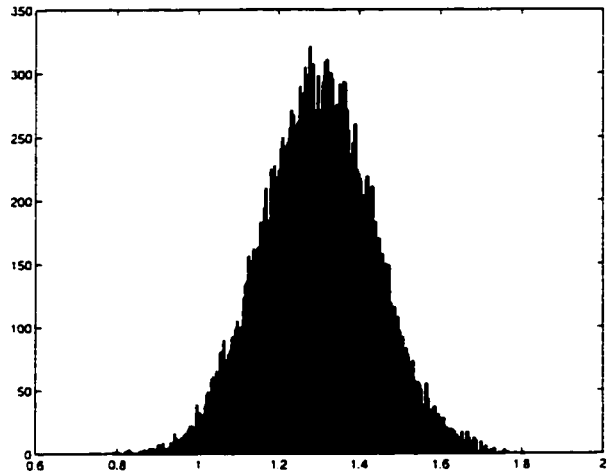


Figure 4.9: Histogram of pixel values of normal(0,1) noise averaged over 7×3 blocks.

mean 0 and covariance matrix

$$\Sigma = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 & \cdots & \sigma_{1n}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 & \cdots & \sigma_{2n}^2 \\ \sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2 & \cdots & \sigma_{3n}^2 \\ \vdots & & & \ddots & \vdots \\ \sigma_{n1}^2 & \cdots & & & \sigma_{nn}^2 \end{bmatrix}.$$

Begin by generating random variables $Z_1, \ldots, Z_n$ which are independent and normally distributed with mean 0 and variance 1. Let

$$\begin{aligned} Y_1 &= a_{11}Z_1 \\ Y_2 &= a_{21}Z_1 + a_{22}Z_2 \\ Y_3 &= a_{31}Z_1 + a_{32}Z_2 + a_{33}Z_3 \\ &\vdots \\ Y_n &= a_{n1}Z_1 + a_{n2}Z_2 + \cdots + a_{nn}Z_n \end{aligned}$$

where the numbers $\{a_{ij}\}$ are determined by the relations

$$\begin{aligned}
\sigma_{11}^2 &= a_{11}^2 \\
\sigma_{12}^2 &= a_{11}a_{21} \\
\sigma_{13}^2 &= a_{11}a_{31} \\
&\vdots \quad \vdots \\
\sigma_{1n}^2 &= a_{11}a_{n1}
\end{aligned}$$

$$\begin{aligned}
\sigma_{22}^2 &= a_{21}^2 + a_{22}^2 \\
\sigma_{23}^2 &= a_{21}a_{31} + a_{22}a_{32} \\
\sigma_{24}^2 &= a_{21}a_{41} + a_{22}a_{42} \\
&\vdots
\end{aligned}$$

Although this is not a set of linear equations, if Σ is positive definite, a solution exists and can be found by judicious substitution. If we let $Y = (Y_1, \ldots, Y_n)$, and let A be the $n \times n$ matrix such that $(A)_{ij} = a_{ij}$ for $i \geq j$ and 0 otherwise, then $Y = AZ$. See Appendix B for a Matlab function that calculates the matrix A.

When generated in this manner, the $Y_1, \ldots, Y_n$ have the covariance structure given by Σ.

This method of simulation of correlated normal random variable can be advantageous when simulating $m \times n$ block-averaged noise on smaller size screens, because it is very efficient, at least in a programming language optimized for matrix multiplication, such as Matlab. However, it is of limited use for producing very large sets of correlated normal random variables, because the matrix describing the linear transform becomes too large for practical use.

An alternate method of simulating $m \times n$ block-averaged noise is the obvious one. Start with a frame of iid normal(0,1) noise, and from it, calculate each pixel value by actually performing the summation and division. This is an undesirable method in Matlab, an interpreted language, because it runs quite slowly because of the many loops in the procedure. However, in C++, where code is optimized by the complier, this brute force method is performed with adequate speed, even on relatively large

frames (160 pixels square). Using the two-dimensional convolution function in the Matlab Image Processing Toolbox to perform the $m \times n$ block averaging greatly improves performance times.

Chapter 5

Hypothesis Testing On I-Frames

It is not difficult to think of situations in which video is corrupted by noise. Ultrasound imaging is no exception; images require a trained technician to be properly interpreted. However, an interesting question is whether it is possible to develop a mathematical method by which it may be concluded that there is something other than just noise on the screen. One approach is through an appropriate hypothesis test.

This chapter begins our examination of the effects of the MPEG algorithm on videos containing simply noise. The basic models of error that are examined were described in Chapter 4. In Section 5.1, the effects of MPEG spatial compression on these different types of noise are studied. Then, in Section 5.2 the actual hypothesis test is developed.

5.1 Effects of the Discrete Cosine Transformation

The characterization of the DCT in the form $\tilde{G} = \tilde{T}\tilde{x}$ from Section 2.1.1 in Chapter 2 is useful because from this perspective it is not difficult to determine the effects of linear transforms on a normally distributed random vector $\tilde{x}$ with mean vector $\underline{\mu}$ and covariance matrix Σ.

5.1.1 Linear Transforms of Normal Random Vectors

The first step in MPEG spatial compression is the Discrete Cosine Transformation, a linear transform of the data. For details on the effects of linear transformation on random variables see Billingsley [2].

Lemma 5.1 *A linear transformation of normal random vectors is again normal.*

PROOF: Suppose X has a normal distribution in R^k with covariance matrix Σ and mean vector $\underline{\mu}$. Without loss of generality, we may assume $\underline{\mu} = \underline{0}$. Let T be a $j \times k$ matrix. Then TX has characteristic function

$$\begin{aligned} E\left[e^{is^t(TX)}\right] &= E\left[e^{i(T^t s)^t X}\right] \\ &= \exp\left(-\frac{1}{2}\left((s^tT)\Sigma(T^ts)\right)\right) \\ &= \exp\left(-\frac{1}{2}(s^t(T\Sigma T^t)s\right). \end{aligned}$$

By the uniqueness of characteristic functions, we conclude TX is normally distributed with mean $\underline{0}$ and covariance matrix $T\Sigma T^t$. □

5.1.2 DCT Applied to Random Noise

Consider an 8×8 array of pixel values in which each component of the array is determined by the output of an independent normal(0,1) random variable. Then the covariance matrix of the 64 pixels is simply the 64×64 identity matrix, I_{64}. The covariance matrix after the DCT is

$$\tilde{T} I_{64} \tilde{T}^t,$$

where $\tilde{T}$ is the matrix defined on page 10. However, in this case, because the original DCT matrix T was orthogonal, and since $\tilde{T} = T \otimes T$, then is $\tilde{T}$ is also orthogonal. Thus $\tilde{T} I_{64} \tilde{T}^t = I_{64}$. So, the statistical distribution of the pixel data is identical, before and after the DCT.

In the $m \times n$ block-averaged noise case, the resulting covariance structure after the application of the DCT is easily calculated by applying the above lemma. However,

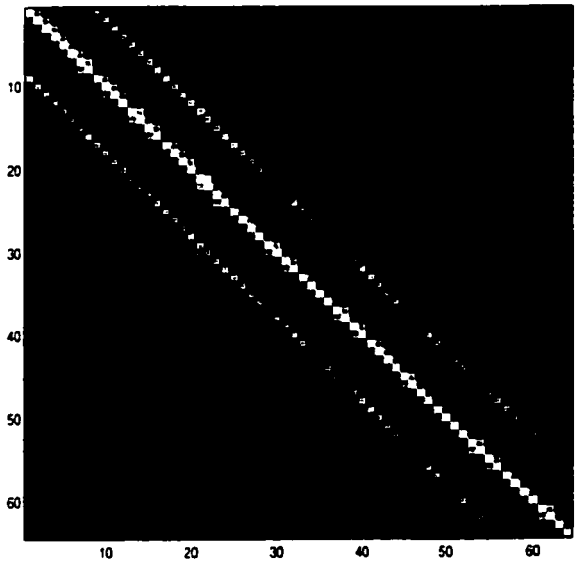


Figure 5.1: A representation of the covariance matrix for a 3×3 block average process. Lighter squares correspond to larger matrix entries.

since the covariance matrix is no longer the identity matrix, the distribution of the DCT coefficients will be different from that of the original pixels. The covariance matrix of a 3×3 block averaged frame of dimensions 64×64 pixels can be visualized as in Figure 5.1.

5.2 Developing a Test for I-frames

Consider one frame in an MPEG-1 video sequence that has been intra-coded, that is, an I-frame. If the variable length and predictive difference coding are undone, a set of (quantized) DCT coefficients remains. Ignoring the complications from the quantization (by simply assuming it never happened, or that the quantization is very fine), a simple hypothesis test can be developed. Since coding is done on the level of the macroblock, each hypothesis test will be developed for application to a single macroblock.

Let the hypothesis test be defined as follows:

H_o : The macroblock contains just noise.

H_1 : The macroblock contains something other than noise.

Let DCT_{ij} represent the DCT coefficient of the (i,j)th pixel. It is now required that a suitable test statistic be developed that will be reasonably effective at detecting the presence of an object in the background noise. Unfortunately, determining the power function[1] of a statistic in general is a difficult problem, and thus one must usually be content with testing the power against a specific alternate hypothesis.

Five possible test statistics will be considered. Here it is assumed that the indexing on macroblocks commences at 0.

$$
\begin{aligned}
K_1 &= \max_{i=0,\ldots,15;j=0,\ldots,15} \{|DCT_{ij}|\} \\
K_2 &= \max_{i=0,\ldots,3,8,\ldots,11;j=0,\ldots,3,8,\ldots,11} \{|DCT_{ij}|\} \\
K_3 &= \max_{i=0,8;j=0,8} \{|DCT_{ij}|\} \\
K_4 &= \sum_{i=0}^{15}\sum_{j=0}^{15} DCT_{ij}^2 \\
K_5 &= \sum_{i=0,\ldots,3,8,\ldots,11} \sum_{j=0,\ldots,3,8,\ldots,11} DCT_{ij}^2
\end{aligned}
$$

These choices for potential test statistics were selected because of the general observation that having pixel values with large magnitudes in the original block results in DCT coefficients with large magnitudes after the transform, particularity in the upper left-hand corner of the transformed matrix. Refer to Figures 2.2, 5.2 and 5.3 for examples.

K_1 and K_4 use all of the available data, but are more computationally expensive than K_2, K_3 and K_5. Notice that K_3 just examines the DC coefficients for each block.[2] In all cases, the null hypothesis will be rejected if the observed value of the test statistic is too large.

To compare the test statistics, simulation is necessary. If all pixels are independent and have the normal(0,1) distribution, it is a well known fact that K_4 has a χ^2

[1]The power function is the function that gives the probability of rejecting a the null hypothesis

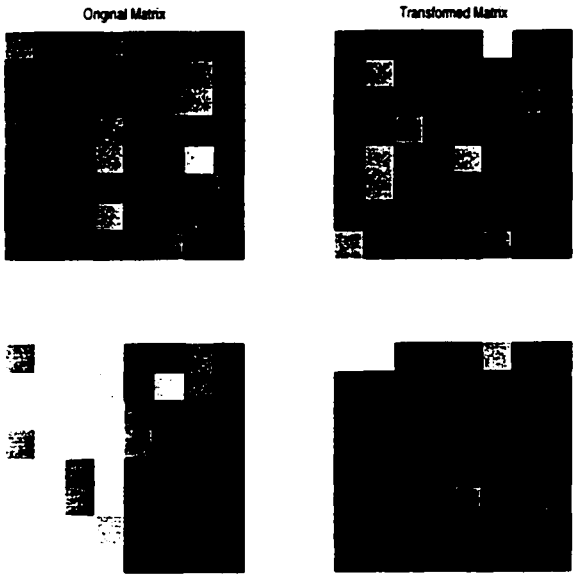


Figure 5.2: Images of 8×8 blocks before and after the DCT is applied. Note the extreme DCT_{ij} values in the upper left hand corner of the transformed image (right side) when there is something other than noise present in the original image (left side).

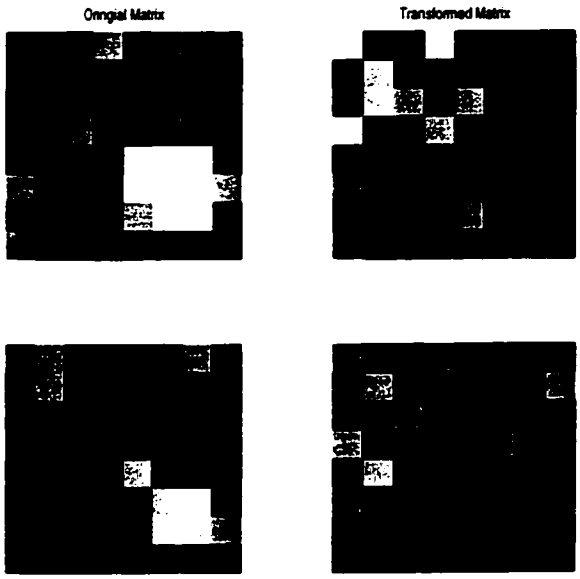


Figure 5.3: More images before and after the DCT is applied.

distribution with 16^2 degrees of freedom (the number of pixels in the macroblock). Thus, rejection regions can easily by found in standard tables. However, in all other cases, simulations are required. Macroblocks with pixel values distributed according to the following noise models were simulated, and the statistics calculated:

Type 1 Noise Independent normal(0,1) noise

Type 2 Noise Independent normal(0,1) noise averaged over 3x3 blocks

Type 3 Noise Independent normal(0,1) noise averaged over 5x5 blocks

Type 4 Noise Independent normal(0,1) noise averaged over 3x7 blocks

Type 5 Noise Independent Rayleigh noise with $\sigma = 1$.

for all the various possibilities of the alternate hypothesis.

[2]Had K_3 turned out to be a reasonable test statistic, the method for testing on I-frames and P-frames could have been consolidated, as there exist efficient algorithms for extracting DC coefficients from P- and B-frames from encoded video data.

Types 1, 2 and 3 of noise were chosen as plausible types of corruption that may occur to data. Type 4 noise was chosen to visually correspond more closely to Rayleigh distributed ultrasound speckle with spatial correlation as outlined in Chapter 3. Type 5 noise, although independent, is one type of Rayleigh distributed noise.

To determine how each test statistic behaves under each of the error models, 5000 macroblocks of each type of error were simulated, and the each statistic calculated using this simulated data. The histograms of these results can be seen in Figures 5.4, 5.5, 5.6, 5.7 and 5.8.

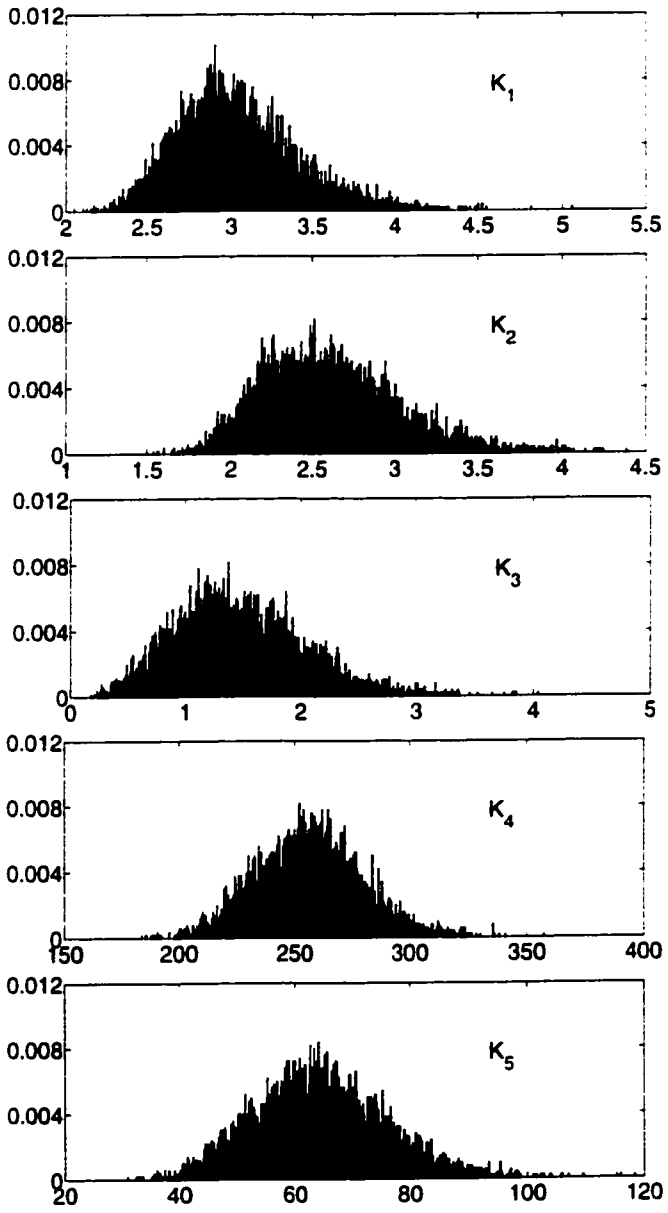


Figure 5.4: Histogram of possible test statistics under Type 1 noise. In each case, the sample consists of 5000 test statistics.

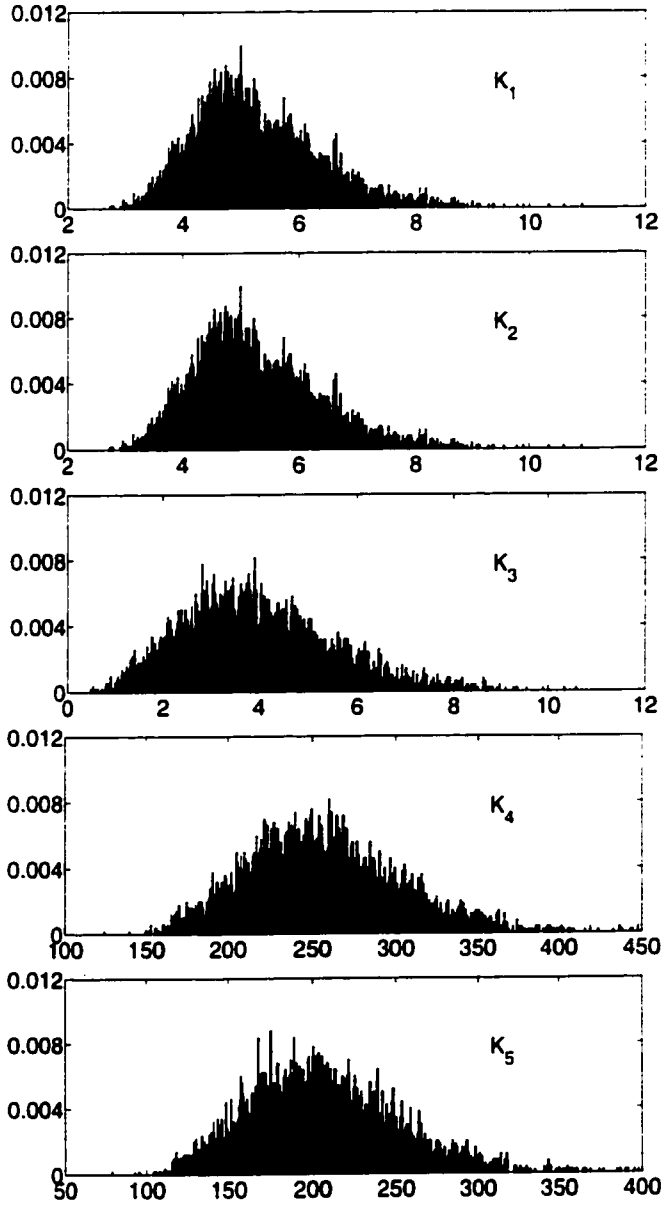


Figure 5.5: Histogram of possible test statistics under Type 2 noise. In each case, the sample consists of 5000 test statistics.

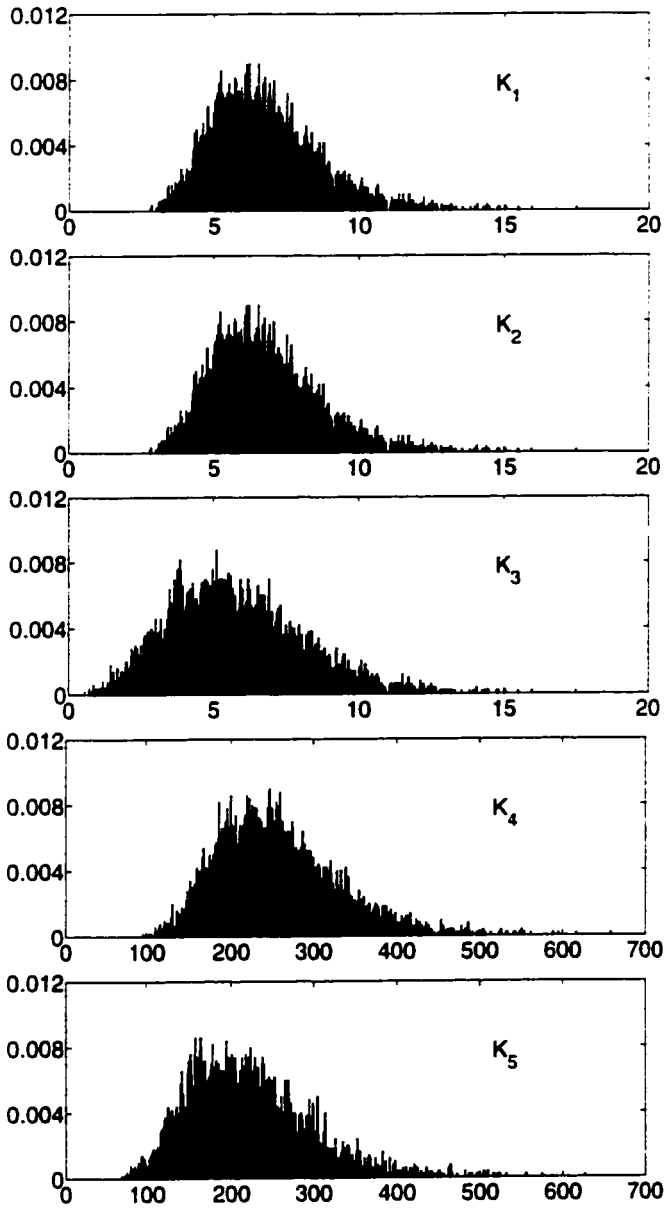


Figure 5.6: Histogram of possible test statistics under Type 3 noise. In each case, the sample consists of 5000 test statistics.

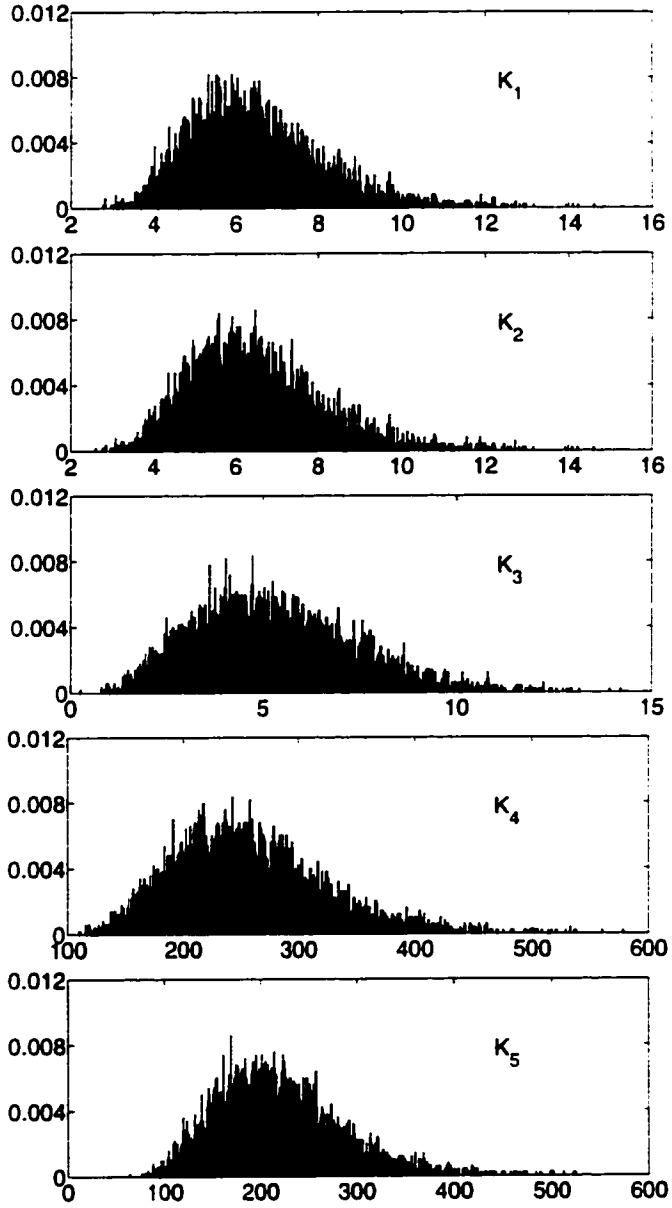


Figure 5.7: Histogram of possible test statistics under Type 4 noise. In each case, the sample consists of 5000 test statistics.

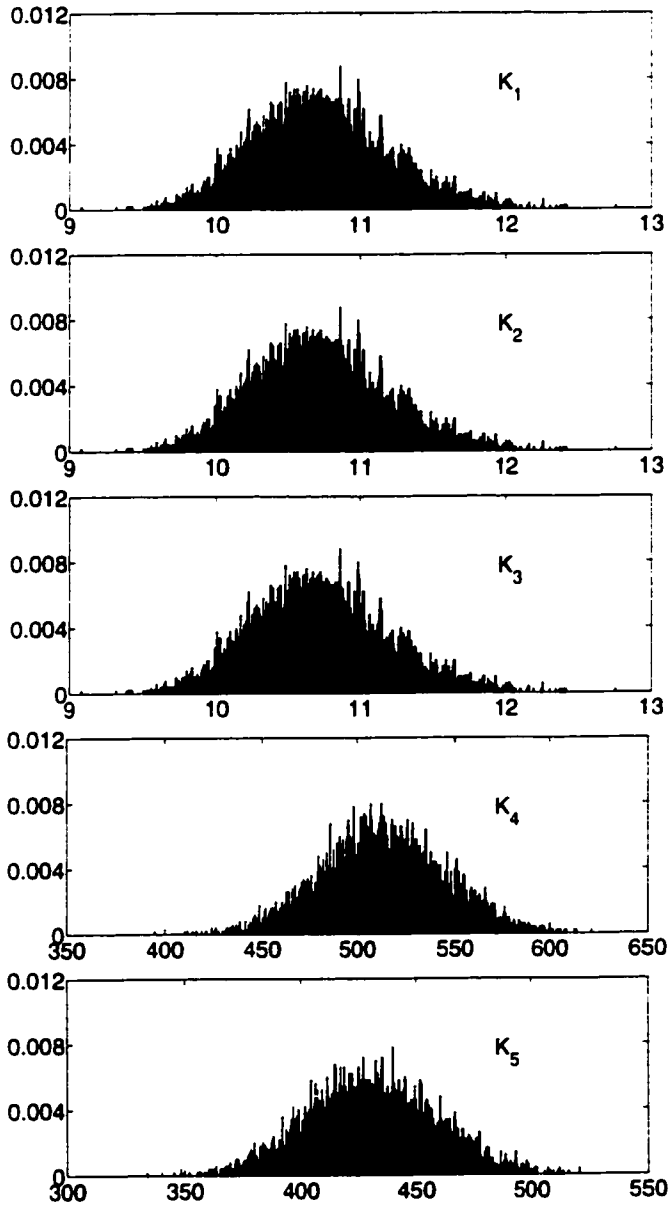


Figure 5.8: Histogram of possible test statistics under Type 5 noise. In each case, the sample consists of 5000 test statistics.

To test the effectiveness of each test statistic under possible instances of the alternate hypotheses, simulations of a small box of size 8 pixels by 8 pixels and of intensity value 3 centered in the middle of a macroblock, against the background of each noise type were performed. The cumulative distribution functions (cdf) of K_1, K_2, K_3, K_4 and K_5 with this small block in the picture and with each type of background noise may be found in Figures 5.9, 5.10,5.11, 5.12, and 5.13.

The same test was repeated but with a larger box of dimensions 12 pixels by 12 pixels and of intensity value 3 centered in the macroblock. The corresponding cdfs are in Figures 5.14, 5.15, 5.16, 5.17 and 5.18.

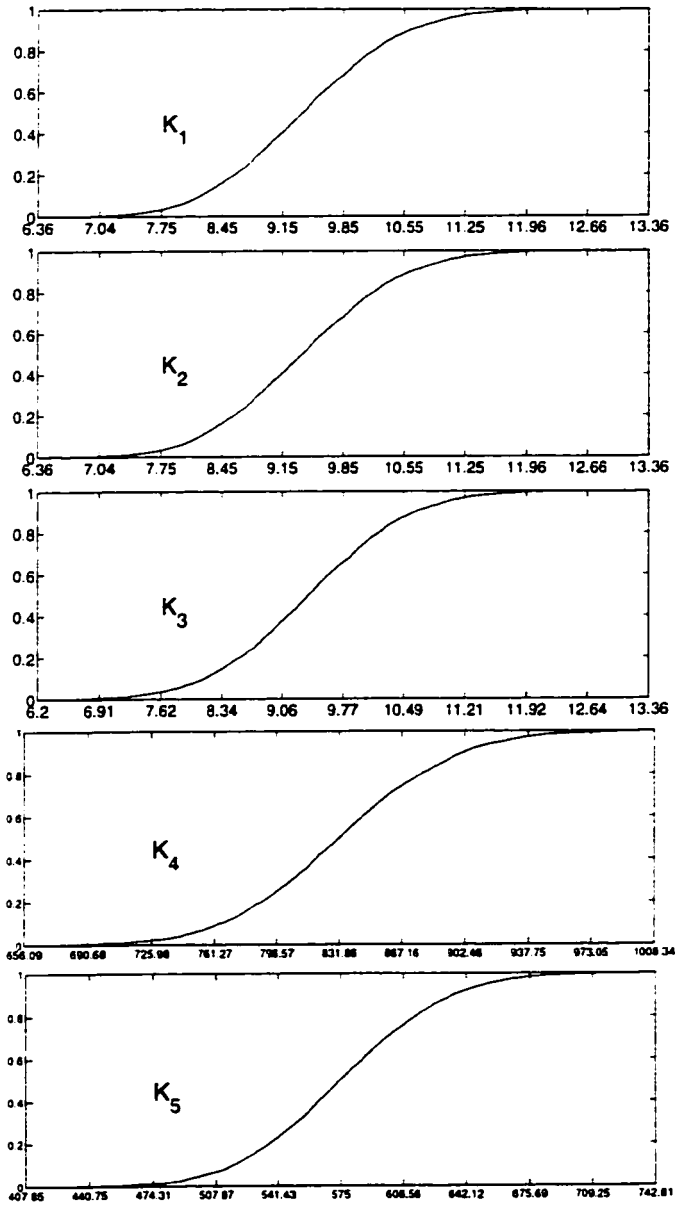


Figure 5.9: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 1 noise, with a box of dimension 8×8 added.

The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

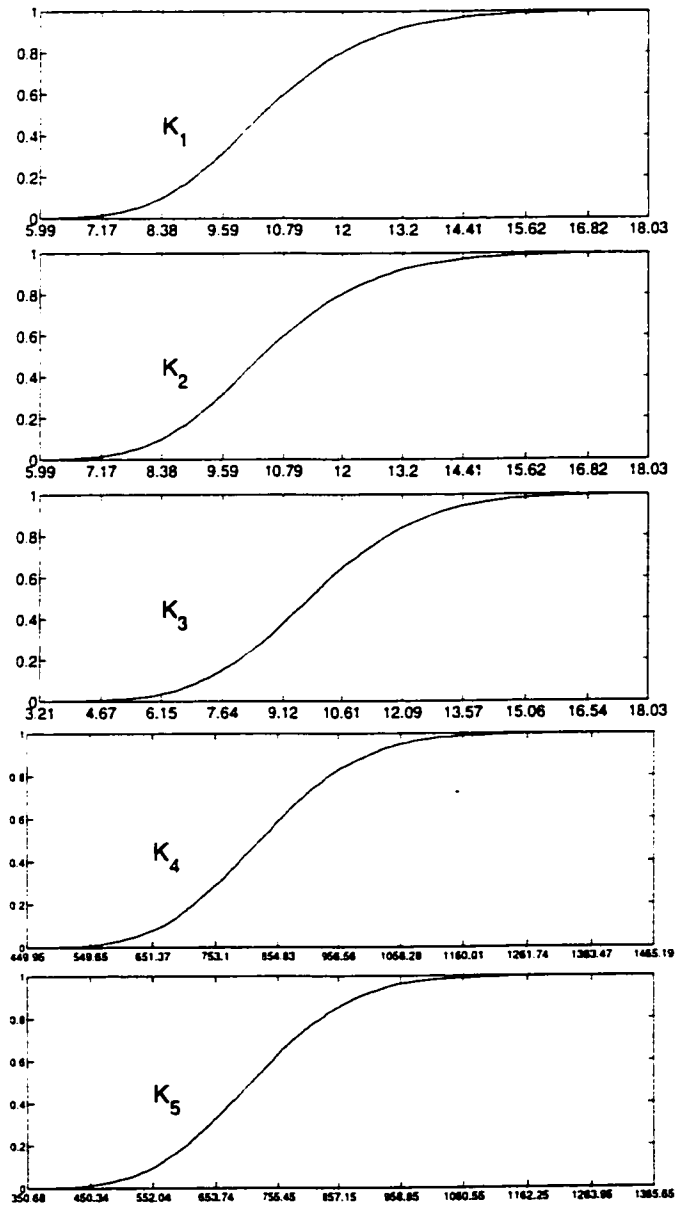


Figure 5.10: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 2 noise, with a box of dimension 8×8 added.

The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

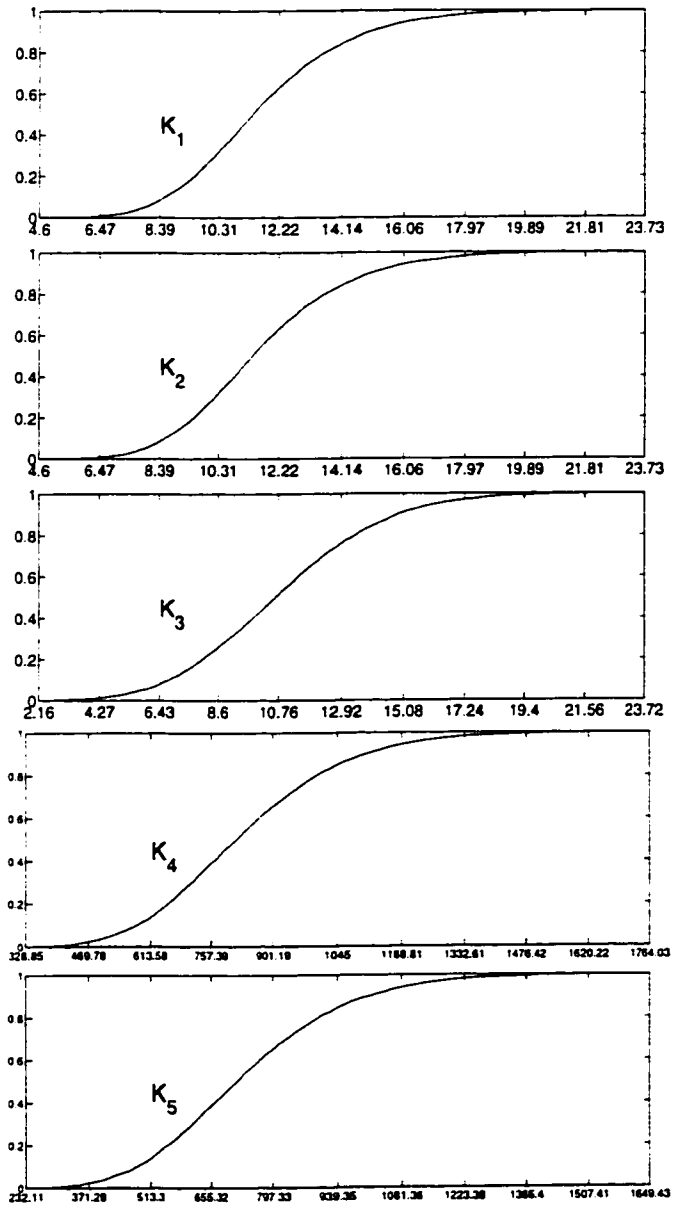


Figure 5.11: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 3 noise, with a box of dimension 8×8 added.

The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

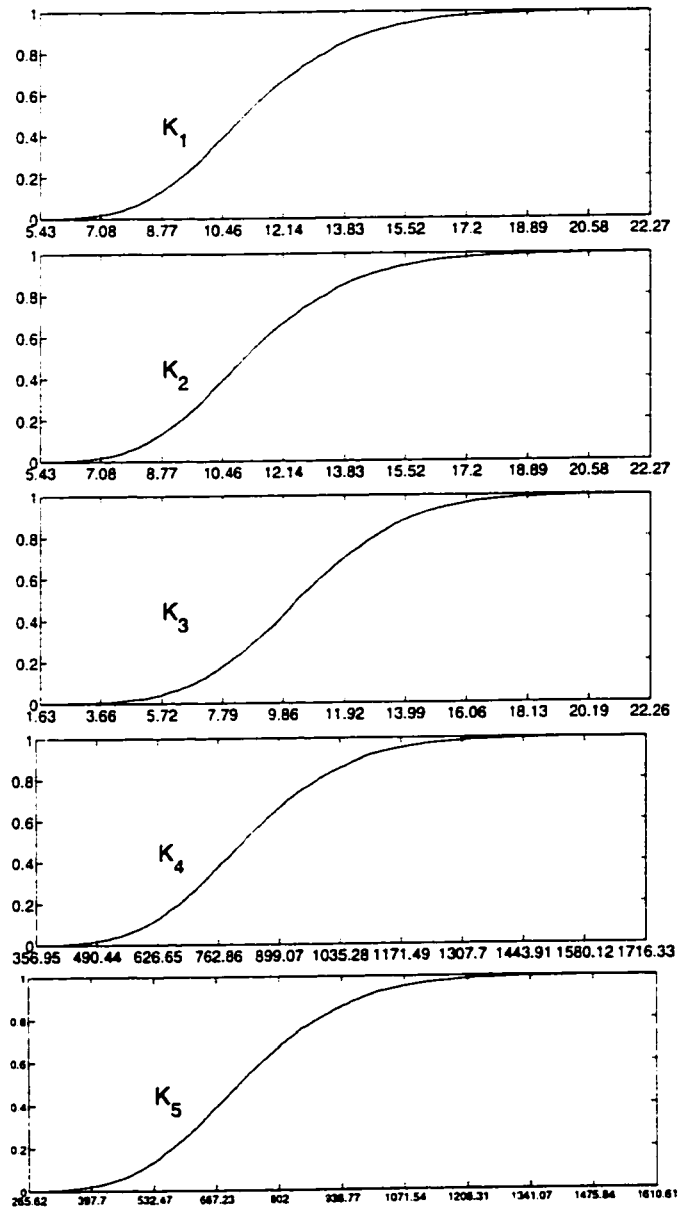


Figure 5.12: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 4 noise, with a box of dimension 8×8 added.
The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

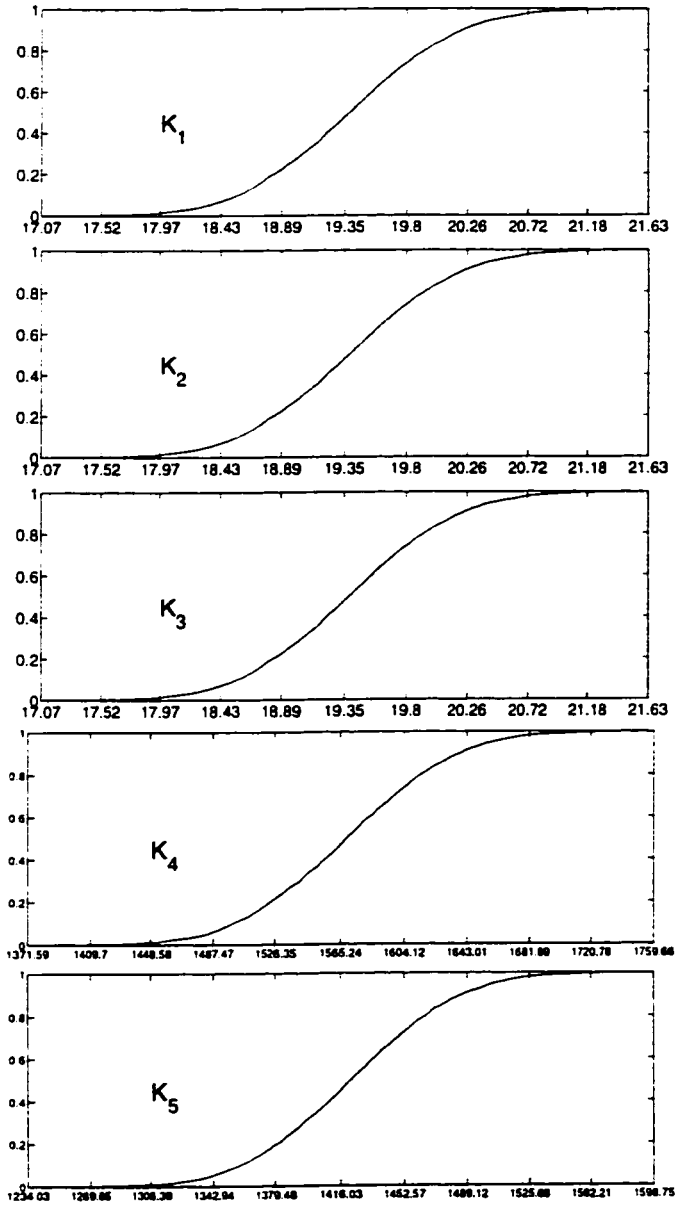


Figure 5.13: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 5 noise, with a box of dimension 8×8 added.
The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

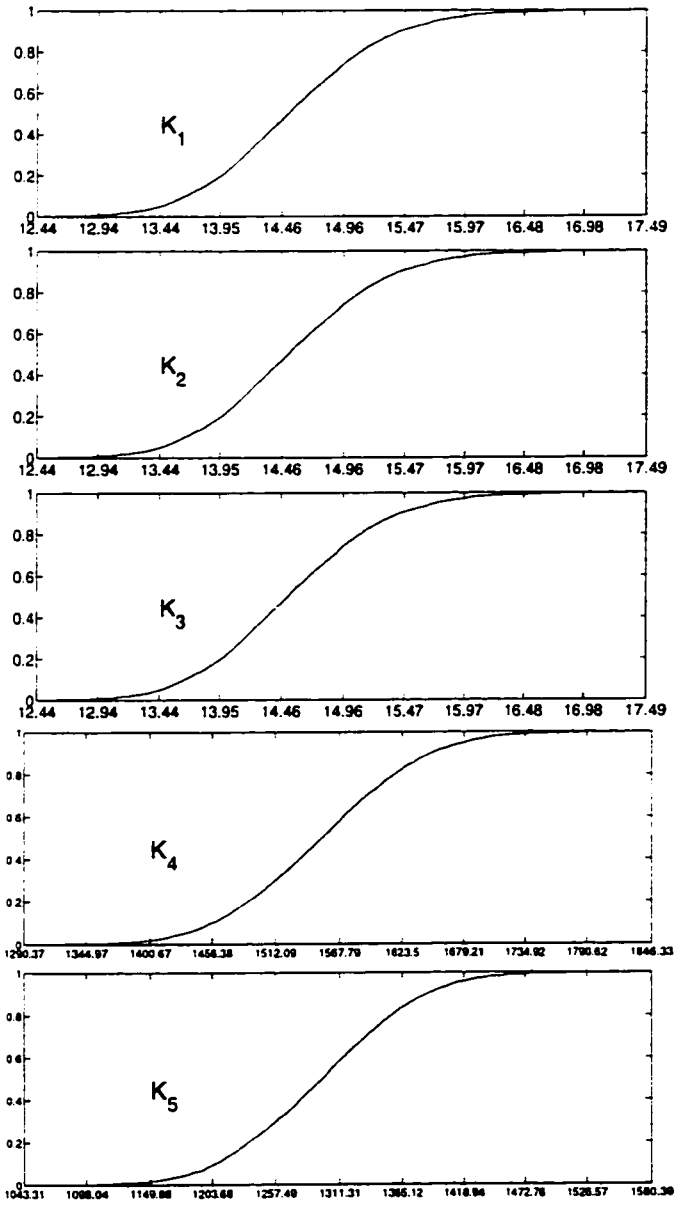


Figure 5.14: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 1 noise, with a box of dimension 12×12 added.

The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

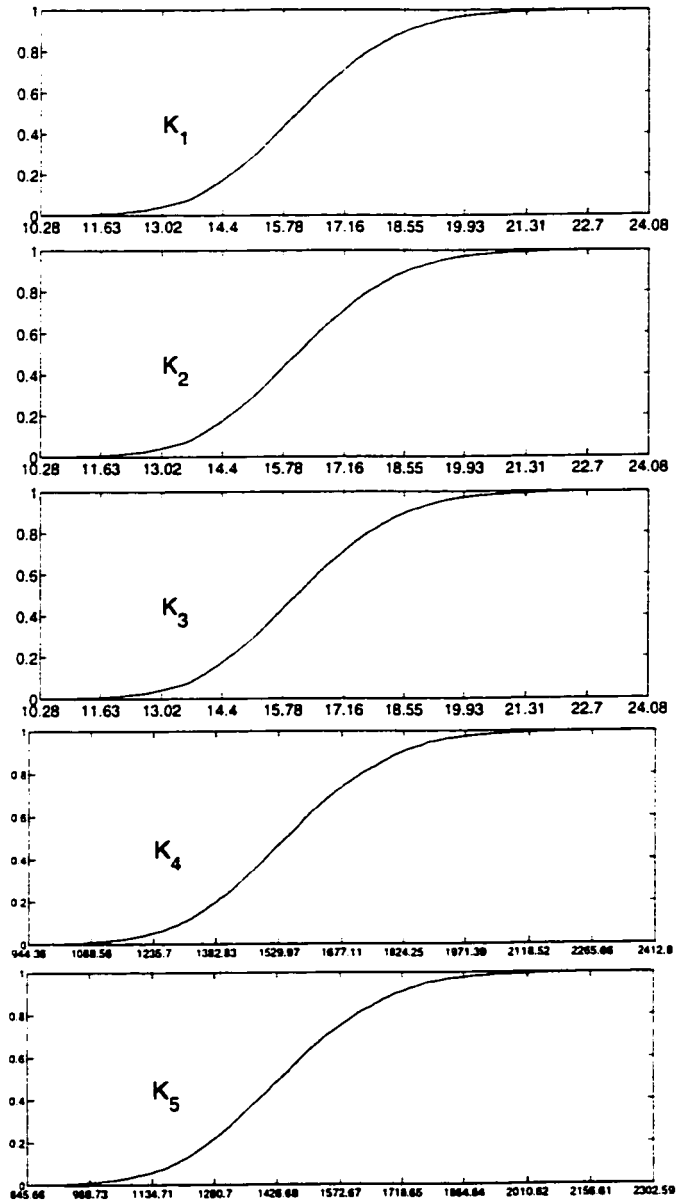


Figure 5.15: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 2 noise, with a box of dimension 12×12 added.
The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

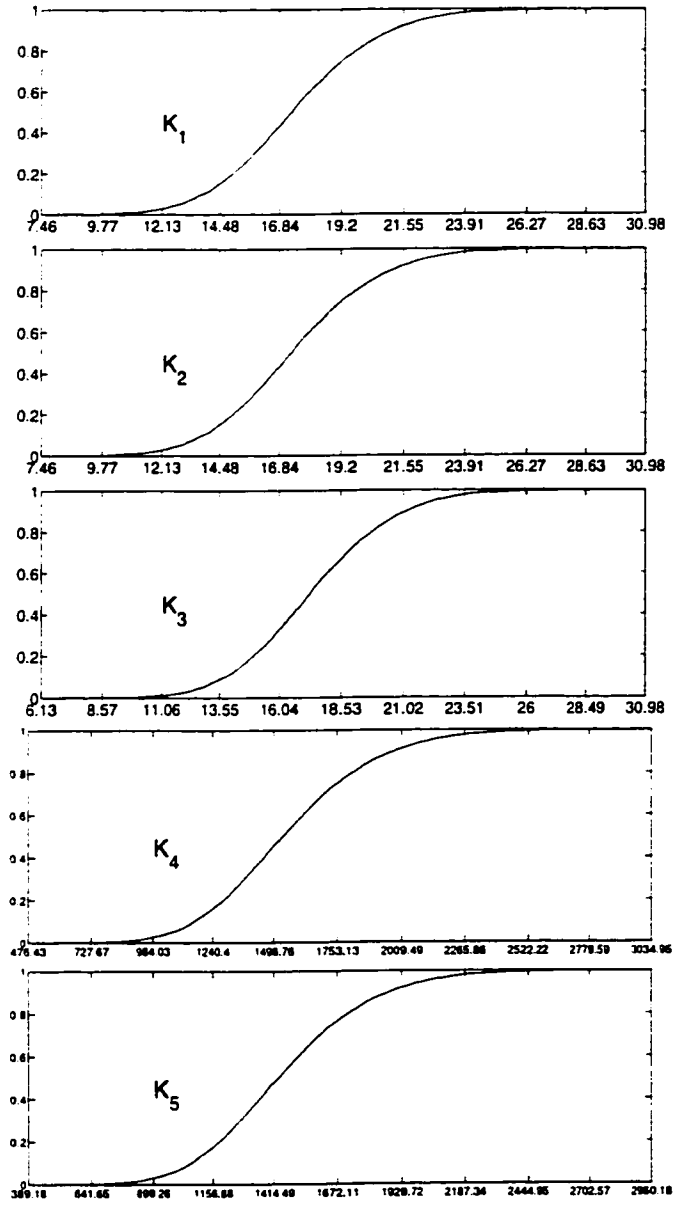


Figure 5.16: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 3 Noise, with a box of dimension 12×12 added.
The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

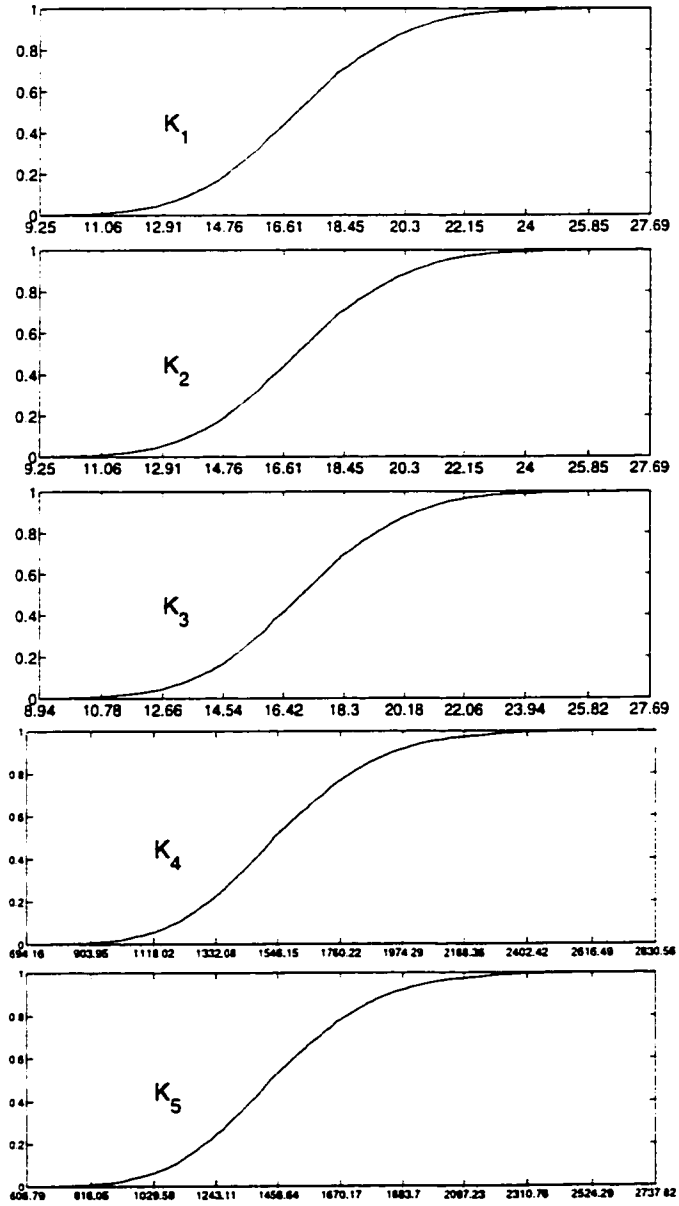


Figure 5.17: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 4 Noise, with a box of dimension 12×12 added.
The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

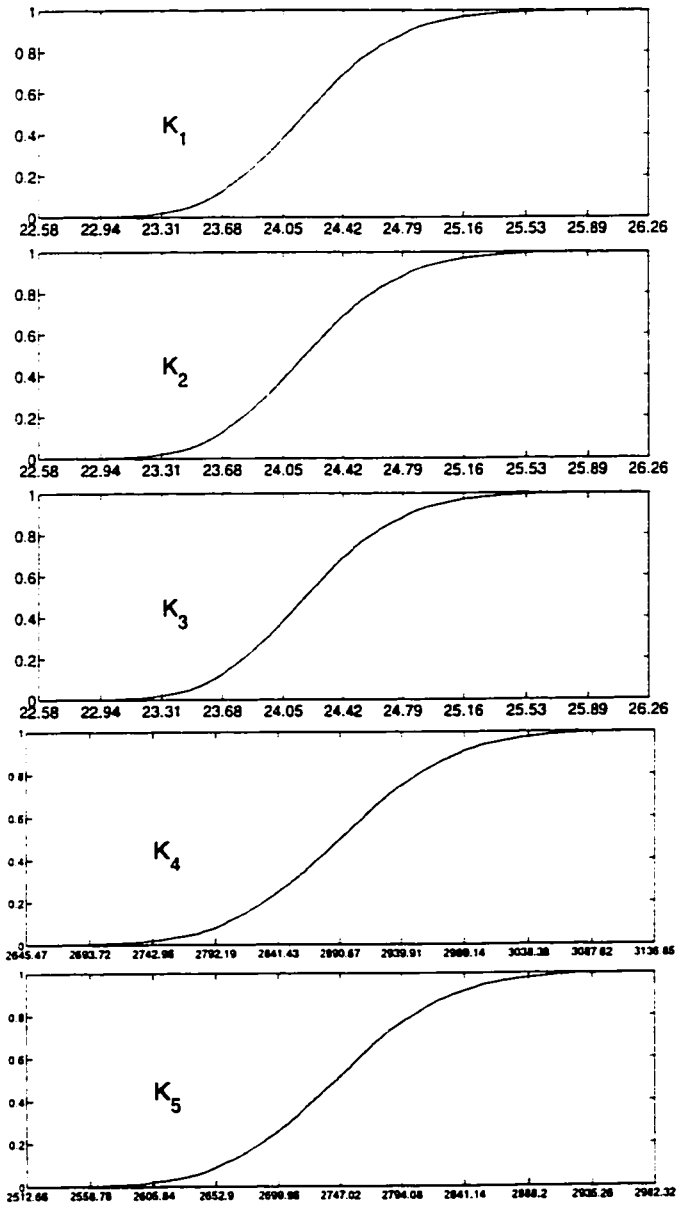


Figure 5.18: Cdfs of K_1, K_2, K_3, K_4, K_5 under Type 5 Noise, with a box of dimension 12×12 added.

The box was added to the middle of each macroblock, and there were 5000 samples collected in each simulation.

From the graphs, we can determine rough estimates of the power of the different statistics. These observations are summarized in Tables 1, 2, 3, 4 and 5.

The tables suggest that K_4 is the most powerful and robust test statistic. It achieves power of 80% or greater in every test case, and in the majority of cases has power approaching 100%. Thus, in the cusum procedure developed in Chapter 7, it will be K_4 that will be used for testing on I-frames. However, K_5 also performed quite well, and might be a practical choice when reducing computational time is a priority.

Error Type	Cut-off Value	Power under H_{1a}	Power under H_{1b}
1	5	100%	100%
2	11	25%	99%
3	16	5%	65%
4	15	5%	75%
5	12.5	100%	100%

Table 1: Estimates of the cut-off values for K_1 to achieve type I error $< 1\%$, and the corresponding power under H_{1a} = small box added, and H_{1b} = large box added.

Error Type	Cut-off Value	Power under H_{1a}	Power under H_{1b}
1	4.5	100%	100%
2	11	25%	98%
3	16	5%	65%
4	15	5%	80%
5	12.5	100%	100%

Table 2: Estimates of the cut-off values for K_2 to achieve type I error $< 1\%$, and the corresponding power under H_{1a} = small box added, and H_{1b} = large box added.

Error Type	Cut-off Value	Power under H_{1a}	Power under H_{1b}
1	4.5	100%	100%
2	11	25%	98%
3	16	5%	60%
4	15	5%	75%
5	12.5	100%	100%

Table 3: Estimates of the cut-off values for K_3 to achieve type I error $< 1\%$, and the corresponding power under H_{1a} = small box added, and H_{1b} = large box added.

Error Type	Cut-off Value	Power under H_{1a}	Power under H_{1b}
1	350	100%	100%
2	450	100%	100%
3	625	80%	99%
4	575	95%	100%
5	650	100%	100%

Table 4: Estimates of the cut-off values for K_4 to achieve type I error $< 1\%$, and the corresponding power under H_{1a} = small box added, and H_{1b} = large box added.

Error Type	Cut-off Value	Power under H_{1a}	Power under H_{1b}
1	120	100%	100%
2	400	99%	100%
3	625	80%	99%
4	550	75%	100%
5	550	100%	100%

Table 5: Estimates of the cut-off values for K_5 to achieve type I error $< 1\%$, and the corresponding power under H_{1a} = small box added, and H_{1b} = large box added.

Chapter 6

Hypothesis Testing on Motion Vectors

In Chapter 5, a hypothesis test was developed to test for the presence of an object in a macroblock that was encoded as part of an I-frame. In this chapter, the analogous test for macroblocks in P-frames will be defined. Section 6.1 explains the basic hypothesis test. Although the details of the test will only be worked out for motion vectors on P-frames, the same method could easily be extended to B-frame motion vectors. However, to minimize simulation complexity in this thesis, the application of the theory to B-frames will be omitted.

If the information contained in MPEG encoder produced motion vectors is to be useful, it will be necessary to analyse the properties of the motion vectors. A motion vector searching algorithm which is mathematically tractable is required; unfortunately, no implemented algorithm meets this criterion. So, in Section 6.2, a motion vector searching algorithm which is more amenable to mathematical manipulations is presented. As for the choice of error distribution, the goal is a workable approximation to reality.

Another casualty of simulation complexity is the flexibility to employ different I and P encoding patterns. The work in this thesis assumes that this pattern is I-P-I-P-I-P- . . . , because for the motion vector data generated, only the immediately previous frame was ever a reference frame. It would certainly be possible to extend this work

to other I/P patterns, but additional simulation would be required to determine the distribution of motion vectors which are calculated with reference to frames with a separation of more than one frame.[1]

Section 6.3 explains the results of the simulation of the distribution of motion vectors.

6.1 Hypothesis Testing on P-frames

Our approach is similar to that for I-frames. However, with P-frames, there is no question as to which test statistic to use: the only reasonable choice is the magnitude of the motion vector. This is justified, because in most practical situations, it is expected that there will be temporal dependence in the background noise in frames, which is modelled in this thesis by averaging the frames in time. With dependence between frames, the search for the closest matching macroblock should yield a nearby macroblock, and thus short motion vectors. This is illustrated in Figure 6.1, in which all motion vectors were found to be zero.

If an object is moving across the field of view with reasonable speed, certain motion vectors will track the movement of the object, and will thus be longer. This can be seen in Figure 6.2. There are two motion vectors that resulted from the motion of the box, the longer ones, and also two non-zero "noise" motion vectors, which are shorter. The remaining motion vectors are zero.

There is almost no information contained in motion vectors if the frames are independent in time, as illustrated in Figure 6.3. The motion vectors are almost never zero, and often the closest matching macroblock is quite distant. In this case, it will be more difficult to distinguish between motion vectors that arise from actual motion tracking of an object and those that do not. This is illustrated in Figure 6.4, in which a box has been added to the pictures, but the examination of individual motion vectors yields little information about the presence of the box because the background

[1] Furthermore, as will been seen, hypothesis testing using motion vectors is not nearly as powerful as using DCT coefficient statistics, and thus in Chapter 7 it will be advantageous to have a large I- to P-frame ratio.

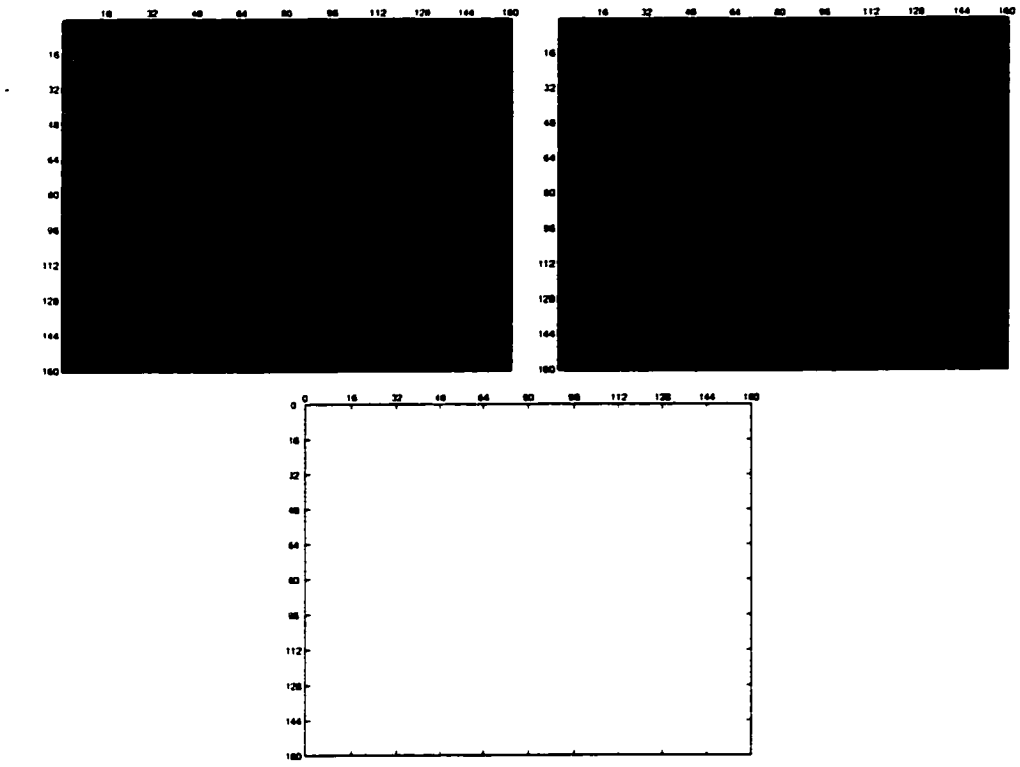


Figure 6.1: Two 7 × 7 block averaged frames and the corresponding motion vectors.

Top Left: Frame of normal(0,1) noise, averaged over 7 × 7 blocks, with 5 frames averaged back in time. Top Right: The next frame in the sequence. Bottom: Motion vectors for the second frame, with reference to the first. Here an empty macroblock indicates a zero motion vector, so all motion vectors are zero.

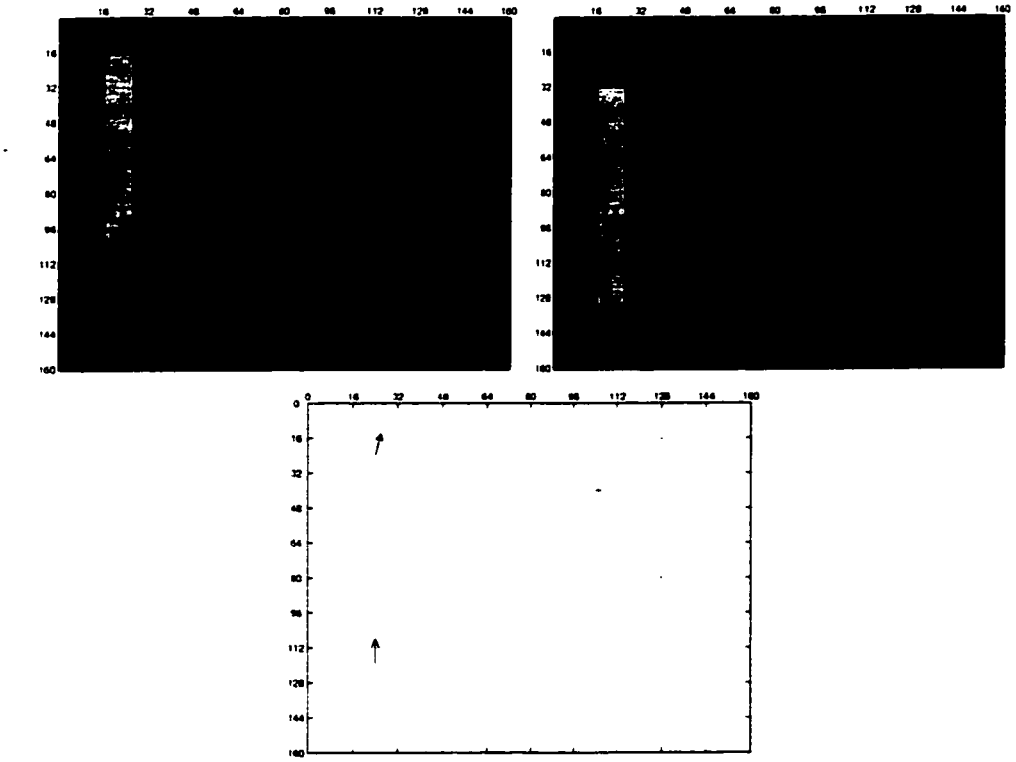


Figure 6.2: Two 7 × 7 block averaged frames and the corresponding motion vectors.

Top Left: Frame of normal(0,1) noise averaged over 7 × 7 blocks, with 5 frames averaged back in time, with an added object. Top Right: The next frame in the sequence. Bottom: Motion vectors for the second frame with reference to the first. Empty macroblocks indicate zero motion vectors. Note the two longer motion vectors in the second column of macroblocks that correspond to actual motion, as well as the other two very small motion vectors that are only tracking "motion" of the background noise.

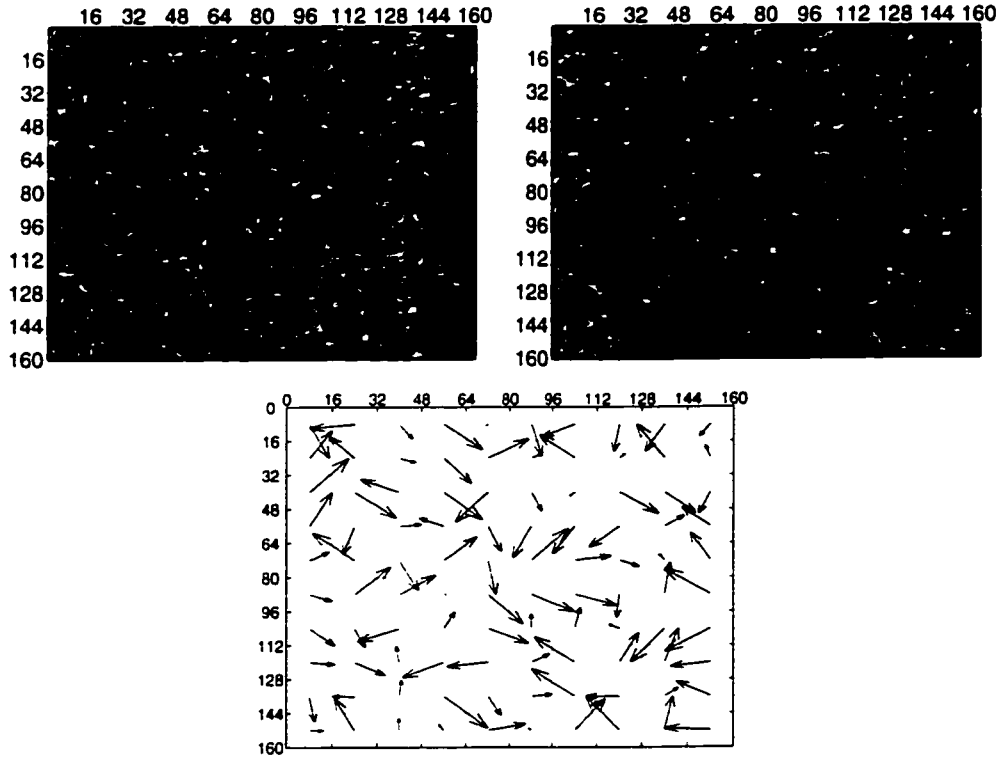


Figure 6.3: Two independent frames and the corresponding motion vectors.

Top: Frame of normal(0,1) noise averaged over 3 × 3 blocks, independent in time. Middle: Next frame in sequence. Bottom: Motion vectors for the second frame with reference to the first.

noise is also generating substantial motion vectors. Averaging background noise back in time and assigning penalties for longer motion vectors is necessary to create the correlation between frames necessary for short or zero length motion vectors to arise when coding the background noise.

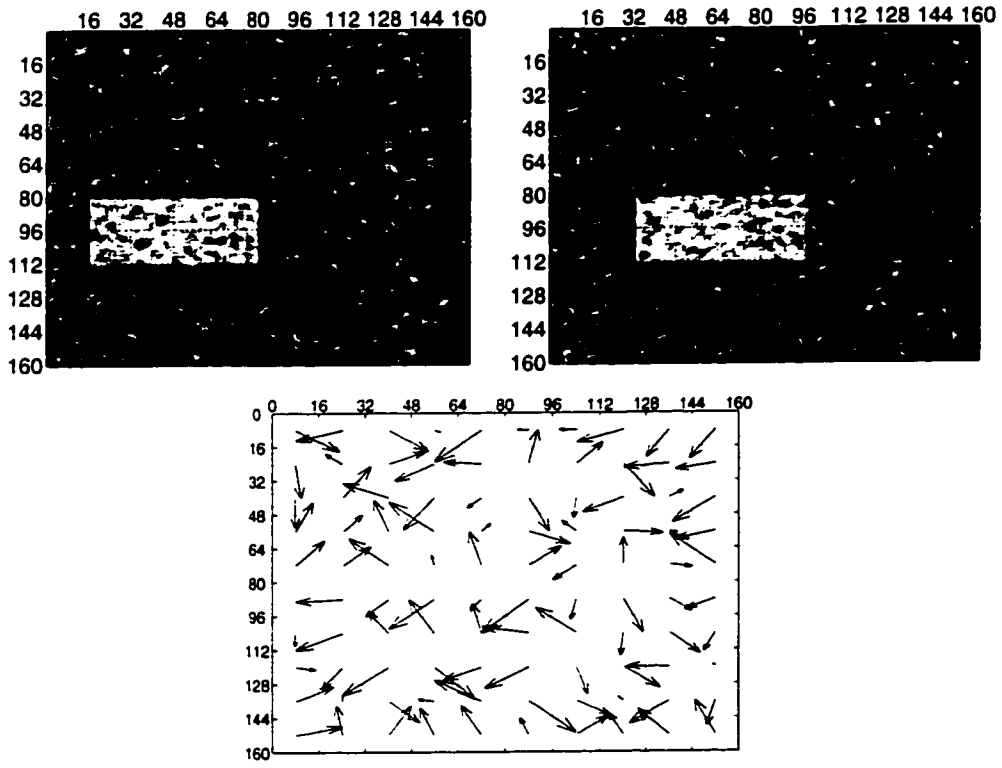


Figure 6.4: Two independent frames with a moving box added and the corresponding motion vectors.

Top: Frame of normal(0,1) noise averaged over 3×3 blocks, independent in time. Middle: Next frame in sequence. Bottom: Motion vectors for the second frame with reference to the first. Note the similarity of the motion vector distribution to that of Figure 6.3, despite the difference in original pictures.

6.1.1 Hypothesis Test for Macroblocks in P-frames

Each macroblock in a P-frame has an associated motion vector.[2] The hypothesis test is as follows:

H_o : The macroblock contains just noise.

H_1 : The macroblock contains a moving object.

Note that the alternate hypothesis is slightly different than that for I-frames; here we can only detect the presence of a moving object, while in an I-frame, any object, stationary or moving, will be detected.

Here the test statistic is

$$M = \sqrt{m_x^2 + m_y^2},$$

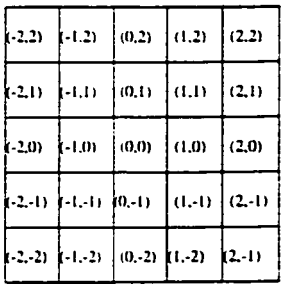
where m_x and m_y are the x and y components of the motion vector. Once again, we reject the null hypothesis if M is too large. The remaining sections of this chapter are concerned with finding the distribution of M, so that "too large" may be quantified.

6.2 Constrained Minimization Algorithm

Here we begin our search for a mathematically tractable motion vector searching algorithm. Due to the complexity of the calculations, we will first examine a simplified motion vector searching problem, and then apply this method to a more advanced situation in Section 6.2.1.

Let $X_{ij}, i, j \in \{-2, -1, 0, 1, 2\}$ represent the values of the process at time $t = -1$ and with coordinates (i, j) in a 5×5 grid with centre $(0, 0)$. Let $Y_{ij}, i, j \in \{-1, 0, 1\}$ represent the values of the process at time $t = 0$ with coordinates (i, j) in a 3×3 grid with centre $(0, 0)$. (X can be thought of as the first frame in a video, and Y as a macroblock from the next frame in the video.) See Figure 6.5.

[2] We will ignore any macroblocks that were encoded using I-frame techniques because no suitable motion vector was found.



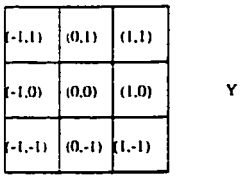


Figure 6.5: Representations of X_{ij} and Y_{ij} for the motion vector searching algorithm.

Consider the following equation in the nine unknowns $w_{-1,-1}, w_{-1,0}, w_{-1,1}$, $w_{0,-1}, \ldots, w_{1,1}$. It gives a measure of the error between X and Y for a given motion vector. Thus, we call it PenSS_E, for penalized sums of squares error.

$$\begin{aligned}\text{PenSS}_E &= \sum_{k=-1}^{1}\sum_{l=-1}^{1} w_{kl}\left(\sum_{i=-1}^{1}\sum_{j=-1}^{1}(Y_{ij} - X_{i+k,j+l})^2\right) \\ &\quad +\lambda_1 \sum_{k=-1}^{1}\sum_{l=-1}^{1} w_{kl}(1 - w_{kl}) + \lambda_2\left(1 - \sum_{k=-1}^{1}\sum_{l=-1}^{1} w_{kl}\right)^2. \qquad (6.1)\end{aligned}$$

The first double sum in PenSS_E is designed to find the closest matching "macroblock", the λ_1 product should keep the w_{kl}'s near 0 or 1, and the λ_2 product should keep the sum of the the w_{kl}'s near 1.

Now, minimize PenSS_E with respect to the w_{kl}. The motion vector is then determined by finding the largest w_{kl}, which implies that the motion vector should be (k, l).

To do the minimization, take partial derivatives with respect to w_{kl}.

$$\begin{aligned}\frac{\partial \text{PenSS}_E}{\partial w_{kl}} &= BF_{kl} + \lambda_1(1-2w_{kl}) + \lambda_2\Big((-1)(1-\sum\sum w_{ij}) + (1-\sum\sum w_{ij})(-1)\Big) \\ &= 2(\lambda_2-\lambda_1)w_{kl} + 2\lambda_2\sum_{i\neq k}\sum_{j\neq l} w_{ij} + BF_{kl} + \lambda_1 - 2\lambda_2\end{aligned}$$

where $BF_{kl} = \sum_{i=-1}^{1}\sum_{j=-1}^{1}(Y_{ij} - X_{i+k,j+l})^2$, (the "best fit" part).

So we have the following system of linear equations in nine unknowns:

$$\begin{bmatrix}
\Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{-1,-1}+\lambda_1-2\lambda_2 \\
2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{-1,0}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{-1,1}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{0,-1}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{0,0}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & BF_{0,1}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & 2\lambda_2 & BF_{1,-1}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & 2\lambda_2 & BF_{1,0}+\lambda_1-2\lambda_2 \\
2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & 2\lambda_2 & \Lambda & BF_{1,1}+\lambda_1-2\lambda_2
\end{bmatrix} \tag{6.2}$$

where $\Lambda = 2(\lambda_2 - \lambda_1)$.

The inverse of the non-augmented matrix is:

$$\frac{-1}{2\lambda_1(\lambda_1-9\lambda_2)}\begin{bmatrix}
\Upsilon & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \Upsilon & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \Upsilon & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \Upsilon & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \Upsilon & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \Upsilon & \lambda_2 & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \Upsilon & \lambda_2 & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \Upsilon & \lambda_2 \\
\lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \lambda_2 & \Upsilon
\end{bmatrix} \tag{6.3}$$

where $\Upsilon = \lambda_1 - 8\lambda_2$.

This yields,

$$w_{kl} = \Psi\left(\Upsilon\,(BF_{kl} + \lambda_1 - 2\lambda_2) + \lambda_2\sum_{i\neq k, j\neq l} BF_{ij}\right). \tag{6.4}$$

So each w_{kl} has a weighted chi-squared distribution.

What we really want to know is the probability that a given w_{kl} is the largest. It seems highly unlikely that it will be possible to determine this analytically. So simulation will be required.

6.2.1 Applying PenSS$_E$ Method to 16 × 16 Macroblocks

It is possible to apply the motion vector searching algorithm of Section 6.2 to larger systems; however, the matrices involved become very large. Let n be the number of possible motion vectors in the system. Then the set of linear equations that must be solved are essentially of the same form as above; the system is now represented by an $n \times n$ matrix, call it A, with $2(\lambda_2 - \lambda_1)$ along the main diagonal and $2\lambda_2$ everywhere else. This matrix has as its inverse the matrix A^{-1} with

$$\frac{-\lambda_1 + (n-1)\lambda_2}{2\lambda_1(\lambda_1 - n\lambda_2)}$$

along the main diagonal and

$$\frac{-\lambda_2}{\lambda_1(\lambda_1 - n\lambda_2)}$$

everywhere else.

The proof that this matrix is in fact the required inverse can be done by direct verification. The matrix multiplication for entries along the main diagonal of AA^{-1} boils down to:

$$(2(\lambda_2 - \lambda_1)(\lambda_1 - (n-1)\lambda_2) + 2\lambda_2^2(n-1))/ - 2\lambda_1(\lambda_1 - n\lambda_2)) = 1.$$

Off the main diagonal, the calculation is:

$$2(\lambda_2 - \lambda_1)\lambda_2 + 2\lambda_2(\lambda_1 - (n-1)\lambda_2) + 2\lambda_2^2(n-2) = 0.$$

Thus, multiplication of the original matrix and the candidate for its inverse results in the identity matrix, confirming that the inverse is correct.

6.3 Motion Vector Distribution by Simulation

Although Section 6.2.1 develops a tractable motion vector searching algorithm, and the distribution of motion vectors is thus known in theory; in practice, the known weighted chi-squared distribution is still too complicated to be useful. Simulation was necessary to obtain tables on which approximate calculations may be based.

The motion vector searching algorithm that was implemented in the following simulations must be explained as the distribution of motion vectors will depend of the precise method chosen. An exhaustive search for the closest matching motion vector was performed within a square search window of "radius"[3] 16 of the current macroblock.

The other significant characteristic of the motion vector searching algorithm is the precise definition of "closest matching." The norm between two macroblocks was defined to be the sum of the difference of squares, with a modification. Suppose $\{a_{ij}\}$ is macroblock one, and $\{b_{ij}\}$ is macroblock two, and the motion vector between a and b is given by (m_x, m_y). Then,

$$\text{Norm}(a,b) = \sum_{i=1}^{16}\sum_{j=1}^{16}(a_{ij} - b_{ij})^2 + \lambda(m_x^2 + m_y^2).$$

The length of the motion vector between two macroblocks influences the value of the norm; the farther apart two macroblocks, the larger the norm, no matter how good a fit at that relatively distant location. In practice, this penalty for longer motion vectors effectively prevents motion vectors longer than about 12 from ever being selected, under any circumstances, which justifies the search window "radius" of 16.

It was necessary to include the length of the motion vector as part of the norm to help favour the selection of short motion vectors to mimic what is generally expected in practice. For example, when three frames were averaged together in time, with an averaging block size of 3×7 and $\lambda = 0$ then 93% of the motion vectors were zero.

[3]This means that shifts up vertically of maximum height 16 pixels, shifts down vertically of maximum depth 16 pixels, and analogous shifts in the horizontal direction were permitted. Near edges, the search could clearly not extend beyond the boundaries of the frame, and was truncated.

However, the remaining motion vectors were almost uniformly distributed over the other possibilities. Furthermore, since more of these remaining possible motion vector lengths are longer, the actual motion vectors tend to be biased towards longer vectors. With $\lambda = 1$, nearly 98% of motion vectors were zero. Having motion vectors with this type of distribution will significantly improve the power of our hypothesis test, because there is a much stronger link between non-zero motion vectors and actual object motion.

6.3.1 Simulations to Determine Motion Vector Lengths

Tables describing the distribution of motion vectors under various hypothesized models are required in Chapter 7. Below are several graphs, which show the probability of observing motion vectors with length less than or equal to a given value for different models. It is important to note, that although simulations have been performed for different variations on the general model, in practice, a careful study of the characteristics of the particular situation to be modelled would need to be studied, such that the parameters in our model (i.e. λ or the search radius) may be selected to most closely match actual observed properties. Also, because we only consider shifts in whole pixel units within a specified range, there are only a finite number of possible motion vectors lengths.

Figure 6.6 gives the cdfs of motion vector lengths for video with 3×3 averaged spatial noise which is averaged back in time 3 frames. The corresponding functions for 5×5 and 3×7 block spatial averaging may be found in Figures 6.7, 6.8 and 6.9.

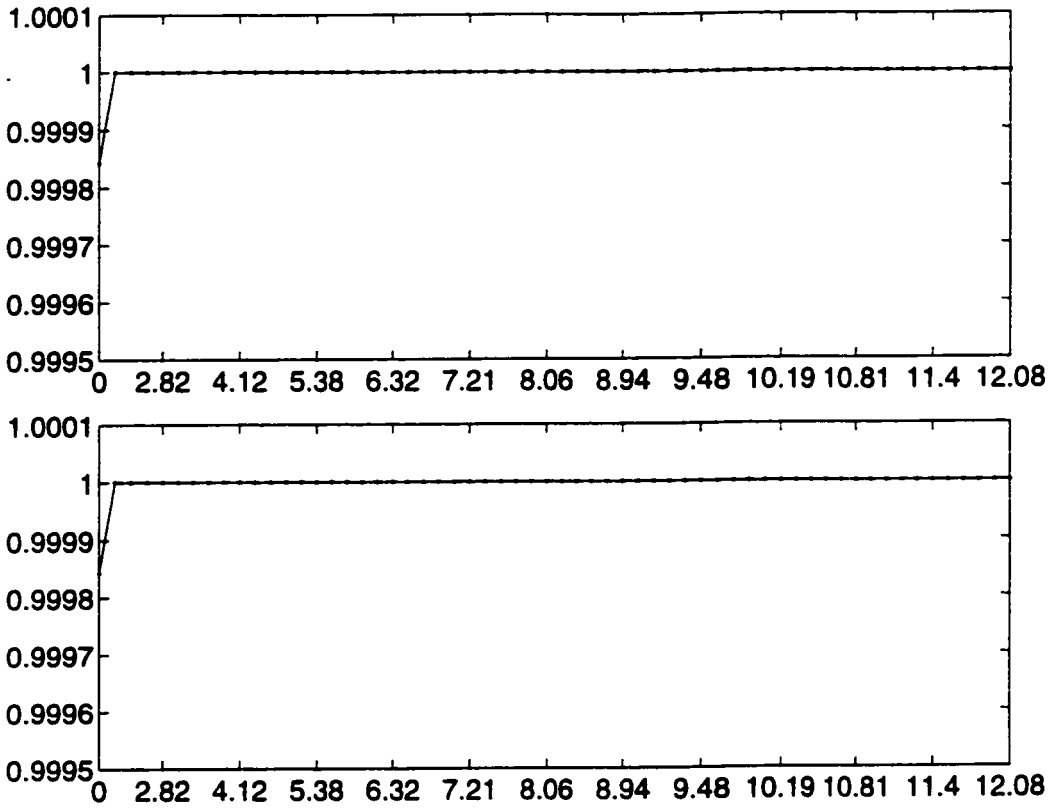


Figure 6.6: Cdfs of motion vector lengths.
This is for a process that has 3×3 spatial averaging and is averaged back in time 3. The upper graph is with $\lambda = 1$, and the lower with $\lambda = 2$.

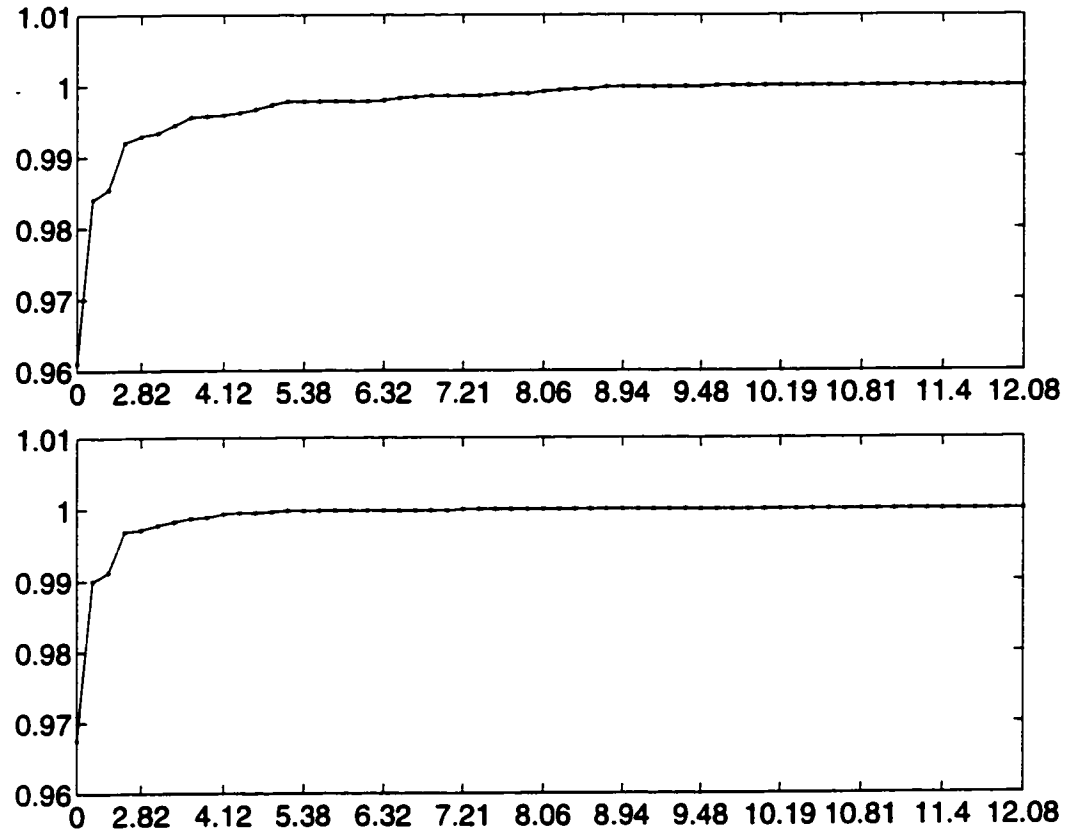


Figure 6.7: Cdfs of motion vector lengths.
This is for a process that has 5×5 spatial averaging and is averaged back in time 3.
The upper graph is with $\lambda = 1$, and the lower with $\lambda = 2$.

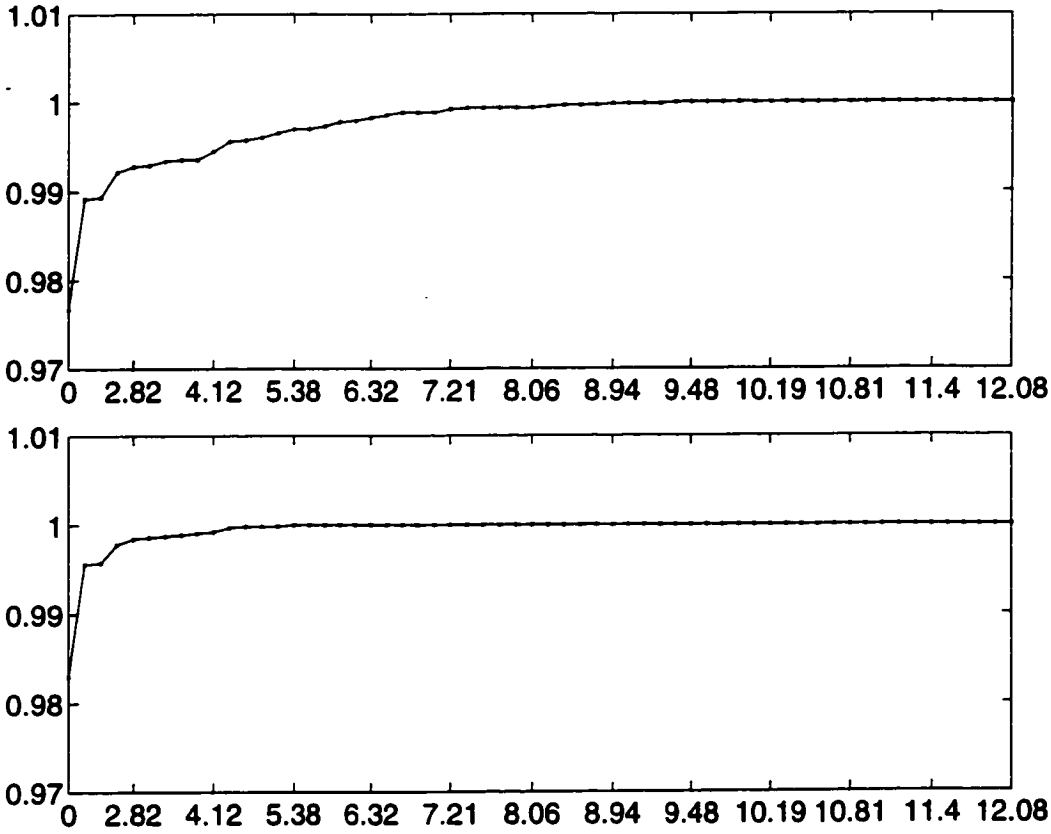


Figure 6.8: Cdfs of motion vector lengths.
This is for a process that has 3×7 spatial averaging and is averaged back in time 3.
The upper graph is with $\lambda = 1$, and the lower with $\lambda = 2$.

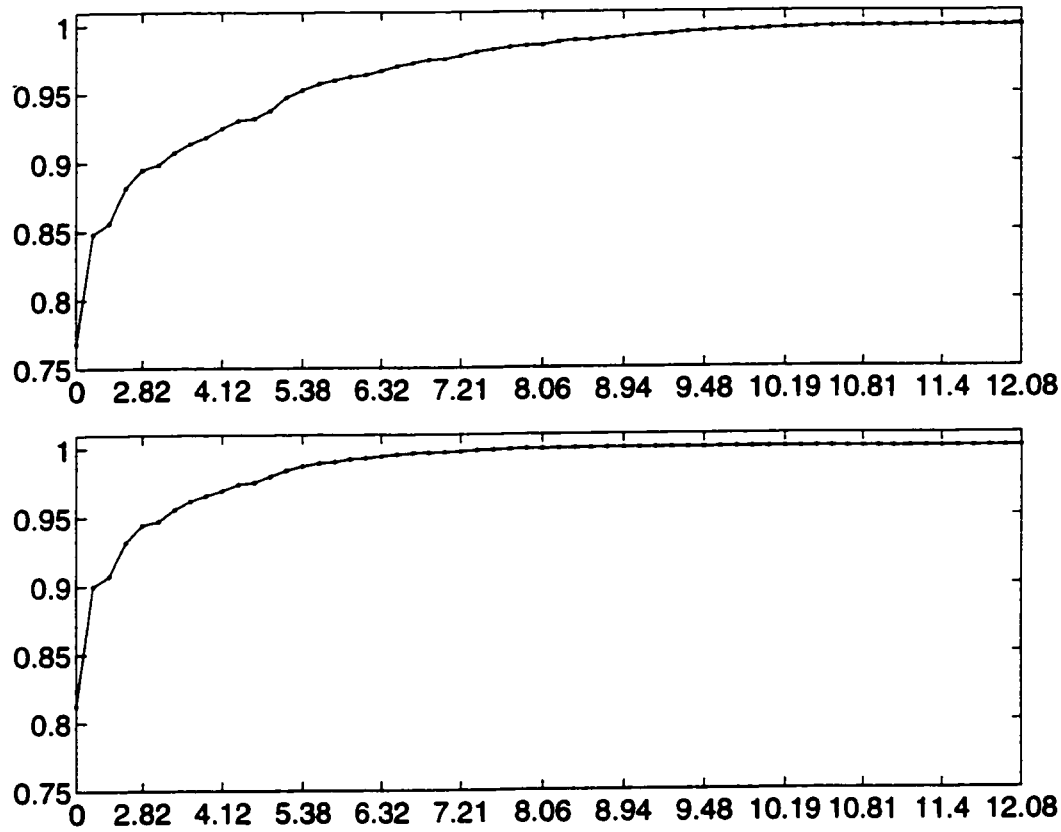


Figure 6.9: Cdfs of motion vector lengths.
This is for a process that has 3×7 spatial averaging and is averaged back in time 2.
The upper graph is with $\lambda = 1$, and the lower with $\lambda = 2$.

Chapter 7

Cumulative Sum Procedures

7.1 Introduction

Control charts have been used in manufacturing quality control processes for decades to detect changes in products. One of the first control charts was the Shewart chart, developed in the early 1930's, which simply plots the means of samples of normally distributed data taken at regular intervals against time. If the plot ever falls outside of control lines (usually drawn three standard deviations above and below the on-target mean value), the process is declared out-of-control. This chart is simple to implement, and quickly detects large deviations in the mean, but smaller or gradual changes may go unnoticed.

More complicated rules for signaling an out-of-control process were soon developed, such as sounding the alarm if three of the last five observations were above a somewhat lower control line. In 1954, Page [16] proposed a test with a simpler stopping rule, that in fact took into consideration all past data when determining if the process was in control. This system is now known as a cumulative sum chart, or cusum chart.

Page's original cusum was described very generally. One was instructed to take samples at regular intervals, and to assign a score to each sample, such that the mean score for samples which are known to be acceptable is negative, and the mean value of out-of-control samples is positive. In mathematical terms, let the score for the kth

sample be x_k. Define

$$\begin{aligned} S_0 &= 0 \\ S_n &= \max\{S_{n-1} + x_n, 0\}, \quad n \geq 1. \end{aligned} \tag{7.1}$$

Take action whenever

$$S_n \geq h, \tag{7.2}$$

where h is a predetermined constant.

Now even when a process is in control, with probability 1, this type of cusum procedure will eventually signal an alarm. The average length of time until a so-called false alarm is signaled is the average run length or ARL. Clearly, a cusum procedure which detects the change in the sampling distribution as quickly as possible, while maintaining a long on-target ARL, is the most desirable. Lorden [9] developed a minimax criterion to determine an optimal stopping procedure; in this sense an optimal cusum is that which minimizes the maximum length of time until the change is detected.

Technically, let the observed independent random variables be $X_1, X_2, \ldots, X_{m-1}, X_m, \ldots$. Suppose X_i has distribution function F before the change at time m and distribution $G \neq F$ after time m. Let N be a stopping time[1] where the outcome $N = \tau$ means that the alarm was sounded at time τ. Define

$$D(N) = \sup_{m \geq 1} \text{ess sup} E_m\{[N - m + 1]^+ \| X_1, \ldots, X_{m-1}\}. \tag{7.3}$$

A stopping rule N is then called optimal if it minimizes $D(N)$ over the class of all stopping rules with the same average run length.

Moustakides [14] was able to generalize the work of Lorden and proved that Page's stopping time is optimal in the sense of (7.3). Working from Page's 1954 observation that a cusum procedure can essentially be viewed as a Wald sequential test (i.e. a Sequential Probability Ratio Test (SPRT)), Moustakides and Lorden used a procedure which is essentially equivalent to the following realization of Page's stopping time.

[1] A *stopping time* or *stopping rule* is an integer valued random variable for the sequence $X_1, X_2, \ldots$ if for all $n = 1, 2, \ldots$ the event $\{N = n\}$ is independent of X_{n+1}, X_{n+2}. See [20] for more details.

Keeping the notation from the above paragraph and letting f and g denote the (Lebesgue) densities of F and G respectively, let

$$\begin{aligned} S_0 &= 0 \\ S_n &= \max\left\{S_{n-1} + \log\left(\frac{g(X_n)}{f(X_n)}\right), 0\right\} \end{aligned} \tag{7.4}$$

and stop at the first n such that $S_n \geq h$, where h is a predetermined constant,[2] usually referred to as the signal level. Essentially, an optimal method for determining Page's x_n value has been found.

7.2 Average Run Lengths

The average run length (ARL) is a characteristic of a cusum process which is very desirable to know. Although it must be remembered that the run length of a particular instance of a process is itself a random variable, knowing the ARL can assist in determining whether a signaled alarm was in fact due to a change in the distribution. Perhaps more importantly, the ARL is a way of evaluating the type I error in the cusum process, as it provides a measure of the frequency of false alarms.

Page tackled the question of determining the ARL of a cusum procedure in his original paper [16]. His approach was to observe that a cusum procedure defined as in (7.1) and (7.2) is equivalent to a SPRT on $(0, h)$ where the test simply continues if there is acceptance on the lower boundary; that is, the test continues until there is rejection of the null hypothesis.

Let the probability that a test ends on the lower boundary be q. Suppose the average sample number (ASN) unconditional on the outcome, that is, the number of samples expected on average before there is acceptance or rejection, is η, the ASN conditional on ending on the lower boundary is η_l and the ASN condition on ending on the upper boundary is η_u.

Define "success" as the event of the SPRT ending on the upper boundary, and let X be the random variable that counts the number of failures until success. The

[2] In fact, this is the logarithm of the cusum analysed by Moustakides and Lorden, but this form is preferred in this thesis because it is additive, rather than multiplicative. See equations (7.5) and (7.6) for the definition without the logarithm.

probability that $X = r$ is $q^r(1-q)$. Thus, $E[X] = \sum_{r=0}^{\infty} rq(1-q) = q/(1-q)$. Denote the ARL of the cusum by L. Then.

$$\begin{aligned} L &= \frac{q}{1-q}\eta_l + \eta_u \\ &= \frac{q\eta_l + (1-q)\eta_u}{1-q} \\ &= \frac{\eta}{1-q}. \end{aligned}$$

However, it is in general difficult to find the values of q and η.

Moustakides studied the cusum procedure in a form for which it is sometimes possible to easily determine the ARL. The exact procedure is defined by the following. Let

$$\begin{aligned} S_0 &= 0 \\ S_n &= \max\{S_{n-1}, 1\}\frac{g(X_n)}{f(X_n)} \end{aligned} \tag{7.5}$$

and let the stopping time N be given by

$$N = \inf\{n \geq 1 : S_n \geq h\} \tag{7.6}$$

where h is again a predetermined non-negative constant, and the infimum of the empty set is infinity. One case in which it is not difficult to relate the ARL and h is when $h < 1$. Then $N = \inf\{n \geq 1 : g(X_n)/f(X_n) \geq h\}$, for if there had been no alarms by time n, we know that $S_k \leq 1$ for $k < n$. To determine the ARL we consider that null hypothesis case in which there is no actual change in distribution. At each new point in time, we either have the "success" of a (false) alarm sounding or not. This success depends only on the most recent observation; all that is tested is whether $g(X_n)/f(X_n) \geq h$. This is a geometric random variable, so

$$\text{ARL} = \frac{1}{P[g(X_n)/f(X_n) \geq h]}.$$

For specific realizations of f and g, calculating L given h may be a tractable calculation. Unfortunately, for many applications, the setting $h < 1$ is not practical.

7.3 Applying a Cusum to MPEG Data

In Chapters 5 and 6 macroblock based hypothesis tests for detecting the presence or motion of objects in I- or P-frames were developed. To tie these tests together into one procedure, we will consider a cusum process on the p-values of the hypothesis tests.

Analysing MPEG data is challenging, because the type of data is constantly changing, depending on whether the type of frame currently being analysed is an I-picture, P-picture or B-picture. A sequence of hypothesis tests is one method for determining if it is reasonable to conclude that the video data contains something other than just noise. Multiple rejections of the null hypothesis in the sequence of hypothesis tests suggest that the overall hypothesis of the video containing only noise should also be rejected.

Recall the definition of the p-value of a hypothesis test.

Definition 7.1 (p-value) *Let the test statistic for the hypothesis test be K, and suppose $K = a$ has been observed. The* p-value *for this observed test statistic is the smallest value of the type I error* [3] *that would lead to the rejection of H_o. Thus, in the current applications in which we are rejecting the null hypothesis if the observed test statistic is too large, we have*

$$p\text{-}value = P[K' \geq a],$$

where K' is a random variable with the same distribution as K.

Thus, an alternate method of performing a hypothesis test is to reject H_o if the observed p-value is smaller than the desired type I error.

Considering the p-values for each test is a method of comparing results from a diverse set of hypothesis tests. Under the null hypothesis the p-values should be uniformly distributed on [0,1], independent of the type of test performed. If the video contains a moving object, the distribution of the p-values will change, and should shift so that small p-values (which lead to rejection of H_o) are more likely. So, if the

[3] Recall that type I error is the probability of falsely rejecting H_o.

point at which the distribution of the p-values changes can be detected, the time in the video in which an object appears will be known.

7.3.1 A Cusum of p-Values

Motivated by observations by McDonald [11], the cusum procedure of p-values will be constructed as follows. Let X_i be 1 minus the ith p-value.[4] Test

H_o : X_i are independent and have density function $f(x) = 1$ on $[0, 1]$

H_1 : X_i are independent and have density function $g(x) = \frac{\delta e^{\delta x}}{e^{\delta} - 1}$, $\delta > 0$, on $[0, 1]$.

Before applying Lorden's cusum procedure (7.4), note that

$$\begin{aligned} \log\left(\frac{g(X_i)}{f(X_i)}\right) &= \log\left(\frac{\delta e^{\delta X_i}}{e^{\delta} - 1}\right) \\ &= \log\left(\frac{\delta}{e^{\delta} - 1}\right) + \delta X_i \end{aligned}$$

which is linear in X_i. Dividing through by δ, then yields a cusum statistic of

$$S_n = \max\left\{S_{n-1} + X_n + \frac{1}{\delta}\log\left(\frac{\delta}{e^{\delta} - 1}\right), 0\right\}.$$

Lorden's cusum, under these conditions, is therefore identical to the cusum developed by McDonald in [11]. Thus, numerically determined results in [11] may be use to determine an appropriate h for a given δ and desired on-target ARL, as given in Table 6.

Unfortunately, it is not easy to redo the numerical calculations done by McDonald to determine the appropriate h for a given ARL and δ. However, extrapolating or interpolating from a plot of ARL versus h for a fixed δ would provide a reasonable guess for an appropriate h for a desired ARL. For $\delta = 12.6407$ see Figure 7.1. Figure 7.2 provides the plot for $\delta = 3.8022$.

Simulations verified that the McDonald cusum, with parameters $\delta = 12.6407$ and $h = 0.456$ does in fact have an ARL of about 1000. When 10000 of this cusum under

[4] We need to subtract the p-value from one because we want to test against a shift in the p-values towards 0. Without this transformation, we would be testing for a shift towards 1

ARL	δ	h
1000	12.6407	0.456
1000	3.8022	1.382
1000	0.7348	2.507
1000	0.3430	5.710
500	12.6407	0.395
500	3.8022	1.2031
500	0.7348	2.125
500	0.3430	4.455

Table 6: Appropriate signal values h for specified ARL and δ. (Copied from Table 7 in McDonald [11].)

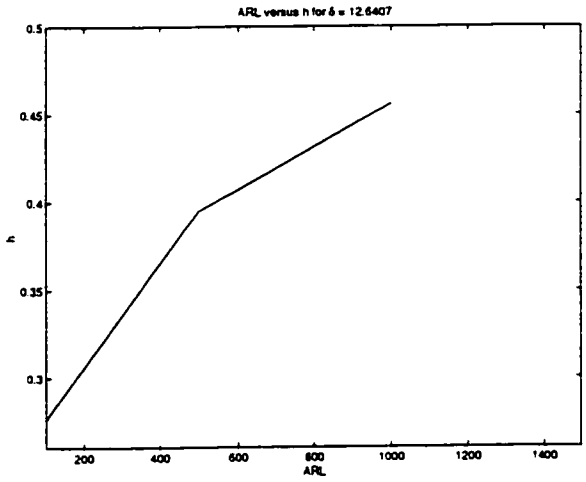


Figure 7.1: Plot of ARL versus h for $\delta = 12.6407$.

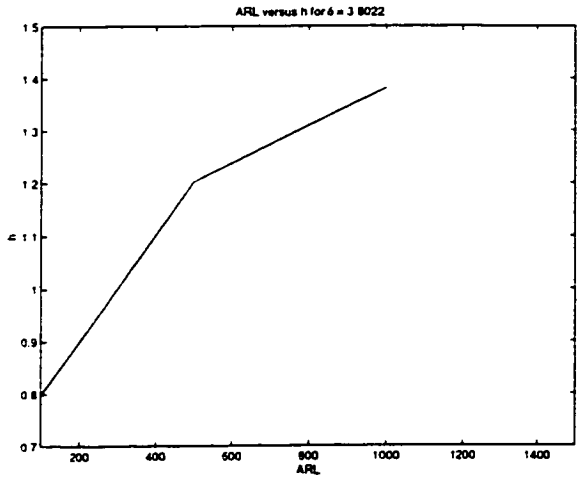


Figure 7.2: Plot of ARL versus h for $\delta = 3.8022$.

null hypothesis conditions of uniform[0,1] input were performed, the calculated ARL was 1010.3, with standard deviation 1005.8. A histogram of the run lengths is in Figure 7.3. It appears that the run lengths have an exponential distribution.

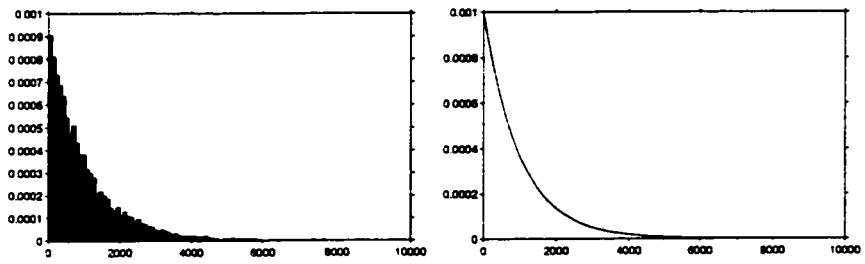


Figure 7.3: Density histogram of run lengths, plot of $y = 0.001e^{-0.001x}$. This is for the case outlined in McDonald's paper: independent, uniform[0,1] samples.

7.4 Simulation Results

To determine the effectiveness in practice of the proposed scheme, computer simulation of the entire process was performed. This simulation consisted of three main steps:

1. Producing the actual video data, and the corresponding test statistics, which, of course, depended on whether the frame was encoded as an I- or P-frame
2. Calculating the p-values for the corresponding hypothesis test from this "MPEG data"
3. Running the cusum procedure on the output p-values.

The C++ code that performs these simulations can be found in Appendix C.

In general, some strengths, but also problems were observed. Detecting the presence of an object in an I-frame is generally quite effective. The P-frame data is much more problematic. In the cusum procedure, it has been assumed that under the null hypothesis the input data is uniformly distributed on $[0, 1]$. However, the motion vector length data is actually from a discrete distribution, and moreover, most of the motion vector lengths are 0, with only a handful of other lengths that occur. So the p-values also have a discrete distribution, with a very limited support. Any motion vector length of 0 gives a p-value of 1. Because non-zero motion vector lengths are relatively improbable, they give small p-values, but not usually small enough so that one such input will sound the alarm in the cusum. However, there are often only a few motion vectors generated by a moving object, and in general they will not be adjacent. Thus, even when required and desired, it is difficult to sound the alarm using the motion vector data. To a certain extent, this inability to detect is affected by the order in which the statistics are scanned from the two-dimensional frame to the one-dimensional list to be processed. In the simulations actually performed (see the code in Appendix C), this scanning is from left to right, top to bottom. Vertical motion of wide objects, which should produce a horizontal row of adjacent motion vectors, would be thus more easily detected than motion in the horizontal direction, which would be expected to produce motion vectors which are vertically adjacent.

Figures 7.18 and 7.19 demonstrate just how much the power of the cusum is affected by the type of motion and position of the block attempting to be detected. In Figure 7.19 the cusum is much less effective than in Figure 7.18. This is the result of the general problems of "splitting" the added block over many macroblocks and having more macroblocks of just pure noise (and thus non-significant statistics) between hits. For example, consider Figure 7.4. If a macroblock contains part of the inserted block, call this a hit, H, and if the entire macroblock is covered by the block, let it be a full hit, fH. Macroblocks that do not contain any of the block are misses, M. Recall that the statistics for each frame are scanned in from left to right, top to bottom. For the first frame, the statistics are scanned in as:

$$12M, fH, fH, M, M, M, fH, fH, 16M.$$

When motion of the inserted block is down, the statistics are scanned is as:

$$12M, H, H, M, M, M, M, fH, fH, M, M, M, M, H, H, 10M.$$

For horizontal motion, we get:

$$12M, H, fH, H, M, M, M, H, fH, H, 15M.$$

Full hits contribute the most to the value of the cusum, hits contribute somewhat (depending on how much and which part of the macroblock is occupied) and misses may increase or decrease the value of the cusum. Several successive increments increase the likelihood of the alarm sounding, while longer periods of misses allow for the possibility that a cusum that is near the alarm level will actually be lowered.

To see how the cusum performs under various conditions, many test cases were run. This results can be seen in Figures 7.5 –7.19.

Because the hypothesis test on I-frames testings for the *presence* of an object, while on P-frames it is the *motion* of an object is detected, running the cusum procedure on the p-values from these two types of frames separately could be used to detect only the precense *or* motion of an object. The results of such as test are in Figure 7.20. Unfortunately, analysing the P-frames separately highlights the ineffectiveness of the cusum on that type of data.

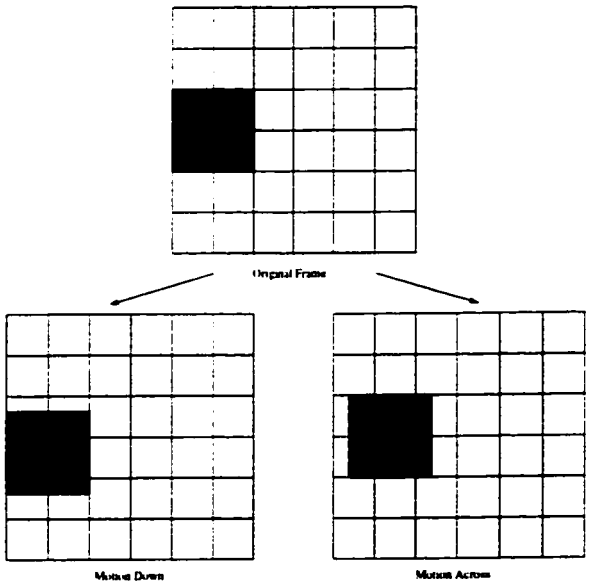


Figure 7.4: Illustration to show how different types of motion affect the power of the cusum.

With horizontal scanning, when motion is down, there are fewer "hits" between longer gaps of "misses." When motion is across, there are more adjacent "hits" and fewer "misses" until the next hit.

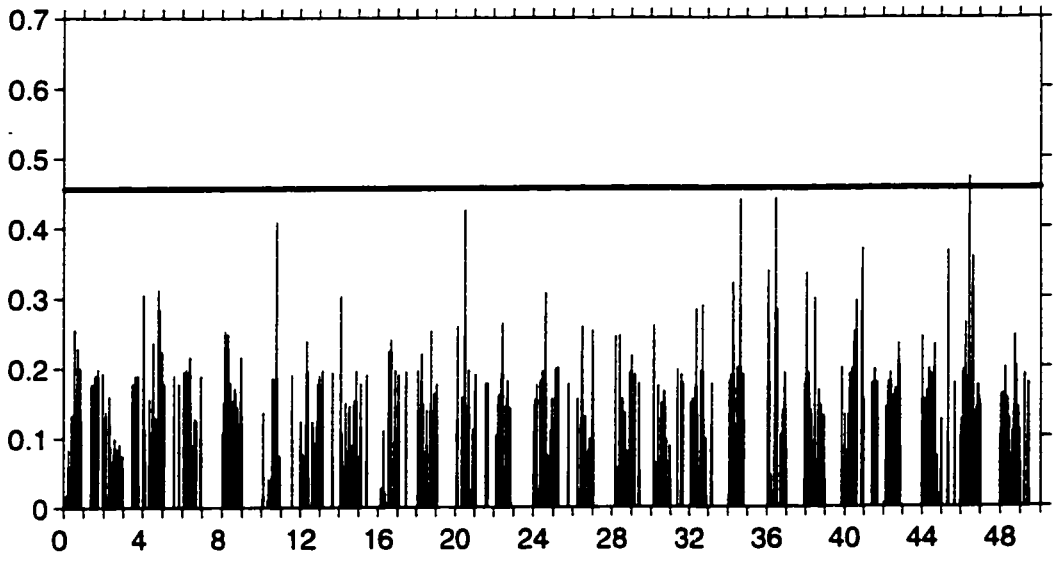
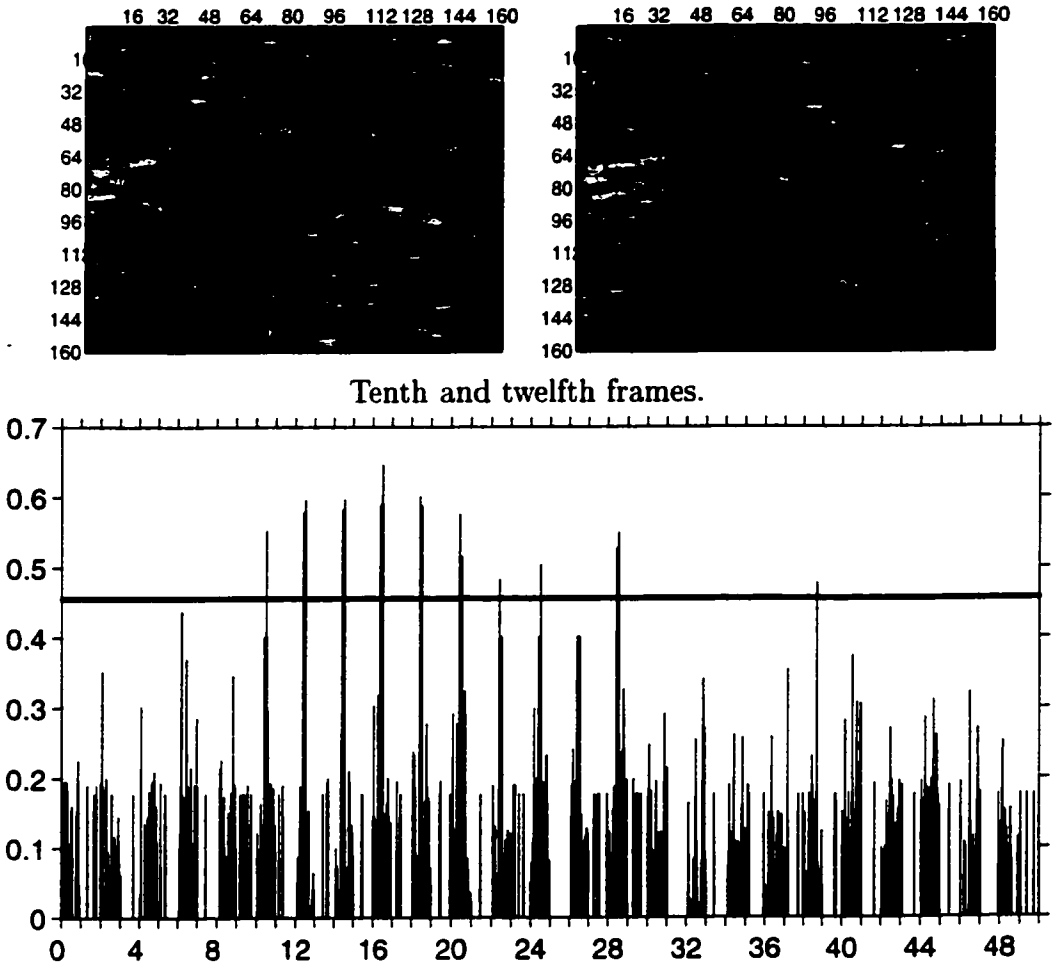


Figure 7.5: Case 0 cusum

Spatial averaging:	3 × 7
Time averaging:	Back 3 frames
Size of added box:	No box added
Intensity of added box:	n/a
Box added in frames:	n/a
Direction of box motion:	n/a
Motion step size:	n/a
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	n/a
False alarm in frame:	46



Tenth and twelfth frames.

Figure 7.6: Case 1 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	1.5
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	1 pixel
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 22, 24, 28
False alarm in frame:	38

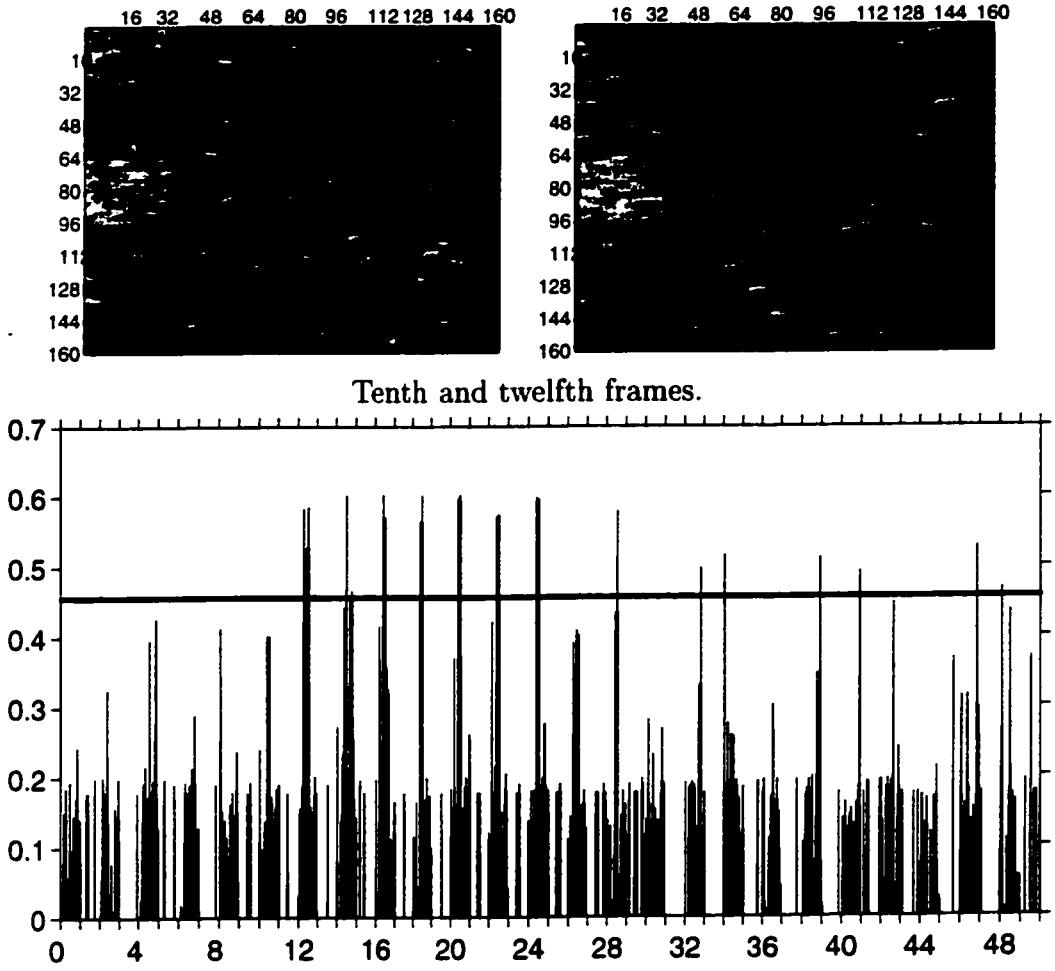


Figure 7.7: Case 2 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	2
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	1 pixel
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	12, 14, 16, 18, 20, 24, 28
False alarms in frames:	32, 34, 38, 40, 46, 48

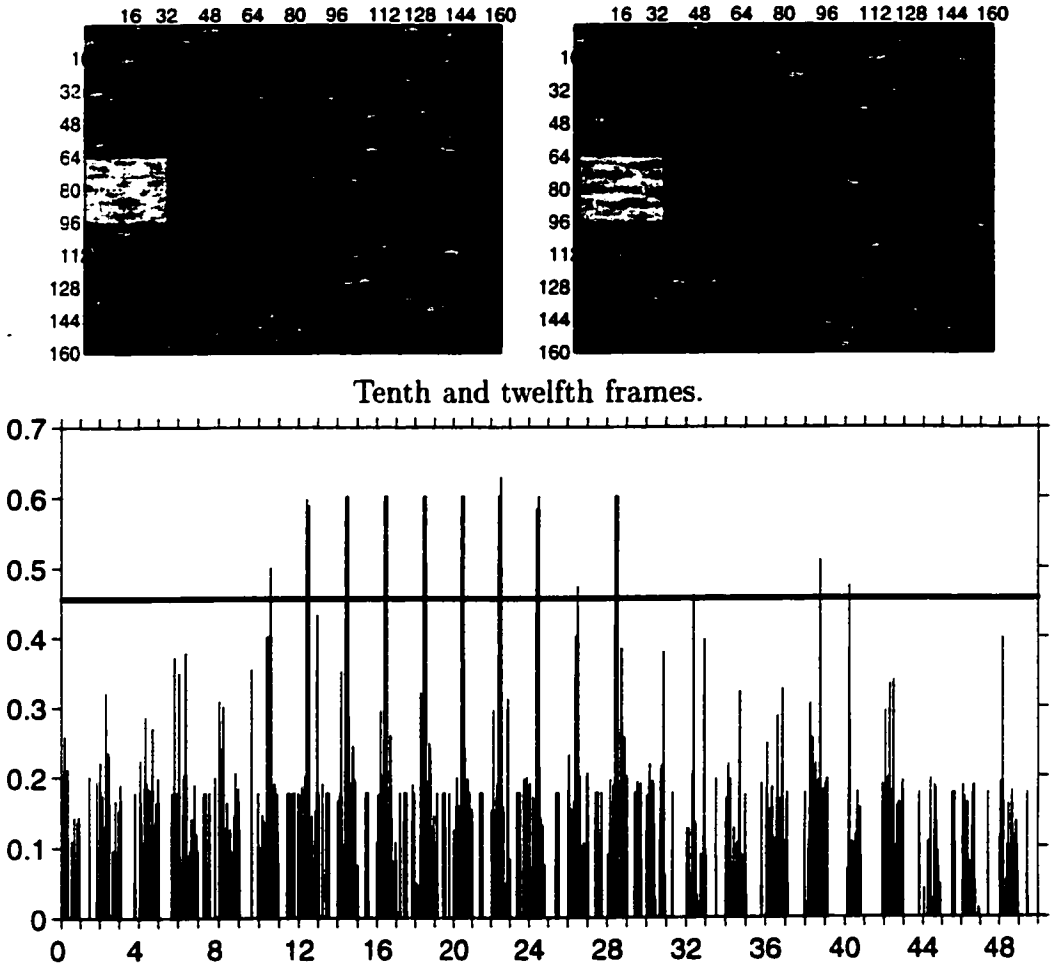


Figure 7.8: Case 3 cusum

Spatial averaging:	3 × 7
Time averaging:	Back 3 frames
Size of added box:	32 × 32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	1 pixel
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 22, 24, 26, 28
False alarms in frames:	38, 40

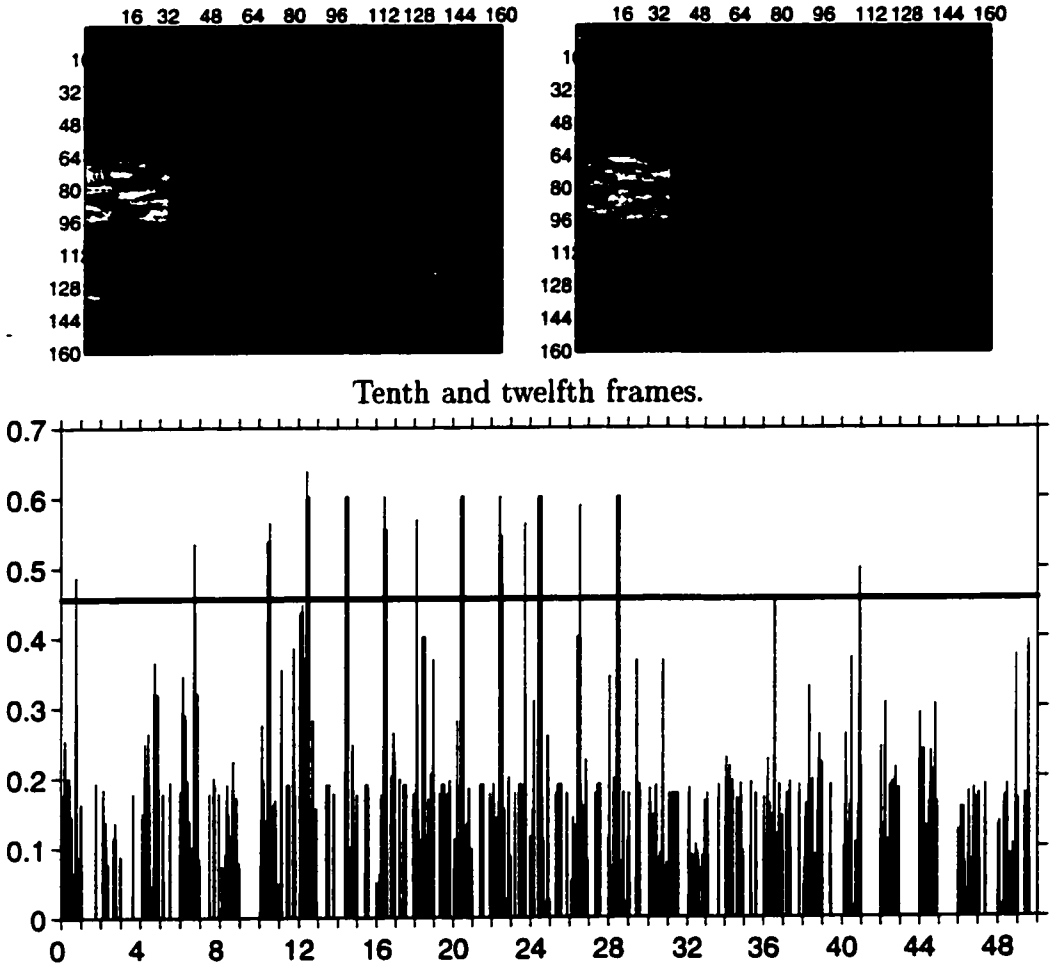
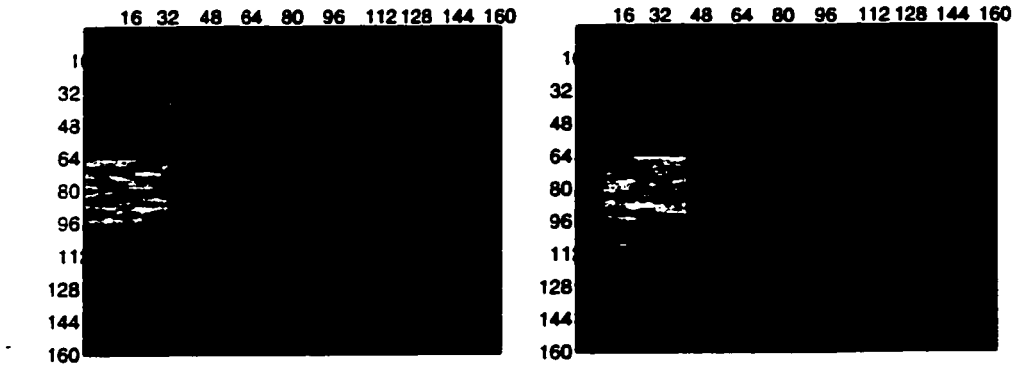


Figure 7.9: Case 4 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	2 pixels
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 22, 23, 24, 26, 28
False alarm in frames:	0, 36, 40



Tenth and twelfth frames.

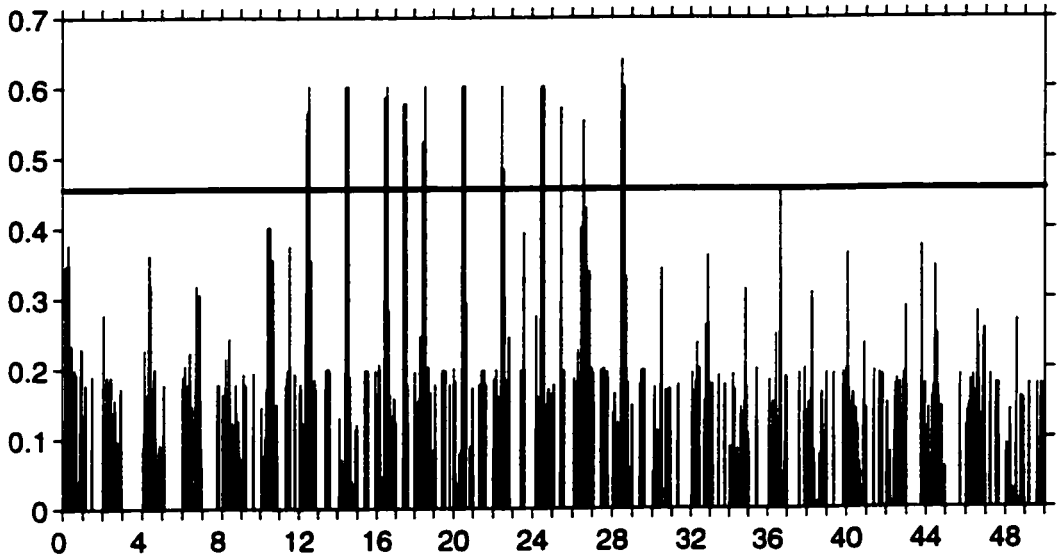
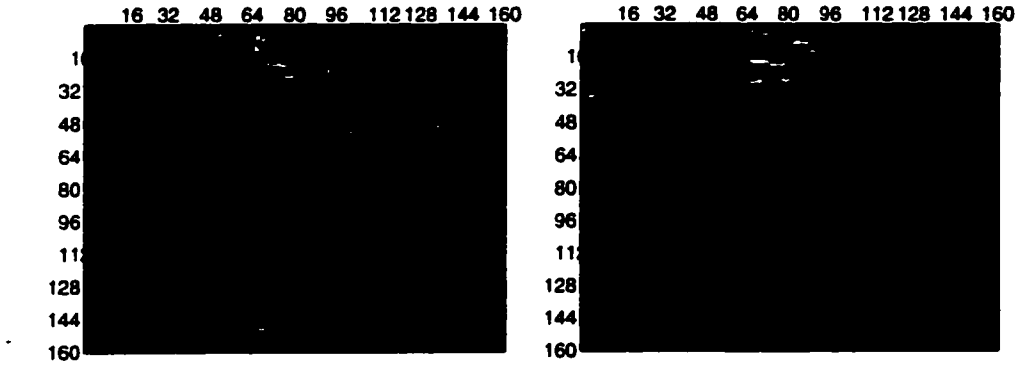


Figure 7.10: Case 5 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	5 pixels
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	12, 14, 16, 17, 18, 20, 22, 24, 25, 26, 28
False alarm in frame:	36



Tenth and twelfth frames.

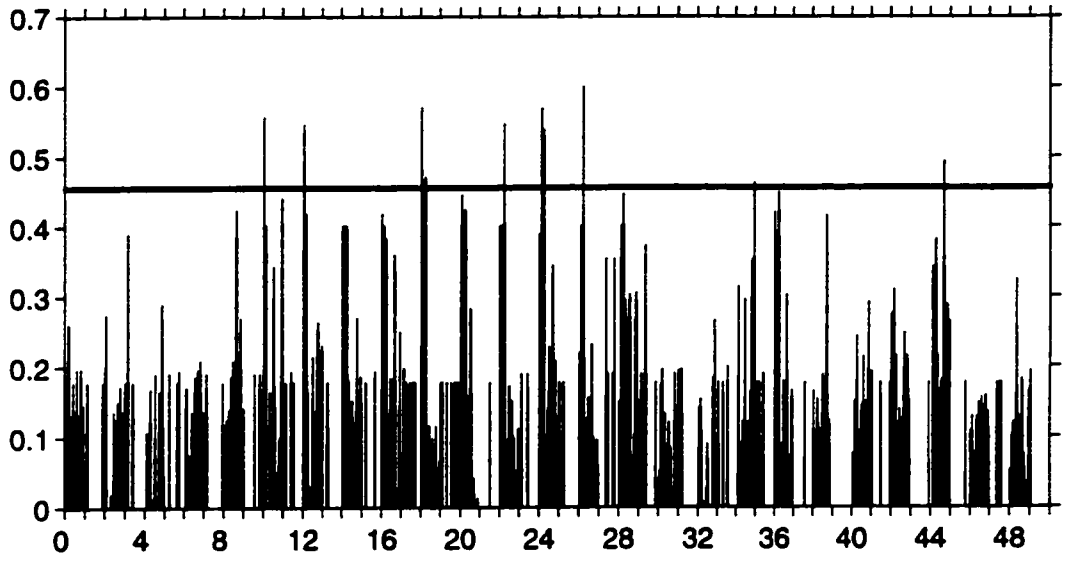
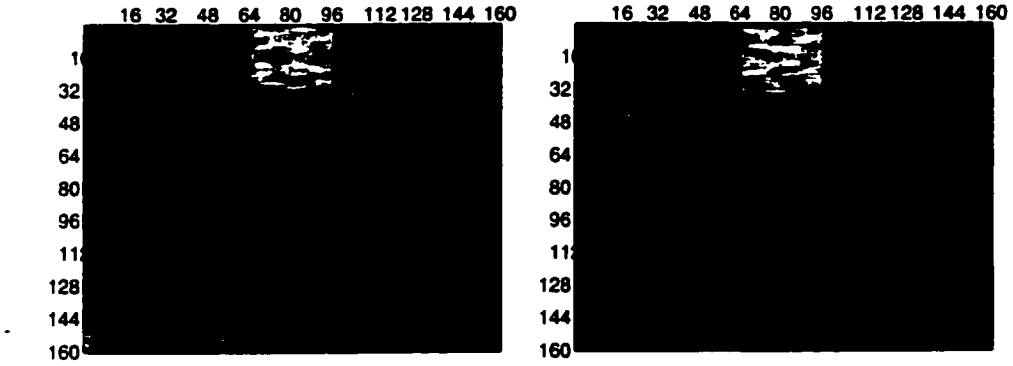


Figure 7.11: Case 6 cusum

Spatial averaging:	3 × 7
Time averaging:	Back 3 frames
Size of added box:	32 × 32 pixels
Intensity of added box:	1.5
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	1 pixel
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 18, 22, 24, 26
False alarms in frames:	34, 36, 44



Tenth and twelfth frames.

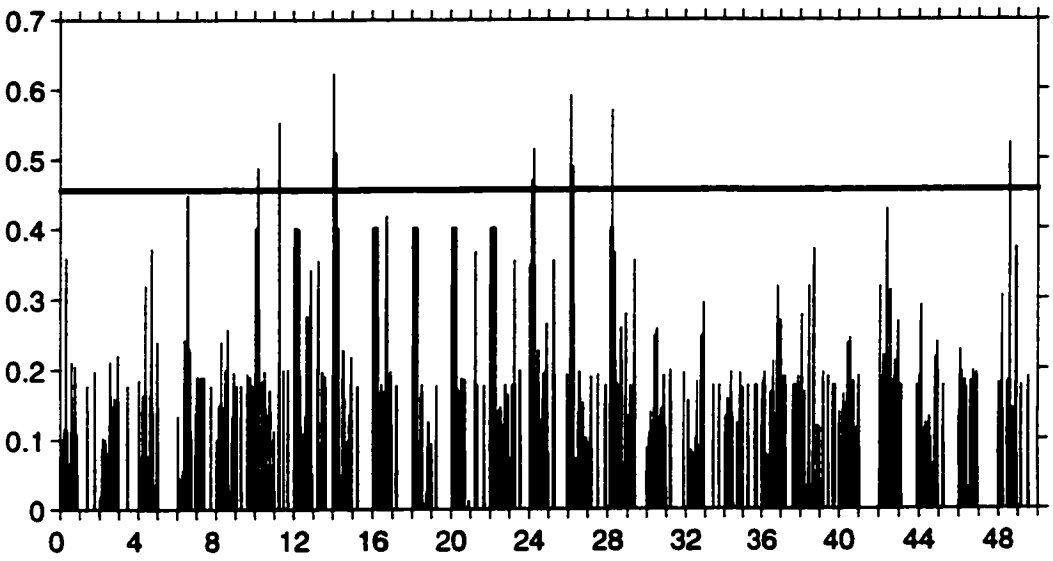
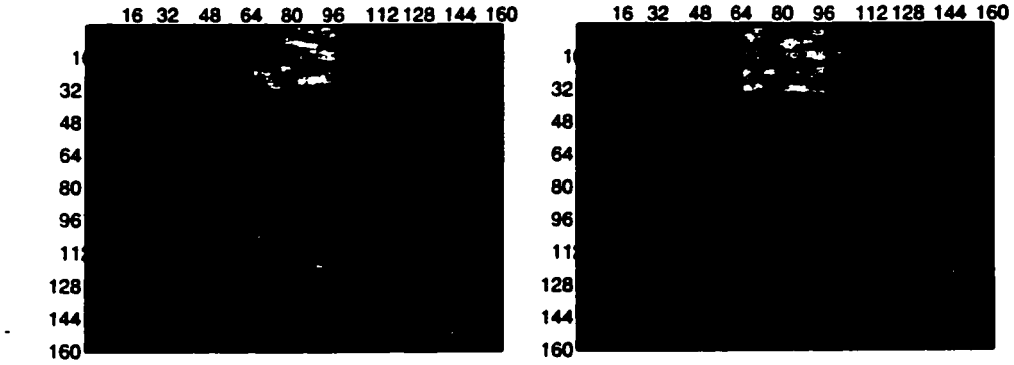


Figure 7.12: Case 7 cusum

Spatial averaging:	3 × 7
Time averaging:	Back 3 frames
Size of added box:	32 × 32 pixels
Intensity of added box:	2
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	1 pixel
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 11, 14, 24, 26, 28
False alarm in frame:	48



Tenth and twelfth frames.

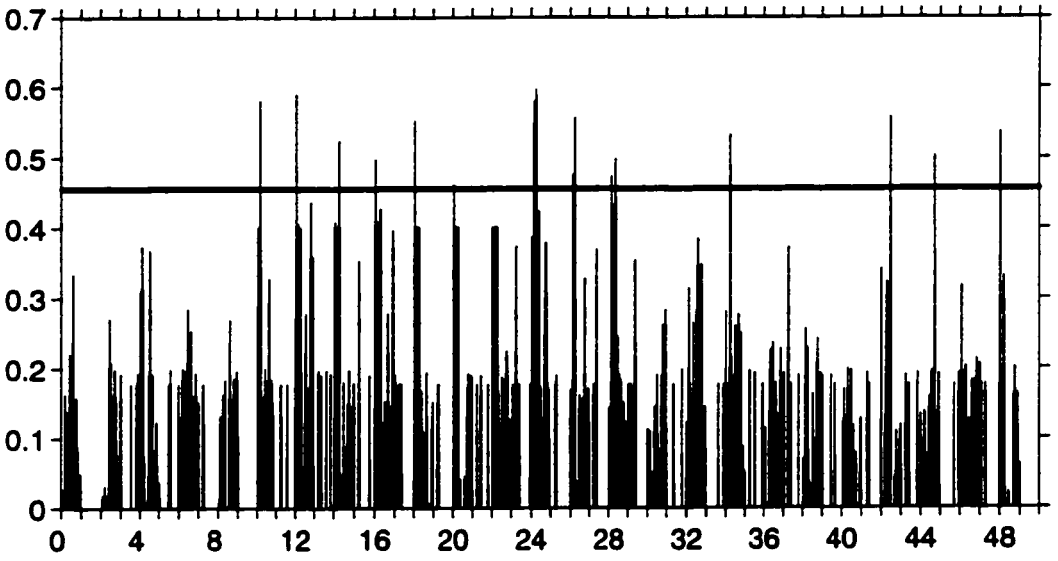


Figure 7.13: Case 8 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	1 pixel
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 24, 26, 28
False alarm in frames:	42, 44, 48

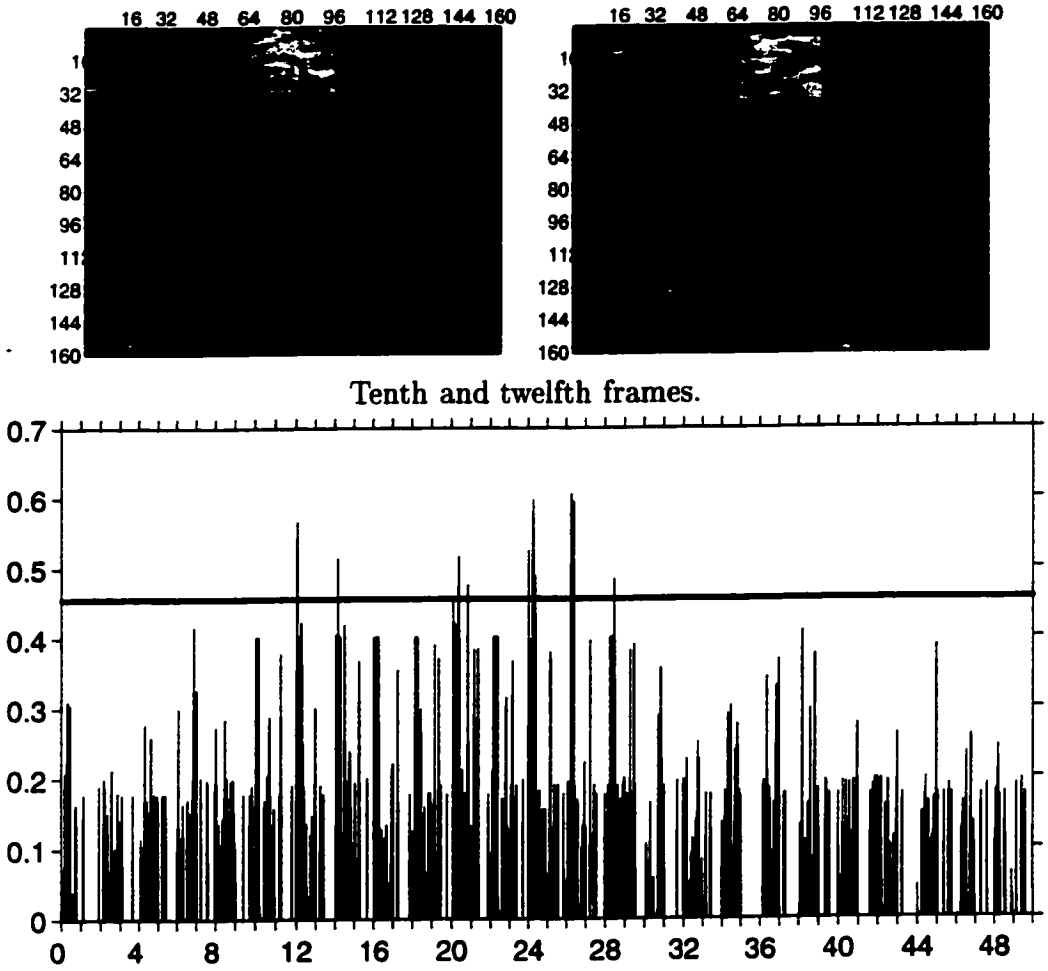


Figure 7.14: Case 9 cusum

Spatial averaging:	3×7
Time averaging:	Back 3 frames
Size of added box:	32×32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	2 pixels
Size of frame:	160×160 pixels
Number of frames:	50
Box detected in frames:	12, 14, 20, 24, 26, 28
False alarms in frames:	No false alarms

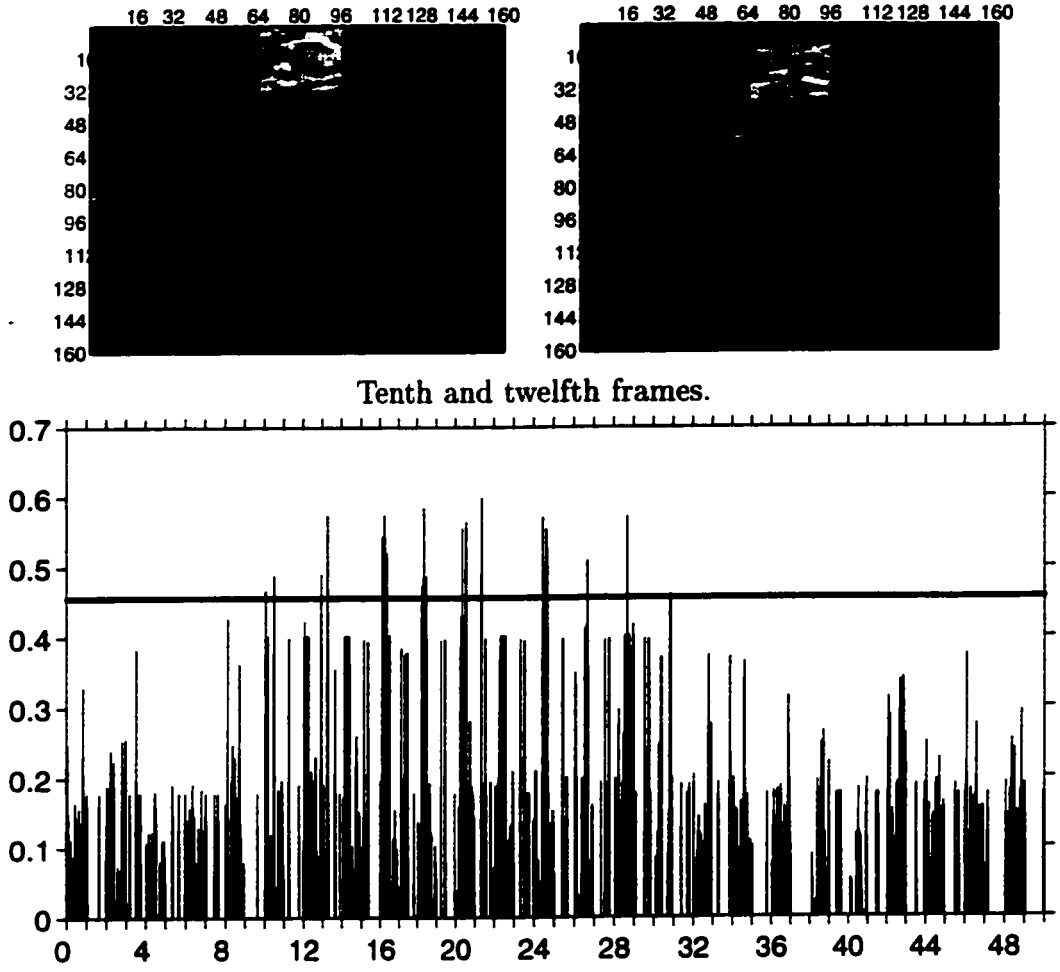


Figure 7.15: Case 10 cusum

Time averaging:	Back 3 frames
Size of added box:	32 × 32 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	5 pixels
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 13, 16, 18, 20, 21, 24, 26, 28
False alarm in frame:	30

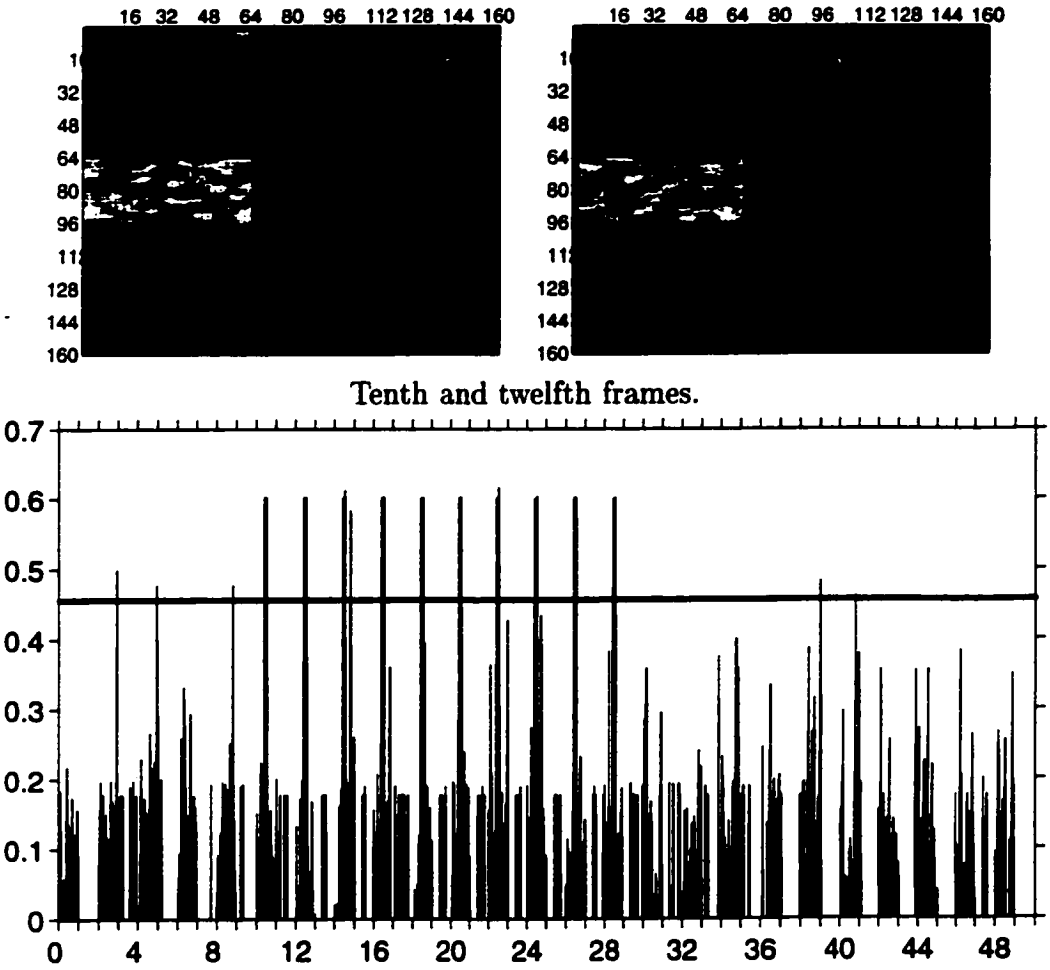


Figure 7.16: Case 11 cusum

Time averaging:	Back 3 frames
Size of added box:	32 × 64 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	1 pixel
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 22, 24, 26, 28
False alarms in frames:	2, 4, 8, 38, 40

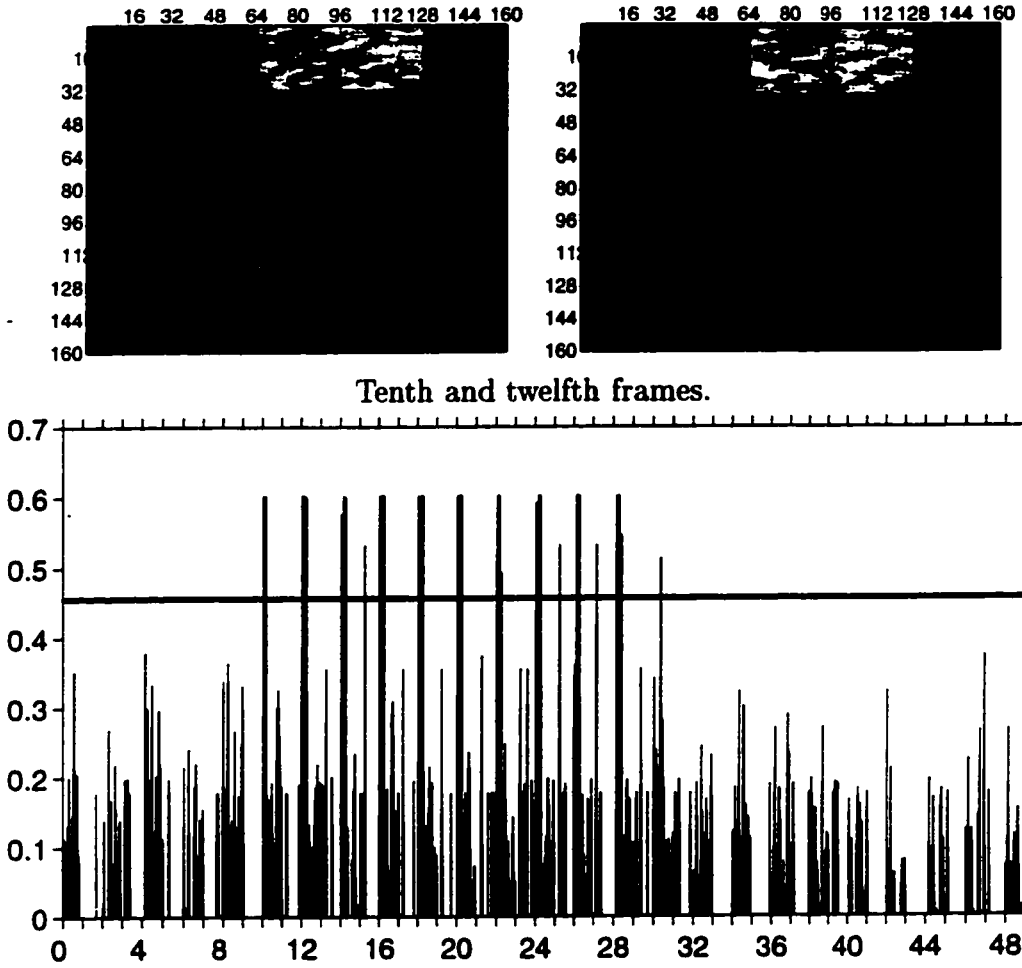


Figure 7.17: Case 12 cusum

Time averaging:	Back 3 frames
Size of added box:	32 × 64 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	1 pixels
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 13, 14, 16, 18, 20, 22, 24, 25, 26, 28
False alarm in frame:	30

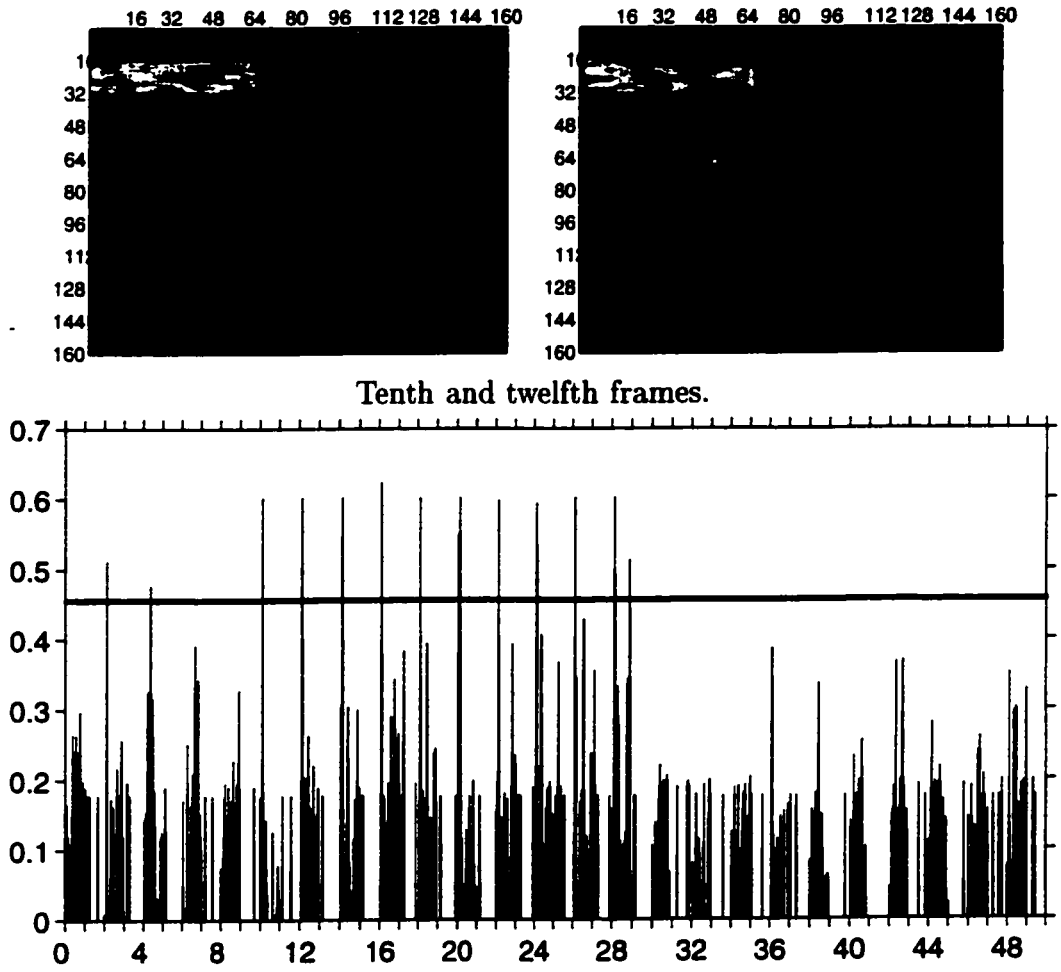


Figure 7.18: Case 13 cusum

Time averaging:	Back 3 frames
Size of added box:	16 × 64 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Horizontal
Motion step size:	1 pixel
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 16, 18, 20, 22, 24, 26, 28
False alarms in frames:	2, 4

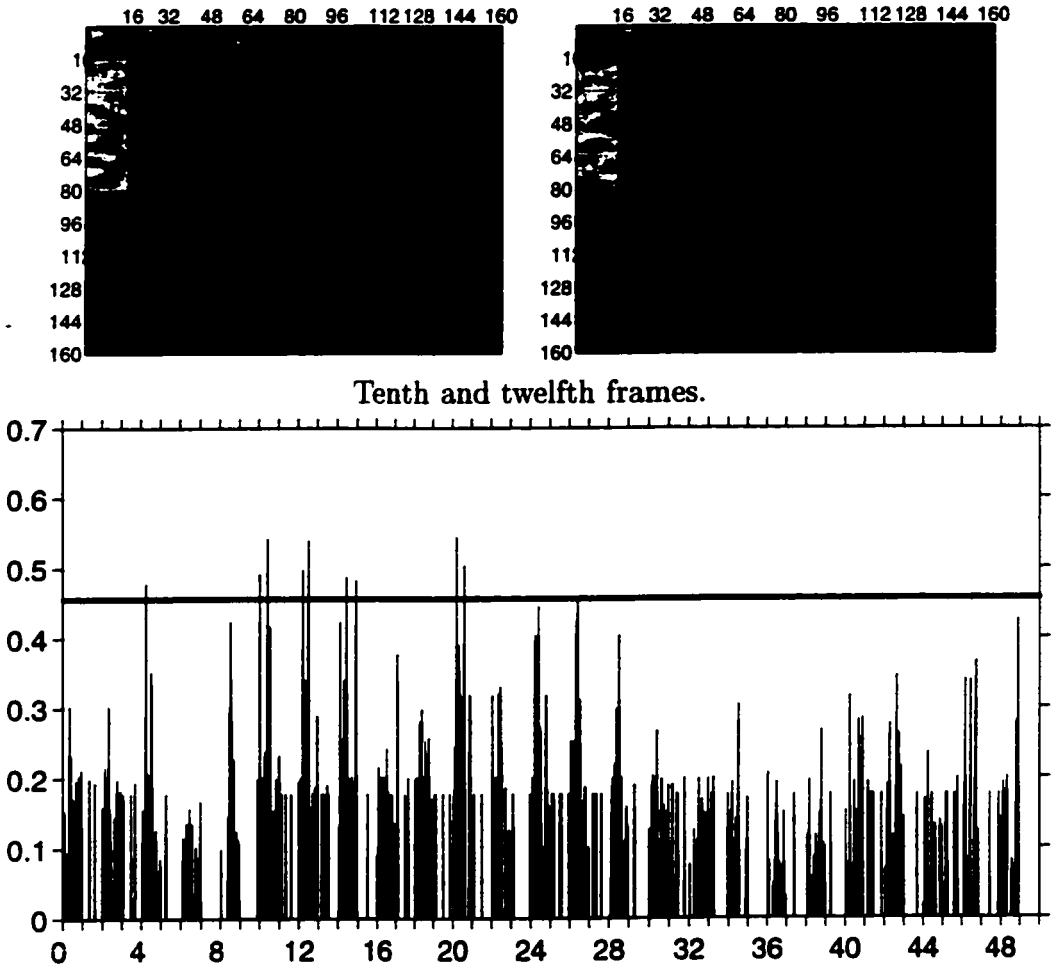
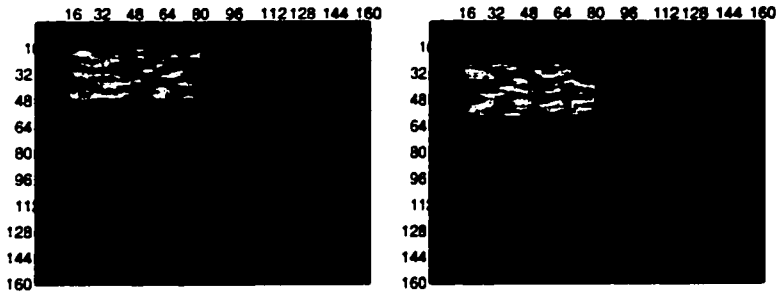
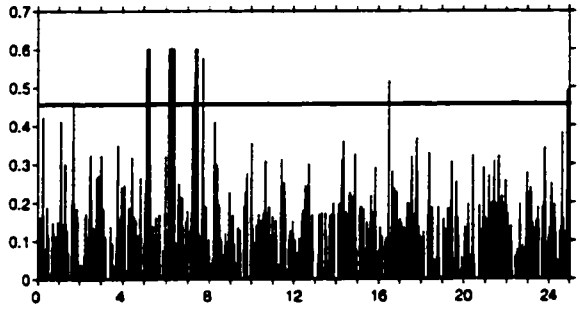


Figure 7.19: Case 14 cusum

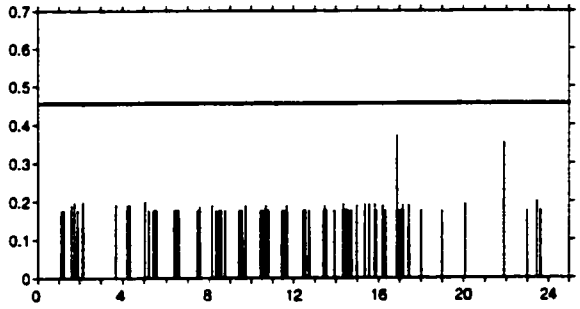
Time averaging:	Back 3 frames
Size of added box:	64 × 16 pixels
Intensity of added box:	3
Box added in frames:	10–29
Direction of box motion:	Vertical
Motion step size:	1 pixels
Size of frame:	160 × 160 pixels
Number of frames:	50
Box detected in frames:	10, 12, 14, 20, 26
False alarm in frame:	4



Tenth and twelfth frames.



I-frame cusum.



P-frame cusum.

Figure 7.20: Cusum of I- and P-frames done separately.

Time averaging:	Back 3 frames
Size of added box:	32 × 64 pixels
Intensity of added box:	3
Box added in frames:	10–15
Direction of box motion:	Vertical
Motion step size:	5 pixels
Size of frame:	160 × 160 pixels
Number of frames:	50 (25 I-frames, 25 P-frames)
Box detected in frames:	10, 12, 14 (I-frame cusum)
False alarm in frame:	4, 32, 48 (I-frame cusum)

7.5 The Problem of Dependence

The attentive reader may have noticed that the implementation of the cusum of p-values contains a serious violation of one of the assumptions of the null hypothesis. Each successive observation is supposed to be independent. However, when frame pixels have been averaged in space and time, and when an added box occupies more than one macroblock, this is clearly no longer true.

What is the impact on the cusum procedure? One effect is beneficial: the power of the test to detect an object increases in certain cases. For example, if an added box spans two horizontally adjacent macroblocks, then given a small p-value for the first macroblock, a small p-value for the second is more likely. However, ARL calculations are no longer accurate. If macroblocks are spatially averaged, a false alarm in one macroblock increases the chances of a false alarm in the next.

Simulations were performed to estimate the actual ARL of the proposed scheme. Framed were spatially averaged over 3×7 blocks and three frames were averaged together back in time. In the macroblock norm, λ was set to 1. Using $\delta = 12.467$ and $h = 0.456$, the cusum was repeated 35472 times, to give an estimated ARL of 1258.5 with standard deviation 1215.2. It is not surprising that the actual observed ARL is larger than 1000, because the motion vector distribution is heavily skewed towards giving p-values of 1, which keeps the cusum down near zero. Figure 7.21 shows the histogram of these lengths. Again, it appears that the run lengths are nearly exponentially distributed. A plot of $y = \frac{1}{1258}e^{-x/1258}$ appears in Figure 7.22.

Estimates of the ARL under a different null hypothesis that assumed dependence between input values were be obtained by simulation. Methods to produce marginally uniform, yet dependent data are not nearly as common as those to produce marginally normal, but dependent sequences. One method for producing uniform, dependent output is discussed (in a different context) by Denker and Keller [6].

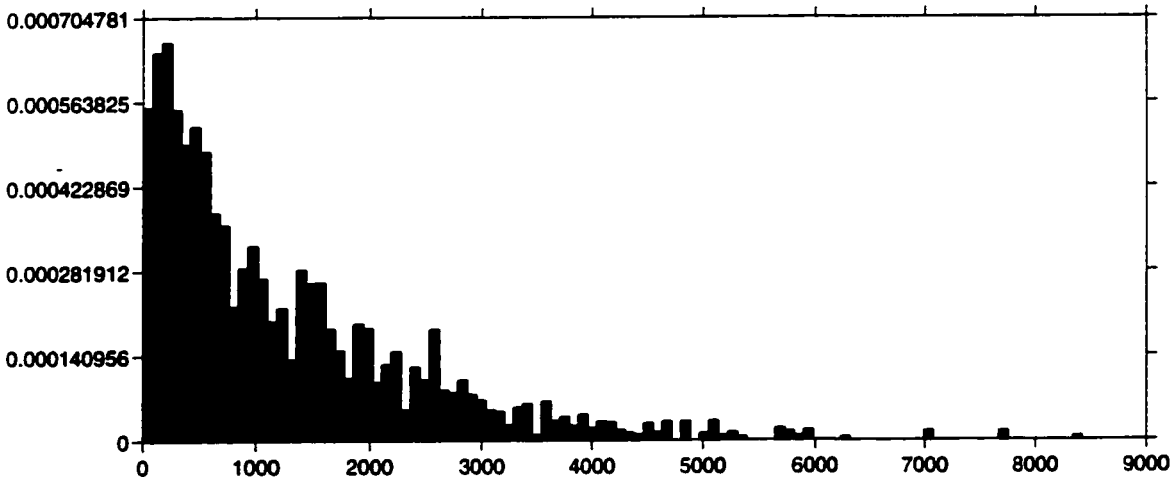


Figure 7.21: Density histogram of the run lengths of the proposed cusum procedure.

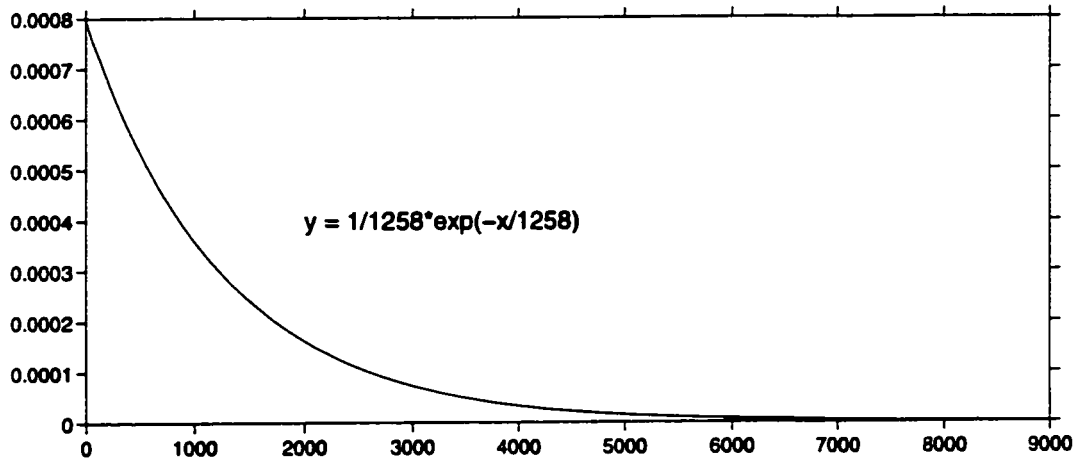


Figure 7.22: Plot of an exponential probability density function with $\lambda = 1258$.

Denker and Keller show that a sequence generated by iterations of an interval transformation form an absolutely regular process.[5] A consequence of this property of absolute regularity is that such a sequence may be regarded as the outcome of a sequence of stationary, asymptotically independent random variables [6].

This method of sequence generation is straight forward to implement. It can be shown that the transform, $T(x) = 1 - |2x - 1|$ defined on the interval $[0, 1]$ has Lebesgue measure as its invariant measure. This means that the terms in the sequence generated by iteratively applying T to its previous outcome each have a marginal distribution that is uniform on $[0, 1]$, but are still dependent (in fact, given the starting seed value, the sequence is deterministic). The graph of this transform is given in Figure 7.23. A plot of a typical sequence generated using this transform procedure is in Figure 7.24. Figure 7.25 shows the histogram of such a sequence of 10000 values, which qualitatively does appear uniform, but Figure 7.24 shows that short-term dependence exists.

Obviously, McDonald's results about the ARL for the cusum of these dependent values are no longer valid, as independence of the input values was integral to the calculations. With $\delta = 12.6407$ and cut-off value $h = 0.456$, (the same parameters as used in previous simulations), the ARL of the cusum of dependent values is 75.6 with standard deviation 78.5, based on 5000 repetitions. Keeping δ at 12.6407 but increasing h to 1 gives an ARL of 607.4 with standard deviation of 606.6, again based on 5000 samples. This is considerably shorter than the expected ARL of 1000 when the samples are independent. Again, it seems likely that the run lengths are exponentially distributed. A histogram of run lengths in Figure 7.26 supports this hypothesis.

[5]Heuristically, a sequence of random variables is absolutely regular if variables that are separated by "large" distances are nearly independent. Technically, a sequence $\{X_i\}$ is called ***absolutely regular*** if the coefficients

$$\beta_n = E\left[\sup\{|P(B||\mathcal{M}_1^k) - P(B)| : B \in \mathcal{M}_{k+n}^{\infty}, k \geq 1\}\right]$$

go to 0 as $n \to \infty$, where $\mathcal{M}_i^j = \sigma(X_i, X_{i+1}, \ldots, X_j)$.

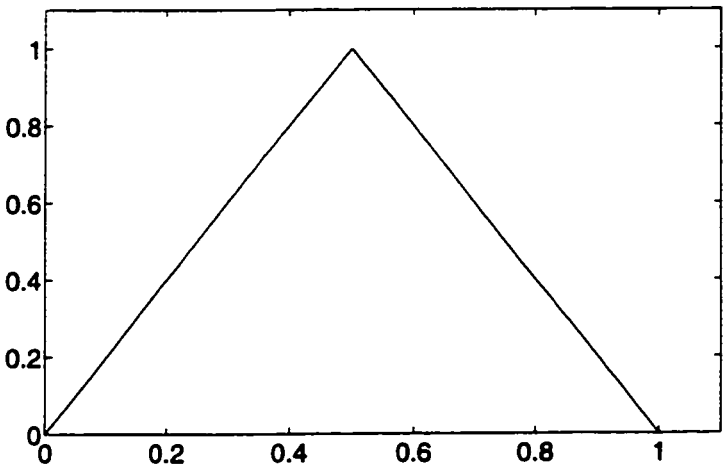


Figure 7.23: Graph of $T(x) = 1 - |2x - 1|$.

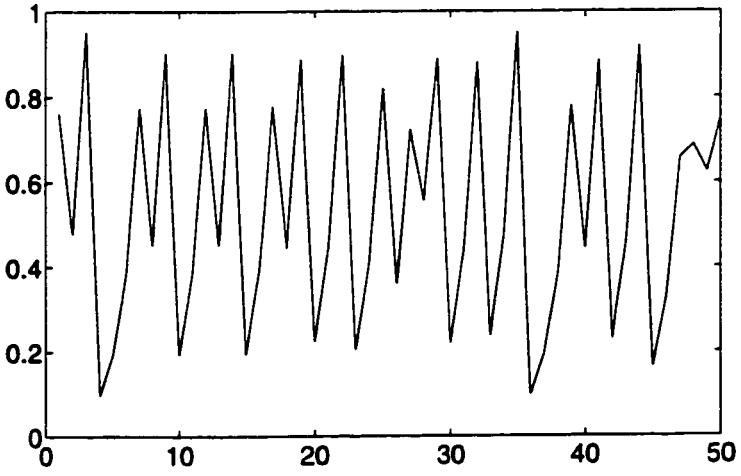


Figure 7.24: A typical sequence generated by transform T. Note the temporal dependence—the values tend to oscillate.

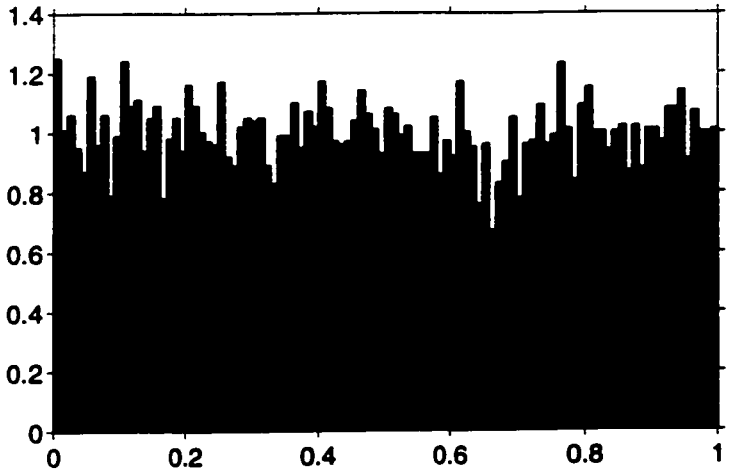


Figure 7.25: Density histogram of the dependent sequence.
There are 10000 values in this sample.

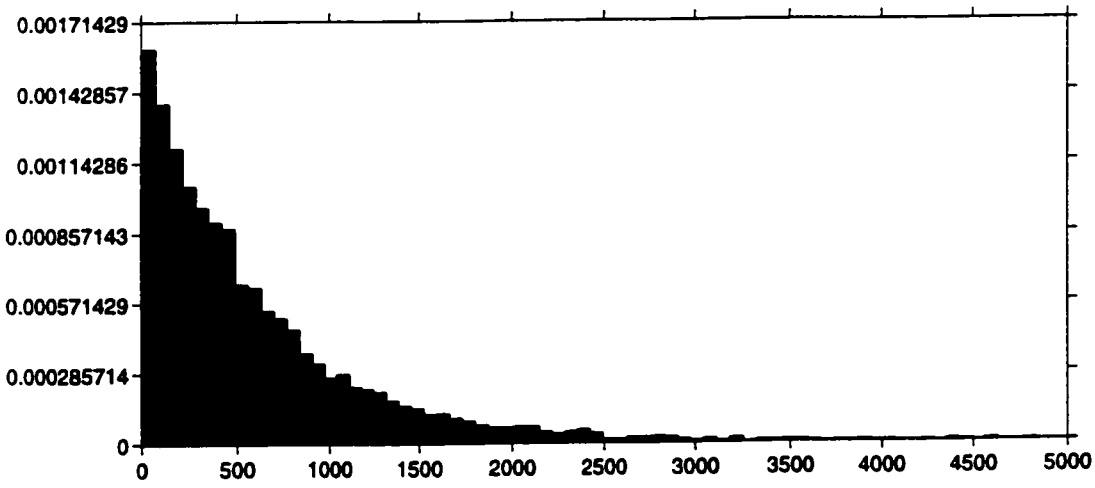


Figure 7.26: Density histograms of run lengths.

In this simulation, the cusum was repeated 5000 times, with $\delta = 12.6407$ and $h = 1.0$. Note the shorter run lengths than in the independent case, shown in Figure 7.3.

Chapter 8

Conclusions and Further Directions

A procedure for determining when an object enters a noisy MPEG compressed video by only examining the "MPEG coefficients" has been developed. Certainly, the efficacy of the this test is significantly affected by the type of object and its direction and manner or motion. Also, it is difficult to extract meaningful information about the motion of an object within a macroblock using only the associated motion vector. This severely hampers the power of the test for the parts of the video encoded as P-frames. Nonetheless, for objects of reasonable size and brightness appearing in I-frames, the test is quite successful in detection.

To develop this procedure to the level in which it could actually be used on heart ultrasound MPEG data, it would be necessary to extract more information from P- (and B-) frames, especially considering that most MPEG encoders achieve high compression rates by using large proportions of these types of frames. Another major issue with analysing heart ultrasound data is removing the moving heart information from the images—heart motion itself should not trigger an alarm.

There are many opportunities for research on cusum procedures as themselves. The most studied cases usually assume independent, normally distributed samples. Although often much less mathematically tractable, other distributions and dependence relationships certainly occur in the real world, and incorporating this additional information into the model may improve the effectiveness of the cusum procedure.

Appendix A

Introduction to Entropy

This appendix provides just enough details about information theory to prove that under certain conditions, the average codeword length of a uniquely decodable code is at least as large as the entropy of the source. For simplicity, we will restrict our discussion to discrete random variables.

Definition A.1 (Entropy) *Let X be a discrete random variable with output $\mathcal{X} = \{x_1, x_2, \dots, \}$ and probability mass function $p(x) = P[X = x]$. The entropy of X, denoted $H(X)$, is given by*

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x),$$

and since we are using $log_2(x)$, it is measured in bits.

Heuristically, entropy is a measure of the uncertainty in a random variable, uncertainty in the sense of the probability of observing different outcomes.

Definition A.2 (Relative Entropy) *Consider the probability mass functions p and q which have the same output, $\mathcal{X}$. The relative entropy of p with respect to q is,*

$$D(p||q) = \sum_{x \in \mathcal{X}} p(x) \log_2 \frac{p(x)}{q(x)}.$$

Although not a true metric, $D(p||q)$ is sometimes thought of as the distance between p and q.

Lemma A.1 *$D(p\|q) \geq 0$, with equality iff $p(x) = q(x)$ for all x.*

PROOF: Let the support of X be $A = \{x : p(x) > 0\}$.

$$\begin{aligned}
-D(p\|q) &= -\sum_{x\in A} p(x)\log_2\frac{p(x)}{q(x)} \\
&= \sum_{x\in A} p(x)\log_2\frac{q(x)}{p(x)} \\
&= E[f(Y)], \quad \text{where } Y = q(X)/p(X), f(y) = \log_2(y) \\
&\leq f(E[Y]) \qquad \text{(A.1)} \\
&= \log_2\left(\sum_{x\in A} p(x)\frac{q(x)}{p(x)}\right) \\
&= \log_2\left(\sum_{x\in A} q(x)\right) \\
&\leq \log_2\left(\sum_{x\in \mathcal{X}} q(x)\right) \\
&= \log_2 1 \\
&= 0.
\end{aligned}$$

Line (A.1) follows from Jensen's inequality. We may now conclude that $D(p\|q) \geq 0$, with equality iff $p(x) = q(x)$ for every x. □

Theorem A.1 *$H(X) \leq \log_2|\mathcal{X}|$, which holds with equality if X is uniformly distributed on $\mathcal{X}$, and where $|\mathcal{X}|$ is the number of elements in the range of X.*

PROOF: Let $u(x) = \frac{1}{|\mathcal{X}|}$, the uniform pdf on $\mathcal{X}$. Let $p(x)$ be the pdf of X.

$$\begin{aligned}
D(p\|u) &= \sum_{x\in\mathcal{X}} p(x)\log_2\frac{p(x)}{u(x)} \\
&= \sum \left(\log_2 p(x) - \log_2 u(x)\right) p(x) \\
&= \sum p(x)\log_2 p(x) + \sum p(x)\log_2\frac{1}{u(x)} \\
&= -H(X) + \sum p(x)\log_2|\mathcal{X}| \\
&= -H(X) + \log_2|\mathcal{X}|.
\end{aligned}$$

Since $D(p||u) \geq 0$, we have $H(X) \leq \log_2 |\mathcal{X}|$. If $p(X) = 1/|\mathcal{X}|$, then $H(X) = \log_2 |\mathcal{X}|$. □

Definition A.3 (Prefix or instantaneous code) *A variable length code in which no codeword is the prefix of any other codeword.*

For example, mapping A, B and C to $01, 001$ and 0001 is a prefix code. The sequence 010001001001 can only be decoded as $ACBB$.

Definition A.4 (Alphabet) *The symbols that are used to make codewords.*

The alphabet of the previous example is $\{0, 1\}$.

Theorem A.2 (Kraft's Inequality) *For any prefix code with a finite number of codewords over an alphabet of size D, the codeword length $l_1, l_2, \ldots, l_m$ must satisfy,*

$$\sum_i D^{-l_i} \leq 1.$$

Conversely, given a set of lengths, there exists a D-ary code satisfying the inequality.

PROOF: Notice that any prefix code can be written along the paths of a D-ary tree, as in Figure A.1.

Reading along the paths to the end node gives the codewords. Because of the prefix condition, no codeword may continue beyond the end node of another codeword.

Let $l_{\max}$ be the length of the longest codeword. There are $D^{l_{\max}}$ possible nodes at level $l_{\max}$. Only some of these nodes are codewords, the others are defunct extensions of codewords, or neither of these. Consider a codeword that ends at level l_i. It has $D^{l_{\max}-l_i}$ (defunct) descendents. The total number of descendents of all codewords is

$$\sum_i D^{l_{\max}-l_i}.$$

This must be less than or equal to the total number of possible nodes at level $l_{\max}$. Thus,

$$\sum_i D^{l_{\max}-l_i} \leq D^{l_{\max}}.$$

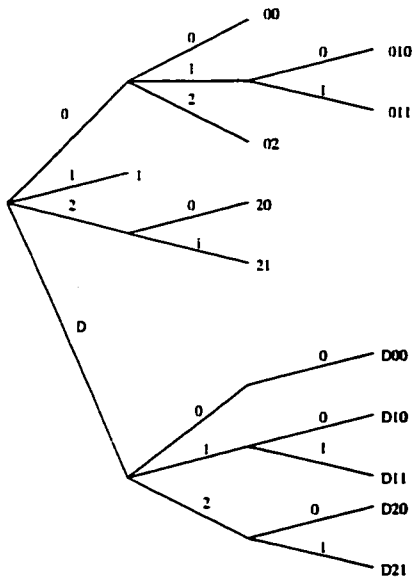


Figure A.1: Tree diagram of a D-ary prefix code.

Dividing by $D^{l_{\max}}$ gives

$$\sum_i D^{-l_i} \leq 1.$$

Conversely, given codeword lengths $l_1, \ldots, l_m$, we can always construct a code along a D-ary tree. □

The Kraft inequality can actually be extended to hold for any uniquely[1] decodable code with a countable number of codewords. For simplicity, the rest of our discussion will consider binary codes only. The results, however, can easily be generalized to hold for the D-ary case.

Theorem A.3 *The expected length L of of a binary prefix code for the output of a random variable X is greater than or equal to the entropy of X. That is,*

$$L \geq H(X),$$

with equality iff $p_i = 2^{-l_i}$, where p_i is the probability of observing codeword i.

[1] A uniquely decodable code need not be a prefix code.

PROOF:

$$\begin{aligned}
L - H(X) &= \sum_i p_i l_i - \sum_i p_i \log_2 \frac{1}{p_i} \\
&= \sum p_i \log_2 2^{l_i} + \sum p_i \log_2 p_i \\
&= \sum p_i \left(\log_2 2^{l_i} + \log_2 p_i + \log_2 \left(\sum_j 2^{-l_j} \right) - \log_2 \left(\sum_j 2^{-l_j} \right) \right) \\
&= \sum p_i \log_2 \left(\frac{p_i}{2^{-l_i}/(\sum 2^{-l_j})} \right) - \log_2 \left(\sum_j 2^{-l_j} \right)
\end{aligned}$$

Let $r_i = 2^{-l_i} / \sum_j 2^{-l_j}$. Note that r is a probability distribution. Continuing,

$$\begin{aligned}
&= D(p\|r) + \log_2 \left(\frac{1}{\sum_j 2^{-l_j}} \right) \\
&\geq 0.
\end{aligned}$$

The last line follows because we have proved that relative entropy is always non-negative, and from the Kraft inequality, $\sum_j 2^{-l_j} \leq 1$. □

So, the lower bound on the average codeword length is the entropy of the source. This is the main reason why the MPEG standard applies the DCT transform to the spatial data; it is an effective method of reducing the entropy of the source, because the distribution of the data becomes much less uniform, and as was demonstrated in Theorem A.1, entropy is maximized for the uniform distribution.

Appendix B

Matlab Routine

An invaluable resource for producing nice graphics and movies in Matlab is Marchand [10].

The following function calculates the matrix A that is discussed in Chapter 4, Section 4.2.

```
function functionanswer=covarequations(CovMat)
%
% covarequations(CovMat: a square matrix)
%
% It is assumed that CovMat is the covariance matrix of a
% normal random vector, say Y. It is thus assumed to be symmetric along the
% main diagonal. This is not checked.
%
% This function will return the set of coefficients need to simulate
% Y with covariance relations given by CovMat.
%
% The output, a, is a square matrix of the same dimension as CovMat, and
% is the matrix required to generate Y, using the equation Y = aZ,
% where Z is a normal(0,1) random vector with all components independent.
%
% Note: When CovMat = ones(n), this function does not work, because
% CovMat is not positive-definite.
%

n_temp = size(CovMat);
n = n_temp(1); % This gives the dimension of the square matrix

a = zeros(n);

for j = 1:n,
   for i = j:n,
if (i == j)
  a(i,j) = CovMat(i,j);
  for k = 1:(j-1),
     a(i,j) = a(i,j) - ( (a(i,k))^2 );
  end
  a(i,j) = sqrt(a(i,j));
else % i > j
  a(i,j) = CovMat(i,j);
  for k=1:(j-1),
     a(i,j) = a(i,j) - ( a(j,k)*a(i,k) );
  end
  a(i,j) = a(i,j)/a(j,j);
end
   end
end

functionanswer = a;
```

Appendix C

C++ Programs

A great resource for learning about programming in C++ is by Deitel and Deitel [5].

The following files are used to create the sequence of frames.

```
////////////////////////////////////////////////

// This header file defines a dynamically resizable two dimensional
// array with overloaded operators []
#ifndef TWODARRAY_H
#define TWODARRAY_H

class TwoDimensionalArray
/* This class defines a dynamically resizable two dimentional array,
   that uses standard array notaion, [].
*/
{
public:
  TwoDimensionalArray(void) { m_rows = 0; m_cols = 0; m_data = 0; }
  //  TwoDimensionalArray(int rows, int cols)
  // { m_data = 0; m_setDimensions(rows, cols);}
  TwoDimensionalArray(const int rows, const int cols)
    { m_data = 0; m_setDimensions(rows, cols); }
  TwoDimensionalArray(const TwoDimensionalArray& array);
  ~TwoDimensionalArray(void) { if (m_data) delete (m_data); }

  void m_setDimensions(const int rows, const int cols);
  double* operator[](const int x) { return m_data + x * m_cols; } // defines
  // an extention of standard array notation, but where the array can
  // be sized dynamically. Below is the const version, to deal with
  // constant functions.
  // So to use these operators on TwoDimensionalArray array, the call is
  //  array[row][col]
  const double* operator[](const int x) const { return (const double*)m_data
    + x * m_cols; }

  const TwoDimensionalArray& operator=(const TwoDimensionalArray& right);
  int m_getRows(void) { return m_rows; }
  int m_getCols(void) { return m_cols; }

private:
  double* m_data; // pointer to the actual data
  int m_rows; // number of rows in the array
  int m_cols; // number of columns in the array
};

#endif TWODARRAY_H

////////////////////////////////////////////////
```

```
// max.h
/* This header file defines the function max, which returns the larger
   of two doubles. */

#ifndef MAX_H
#define MAX_H

inline static double max(double a, double b)
{
  if (a > b)
    return a;
  else
    return b;
}

inline static double min(double a, double b)
{
  if (a < b)
    return a;
  else
    return b;
}

#endif MAX_H

////////////////////////////////////////////////

// This is the header file for the function that is used to create
// normally distributed data.

#ifndef GAUSS_H
#define GAUSS_H

double gauss(double vrnc);

#endif GAUSS_H


////////////////////////////////////////////////

// Frame parameter header file
// March 12, 2001
// This file contains the definition of the frame parameter class

#ifndef FRAME_PARAM_H
#define FRAME_PARAM_H
```

```
class FrameParam
{
 public:
  const int m_avgBH; // averaging block height -- this is assumed to be odd
  const int m_avgBW; // averaging block width -- this is assumed to be odd
  const int m_avgT;  // number of frames averaged back in time -- even or odd
  const int m_numMB; // number of macroblocks along one side (square
                     // frames have been assumed)
  const int m_frameRows; // number of rows in the frame, 16*m_numMB
  const int m_frameCols; // number of cols in the frame, 16*m_numMB
  const double m_lambda; // parameter in norm() -- must be non-negative
  char* m_baseFrameFile;     // base file name that frames will be written
                             // to
  const int m_collectMV; // if this is non-zero, then the actual
  // motion vectors will be collected, a tally of the different motion
  // vectors for a movie will be kept and all of the information will
  // be written to disk. Note: motion vector data will only be
  // collected for P-frames.

  FrameParam(const int avgBH, const int avgBW,
     const int avgT, const int numMB,
     const double lambda, char baseFrameFile[],
     const int collectMV);
  ~FrameParam(void);
};

#endif FRAME_PARAM_H

////////////////////////////////////////////////////

// box.h
// This header file defines a class for storing the parameters needed
// to add a box to a frame.

#ifndef BOX_H
#define BOX_H

class BoxParam
{
 public:
  const int m_boxH; // height of box
  const int m_boxW; // width of box
  const int m_startRow; // row to put upper left hand corner of box
  const int m_startCol; // column to put upper left hand corner of box
  const double m_intensity; // amount to add to each pixel value
  const int m_stepVal; // how much to shift each box each frame
  const int m_direction; // non-zero for horizontal movement, 0 for
                         // vertical movement
  const int m_startTime; // first frame to add box in
  const int m_stopTime; // frame after the last frame to have a box added

  BoxParam(const int boxH, const int boxW, const int startRow,
   const int startCol, const double intensity, const int stepVal,
   const int direction,
   const int startTime, const int stopTime);
};
#endif BOX_H

////////////////////////////////////////////////////

// Frame header file
// March 7, 2001
// This file contains the definition of the Frame, Iframe, Pframe classes
#ifndef FRAME_H
#define FRAME_H

#include "twoDarray.h"
#include "frame_param.h"

class Frame
{
 public:
  Frame(const int number, const FrameParam* param);
  virtual ~Frame(void);

  void initFrame(void); // puts normal(0,1) independent random noise
                        // in m_mid "temporary" pixels
  void addBox(const int boxH, const int boxW, const int row,
      const int col, const double intensity); // adds a box of
  // intensity "intensity" at top left hand corner row, col and of
  // height boxH and width boxW
  virtual void findStats(const Frame& refFrame) = 0;
  // Will be defined to calculate the appropriate test statistics for
  // either I or P frames.
  // I-frames will not need the input data, but it is required in
  // P-frames, and so must be included because virtual functions must
  // have the same signature
  void writeMpegData(void); // writes m_mpegData to file
  void writePixelData(void); // write m_pixels to file
  const double blockSum(const int row, const int col) const; // finds
  // the sum of pixels in an "averaging" block; used for finding
  // actual pixel values in averaged frame

  TwoDimensionalArray m_pixels; // final frame pixel values
```

```
  const int m_number; // frame number: I-frames have even frame
  // number, P-frames odd frame number

  virtual void writeMotionVectors(void) = 0;
  int getCollectMV(void) { return m_param->m_collectMV; }
 protected:
  const FrameParam* m_param; // frame parameters (averaging block size etc)
  char* m_frameFile; // actual filename m_mpeg data will be written to

  TwoDimensionalArray m_iid;   // initial iid normal pixel values

  double* m_mpegData; // dynamically allocated array for either motion
                      // vector lengths or DCT coefficient statistics
};

class Iframe : public Frame
{
  friend void initDCTmatrix(void); // will initialize the DCTmatrix
 public:
  Iframe(const int number, const FrameParam* param);
  void findStats(const Frame& refFrame);
  void writeMotionVectors(void) {}; // we only need this virtual
  // function for P-frames

 protected:
  static TwoDimensionalArray matrixMultBlock(const TwoDimensionalArray& A,
     const int AtopRow,
     const int AtopCol,
     const TwoDimensionalArray& B,
     const int BtopRow,
     const int BtopCol);
  static TwoDimensionalArray DCTmatrix; // matrix needed to calculate
  // DCT
};

class Pframe : public Frame
{
  friend void initStaticMvTally(void); // initializes mvTally
  friend void writeMvTally(const FrameParam* param); // writes mvTally to disk
 public:
  Pframe(const int number, const FrameParam* param);
  void findStats(const Frame& refFrame);
  void writeMotionVectors(void); // writes actual motion vectors to disk

 protected:
  const double norm(const Frame& refFrame, const int refRow,
    const int refCol, const int currRow,
    const int currCol) const;
  TwoDimensionalArray m_motionVectors; // a (number of motion vectors) x
  // 2 matrix that will hold the x and y coordinates of the motion
  // vectors. NOTE: motion vectors will be reported using CARTISEAN
  // coordinates.
  static int mvTally[33][33]; // will hold a count of each type of
                              // motion vector in a movie
};
#endif FRAME_H

//////////////////////////////////////////////

// File queue.h
// This file contains the class definitions required for a linked list
// queue

#ifndef QUEUE_H
#define QUEUE_H

#include "frame.h"
#include "frame_param.h"

class Node
{
  friend class Queue;
 public:
  Node(const int frameType, const int frameNum, const FrameParam* frameParams);
  Frame* frame; // data
  Node* nextPtr; // pointer to next element in list
};

class Queue
// This class implements a LIFO queue. The head represents the oldest
// members and the tail the newest. Thus, we will add to the tail (as
// one normally goes to the end of a line), and
// delete from the head (as those in front are served first).
{
 public:
  Queue(void);
  ~Queue(void);
  void enqueue(const int frameType, const int frameNum,
       const FrameParam* frameParams);
  int dequeue(void);
  int isEmpty(void) const;

  Node* headPtr;
  Node* tailPtr;
```

```
 private:
  Node* getNewNode(const int frameType, const int frameNum,
  const FrameParam* frameParams);
};

#endif QUEUE_H

////////////////////////////////////////////////////

// twoDarray.cxx
// This file implements the member functions in the class TwoDimensionalArray

#include "twoDarray.h"
#include <memory.h> // for use of memcopy
#include <stdlib.h>

// copy constructor
TwoDimensionalArray::TwoDimensionalArray(const TwoDimensionalArray& array)
{
  m_data = 0;
  m_setDimensions(array.m_rows, array.m_cols);
  memcpy(m_data, array.m_data, sizeof(double) * m_rows * m_cols);
}

void TwoDimensionalArray::m_setDimensions(const int rows, const int cols)
{
  if (m_data)
    delete m_data;
  m_rows = rows;
  m_cols = cols;
  m_data = new double[m_rows * m_cols];
}

const TwoDimensionalArray& TwoDimensionalArray::operator=
                  (const TwoDimensionalArray& right)
{
  if (&right != this)
    {
      m_setDimensions(right.m_rows, right.m_cols);
      memcpy(m_data, right.m_data, m_rows * m_cols * sizeof(double));
    }
  return *this;

}

////////////////////////////////////////////////
```

```
// gauss.cxx

#include "gauss.h"
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

/* ************************************************************
c. *    This subroutine returns a Gaussian r.v. g with        *
c. *    variance vrnc.  Two r.v.'s, which are iid and         *
c. *    uniformly distributed over [0,1], are generated       *
c. *    using the Knuth's algorithm.                          *
c. *                                                          *
c. *    Depends on external call to srand(seed)           *
c. ************************************************************ */

double gauss(double vrnc)
{
  double   z;
  double   u1, u2;
  double   theta, raylgh;

  u1=(double)rand()/(RAND_MAX+1.0);
  if (u1 == 0)
    {
      u1 = 1;
    }

  u2=(double)rand()/(RAND_MAX+1.0);
  if (u2 == 0)
    {
      u2 = 1;
    }


  theta=2.0*M_PI*u1;
  raylgh=sqrt(-2.0*vrnc*log(u2));
  z=raylgh*cos(theta);
  if  (z > 100)
    {
      fprintf(stderr,"Large Normal Value %8e\n",z);
      z = 3;
    }

  if  (z < -100)
    {
      fprintf(stderr,"Large Normal Value %8e\n",z);
```

```
    x = -3;
  }

 return(x);
}

////////////////////////////////////////////////

// This file contains the definitions of the constructor and
// destructor of the FrameParam class.

#include <string.h>
#include <assert.h>
#include "frame_param.h"


FrameParam::FrameParam(const int avgBH, const int avgBW,
       const int avgT, const int numMB,
       const double lambda, char baseFrameFile[],
       const int collectMV)
  : m_avgBH(avgBH), m_avgBW(avgBW), m_avgT(avgT),
    m_numMB(numMB), m_frameRows(16*m_numMB), m_frameCols(16*m_numMB),
    m_lambda(lambda), m_collectMV(collectMV)
{
  assert(m_avgBH % 2 == 1  && m_avgBW % 2 == 1  && m_lambda >= 0
 && m_avgT >= 1 && m_numMB >= 1  && baseFrameFile != "");
  m_baseFrameFile = new char[strlen(baseFrameFile) + 1]; // space
  // allocated for the base file name only
  m_baseFrameFile = strcpy(m_baseFrameFile, baseFrameFile);
}

FrameParam::~FrameParam(void)
{
  if (m_baseFrameFile)
    delete m_baseFrameFile;
}

////////////////////////////////////////////////

// box.cxx
// This file defines the member functions of the class BoxParam

#include "box.h"

BoxParam::BoxParam(const int boxH, const int boxW, const int startRow,
   const int startCol, const double intensity,
   const int stepVal, const int direction,
   const int startTime, const int stopTime)
  : m_boxH(boxH), m_boxW(boxW), m_startRow(startRow), m_startCol(startCol),
    m_intensity(intensity), m_stepVal(stepVal), m_direction(direction),
    m_startTime(startTime), m_stopTime(stopTime)
{}

////////////////////////////////////////////////

// frame.cxx
// This file defines the member functions of the Frame class
// in file frame.h

#include <assert.h>
#include <values.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include "frame_param.h"
#include "frame.h"
#include "gauss.h"
#include "max.h"

// static member declarations
TwoDimensionalArray Iframe::DCTmatrix(8,8);
int Pframe::mvTally[33][33];

// friend function definition
// This function initializes the DCTmatrix, so should be run once at
// the start of the program. It is this matrix that is used to
// calculate the DCT.
void initDCTmatrix(void)
{
  int r, c;

  for(r = 1; r < 8; r++)
    for (c = 0; c < 8; c++)
      Iframe::DCTmatrix[r][c] = sqrt(0.25) * cos( (r * (c + 0.5) * M_PI) / 8);
  for (c = 0; c < 8; c++)
    Iframe::DCTmatrix[0][c] = sqrt(0.5) * sqrt(0.25);
}

// friend function to initialize static mvTally in P-frames
void initStaticMvTally(void)
{
```

```
  int r, c;
  for (r = 0; r < 33; r++)
    for (c = 0; c < 33 ; c++)
      Pframe::mvTally[r][c] = 0;
}

//  function to write mvTally to disk
void writeMvTally(const FrameParam* param)
{
  ofstream outFile;
  int r, c;
  char tallyFile[100];

  snprintf(tallyFile, 11 + strlen(param->m_baseFrameFile) + 1,
   "%s%s", param->m_baseFrameFile, ".tally");


  outFile.open(tallyFile, ios::out);
  if (outFile == 0)
    {
      cerr << "Problems opening mvTally file for writing." << endl;
      exit(1);
    }

  for(r = 0; r < 33; r++)
    {
      for (c = 0; c < 33; c++)
outFile << Pframe::mvTally[r][c] << "\t";
      outFile << endl;
    }
  outFile.close();
}

Frame::Frame(const int number, const FrameParam* param)
  : m_number(number), m_param(param)
{
  m_pixels.m_setDimensions(m_param->m_frameRows, m_param->m_frameCols);
  m_niid.m_setDimensions(m_param->m_frameRows + m_param->m_avgBH - 1,
 m_param->m_frameCols + m_param->m_avgBW - 1);
  // allocating space for the output statistics
  m_mpegData = new double[m_param->m_numMB * m_param->m_numMB];
  if (m_mpegData == 0)
    {
      cerr << "Could not allocate memory for m_mpegData" << endl;
      exit(1);
    }
  // sets the proper m_frameFile name using the base filename
  // assuming our frames will not be numbered higher than
  // 99999
  m_frameFile = new char[5 + strlen(m_param->m_baseFrameFile) + 1];
  snprintf(m_frameFile, 5 + strlen(m_param->m_baseFrameFile) + 1,
   "%s%d", m_param->m_baseFrameFile, m_number);
}

Frame::~Frame(void)
{
  if (m_mpegData)
    delete m_mpegData;
}

void Frame::initFrame(void)
  // Puts independent, normal(0,1) data in each "temporary" (before
  // averaging) pixel.
{
  int r, c;

  for (r = 0; r < (m_param->m_frameRows + m_param->m_avgBH - 1); r++)
    for(c = 0; c < (m_param->m_frameCols + m_param->m_avgBW - 1); c++)
      m_niid[r][c] = gauss(1);
}


const double Frame::blockSum(const int row, const int col) const
  // Finds the sum of the m_niid "pixels" in the block (with size determined by
  // avgBH and avgBW parameters) with upper left hand corner given by
  // (row, col). Note that this is the appropriate sum to be using to
  // calculate the pixel value in m_pixels position (row, col).
{
  int r, c; // counters
  double sum = 0;

  for (r = 0; r < m_param->m_avgBH; r++)
    for (c = 0; c < m_param->m_avgBW; c++)
      sum += m_niid[row + r][col + c];
  return sum;
}

void Frame::addBox(const int boxH, const int boxW, const int row,
   const int col, const double intensity)
  // Adds a box of with boxH rows and boxW columns starting in upper
  // left hand corner (row, col), ie value intensity is added to the
  // pixels in this range.
{
```

```
  int r, c;

  if ( ((boxH + row) > m_param->m_frameRows) || ((boxW + col) >
m_param->m_frameCols))
    {
      cerr << "Box addition extends beyond frame boundaries." << endl;
      exit(1);
    }

  for (r = 0; r < boxH; r++)
    for (c = 0; c < boxW; c++)
      m_pixels[row + r][col + c] += intensity;
}



void Frame::writeMpegData(void)
  // This function will write m_mpegData to file
{
  ofstream outFile;
  int i;

  outFile.open(m_frameFile, ios::out);
  if (outFile == 0)
    {
      cerr << "Problems opening m_mpegData file for writing." << endl;
      exit(1);
    }

  for (i = 0; i < m_param->m_numMB * m_param->m_numMB; i++)
    outFile << m_mpegData[i] << endl;
  outFile.close();
}

void Frame::writePixelData(void)
  // This function will write m_pixels to file.
   // It will probably be used mostly for testing purposes.
{
  ofstream outFile;
  int r, c;
  char newFrameFile[80];

  sprintf(newFrameFile, 9 + strlen(m_param->m_baseFrameFile) + 1,
   "%s%d%s", m_param->m_baseFrameFile, m_number, ".pix");

  outFile.open(newFrameFile, ios::out);
  if (outFile == 0)
    {
      cerr << "Problems opening pixel data file for writing." << endl;
      exit(1);
    }

  for (r = 0; r < m_param->m_frameRows; r++)
    for (c = 0; c < m_param->m_frameCols; c++)
      outFile << m_pixels[r][c] << endl;
  outFile.close();
}

Iframe::Iframe(const int number, const FrameParam* param)
  : Frame(number, param)
{}

TwoDimensionalArray Iframe::matrixMultBlock(
    const TwoDimensionalArray&
    A,
    const int AtopRow,
    const int AtopCol,
    const TwoDimensionalArray&
    B,
    const int BtopRow,
    const int BtopCol)
  // This function calculates
  // the matrix multiplication of A*B, but for the sub arrays of size
  // 8x8 in A and B with top left hand corners (AtopRow, AtopCol) and
  // (BtopRow, BtopCol).

{
  int r, c, k;
  TwoDimensionalArray C(8, 8);

  for (r = 0; r < 8; r++)
    for (c = 0; c < 8; c++)
      C[r][c] = 0;

  for (r = 0; r < 8; r++)
    for (c = 0; c < 8; c++)
      for (k = 0; k < 8; k++)
C[r][c] += A[AtopRow + r][AtopCol + k] * B[BtopRow + k][BtopCol+ c];

  return C;
}

void Iframe::findStats(const Frame& refFrame)
  // This function finds the DCT statistics for I-frames.
```

```
// We never use the input parameters in this instantiation of the
// virtual function.
{
  int currRow, currCol; // counters to move through macroblocks
  int mbRow, mbCol; // counters to move through the four blocks in a macroblock
  double SSmbDCT = 0;
  int i = 0; // counter for m_mpgData
  TwoDimensionalArray matrix1(8, 8);
  TwoDimensionalArray matrix2(8, 8);

  for(currRow = 0; currRow < 15 < m_param->m_frameRows; currRow += 16)
    for(currCol = 0; currCol < 15 < m_param->m_frameCols; currCol += 16)
      {
  SSmbDCT = 0;
  for (mbRow = 0; mbRow < 16; mbRow += 8)
    for (mbCol = 0; mbCol < 16; mbCol += 8)
      {
        // Calculates the DCT on the block
        matrix1 = matrixMultBlock( m_pixels, currRow + mbRow,
  currCol + mbCol, DCTmatrix, 0, 0);
        matrix2 = matrixMultBlock( DCTmatrix, 0, 0, matrix1,
  0, 0);
        // Calculates the sum of squares of the DCT values
        for (int r = 0; r < 8; r++)
  i++;
        }
  for (int c = 0; c < 8; c++)
    SSmbDCT += matrix2[r][c] * matrix2[r][c];
      }
  m_mpgData[i] = SSmbDCT;
  i++;
    }
}

Pframe::Pframe(const int number, const FrameParam* param)
  : Frame(number, param)
{
  m_motionVectors.m_setDimensions(m_param->m_numMB * m_param->m_numMB, 2);
}

const double Pframe::norm(const Frame& refFrame, const int refRow,
  const int refCol, const int currRow,
  const int currCol) const
  // Calculates the sum of the squared difference between each pixel
  // in the two macroblocks, plus lambda times the squared length of
  // the potential motion vector. The current MB is a member of the current
  // object, with top left hand corner (currRow, currCol) and the
  // reference MB is in refFrame with corner (refRow, refCol).
{
  int r, c; // row and column counters
  double weight = 0;

  for(r = 0; r < 16; r++)
    for (c = 0; c < 16; c++)
      {
weight += (refFrame.m_pixels[refRow + r][refCol + c]
  - m_pixels[currRow + r][currCol + c] ) *
  (refFrame.m_pixels[refRow + r][refCol + c]
  - m_pixels[currRow + r][currCol + c] );
      }

  weight += m_param->m_lambda * ((refRow - currRow)*(refRow - currRow) +
      (refCol - currCol)*(refCol - currCol));
  return weight;
}

void Pframe::findStats(const Frame& refFrame)
  // This function finds the motion vectors for the current frame with
  // respect to refFrame.
{
  int r, c; // counters
  int i = -1;
  int currRow, currCol; // counters to show current MB
  double weight = 0;
  double bestWeight = MAXDOUBLE;

  // check that the refFrame is the correct frame (the one immediately
  // preceding the current frame
  assert(m_number == refFrame.m_number + 1);

  for (currRow = 0; currRow + 15 < m_param->m_frameRows; currRow += 16)
    for (currCol = 0; currCol + 15 < m_param->m_frameCols; currCol += 16)
      {
bestWeight = MAXDOUBLE;
i++; // tally of the number of MBs
// Perform a search in a window the size of four macroblocks
for (r = (int)max(currRow - 16, 0);
     r < (int)min(currRow + 17, m_param->m_frameRows - 15);
     r++)
for (c = (int)max(currCol - 16, 0);
     c < (int)min(currCol + 17, m_param->m_frameCols - 15);
     c++)
{
  weight = norm(refFrame, r, c, currRow, currCol);
  if (weight < bestWeight)
```

```
{
  hostHeight = weight;
  m_mpgData[i] = sqrt( (r - currRow)*(r - currRow) +
    (c - currCol) * (c - currCol) );
  if (m_param->m_collectMV)
    {

      m_motionVectors[i][0] = c - currCol;
      m_motionVectors[i][1] = currRow - r;

    }
  }
  }
if (m_param->m_collectMV)
  {
    mvTally[(int)m_motionVectors[i][0] + 16][(int)m_motionVectors[i][1] + 16]  += 1;
  }
    }
}

void Pframe::writeMotionVectors(void)
  /* This function writes the actual x and y coordinates of the motion
     vectors to disk. */
  {
  ofstream outFile;
  int i;
  char mvFrameFile[100];

  sprintf(mvFrameFile, 8 + strlen(m_param->m_baseFrameFile) + 1,
    "%d%s", m_param->m_baseFrameFile, m_number, ".mv");

  outFile.open(mvFrameFile, ios::out);
  if (outFile == 0)
    {
      cerr << "Problems opening motion vector data file for writing." << endl;
      exit(1);
    }

  for (i = 0; i < (m_param->m_numMB * m_param->m_numMB); i++)
    outFile << m_motionVectors[i][0] << "\t" << m_motionVectors[i][1]
      << endl;

  outFile.close();
}

//////////////////////////////////////////////////////////////////////
```

```
// This file defines the member functions of the classes in queue.h

#include <assert.h>
#include <stdlib.h>
#include <iostream.h>
#include "frame.h"
#include "frame_param.h"
#include "queue.h"

Node::Node(int frameType, const int frameNum, const FrameParams* frameParams)
  {
    assert( frameType == 0 || frameType == 1);
    if (frameType == 0)
      {
        frame = new Iframe(frameNum, frameParams);
        nextPtr = 0;
      }
    else // frameType == 1
      {
        frame = new Pframe(frameNum, frameParams);
        nextPtr = 0;
      }
  }

Queue::Queue(void)
  {
    headPtr = tailPtr = 0;
  }

Queue::~Queue(void)
  {
    if (!isEmpty() )
      {
        Node* tempPtr;
        while (headPtr != 0) // delete remaining nodes
  {
    tempPtr = headPtr->nextPtr;
    delete headPtr;
    headPtr = tempPtr;
  }
      }
  }

Node* Queue::getNewNode(const int frameType, const int frameNum,
      const FrameParam* frameParams)
    // creates a new node (usually to be added to the queue)
    {
```

```
  Node* pointer = new Node(frameType, frameNum, frameParams);
  if (pointer == 0)
    {
      cerr << "Could not allocate space for a new node." << endl;
      exit(1);
    }
  return pointer;
}

void Queue::enqueue(const int frameType, const int frameNum,
    const FrameParam* frameParams)
  // Adds a new node at the end of the queue
{
  Node* tempPtr;
  tempPtr = getNewNode(frameType, frameNum, frameParams);

  if (isEmpty() == 1)
    headPtr = tailPtr = tempPtr;
  else
    {
      tailPtr->nextPtr = tempPtr;
      tailPtr = tempPtr;
    }
}

int Queue::dequeue(void)
  // Returns 0 if there is something in the queue to remove, returns 1
  // otherwise.
{
  if (isEmpty() == 0)
    {
      Node* tempPtr = headPtr->nextPtr;
      delete headPtr;
      headPtr = tempPtr;
      return 0;
    }
  else
    return 1;
}

int Queue::isEmpty(void) const
  // This function returns 1 if the queue is empty, and returns
  // otherwise.
{
  if (headPtr == 0)
    return 1;
  else
    return 0;
}

//////////////////////////////////////////////////////////

// frame_main.cxx
// This file provides the main function for driving the frame
// simulation program, as well as a movie generation function, and
// other necessary functions.

#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <string.h>
#include <time.h>
#include "frame.h"
#include "frame_param.h"
#include "twoDarray.h"
#include "gauss.h"
#include "queue.h"
#include "box.h"

void movie(const int numFrames, const FrameParam* frameParams,
   const BoxParam* box, const int printPixels)
  // This function will make a movie with numFrame frames with each
  // frame satisfying frameParams.

{
  int i;
  Queue averagingFrames;
  Node* refFrame;

  // Steps in making a movie:
  if (frameParams->m_avgT > 1)
    {
      // make avgT - 1 frames
      for (i = 0; i < frameParams->m_avgT - 1; i++)
{
  averagingFrames.enqueue(0, -1, frameParams);
  averagingFrames.tailPtr->frame->initFrame();
}
    }
  else // frameParams->m_avgT == 1
    {
      averagingFrames.enqueue(0, -1, frameParams);
      averagingFrames.tailPtr->frame->initFrame();
    }
```

```
// Now start the repetitions that will each generate a frame
refFrame = averagingFrames.tailPtr;
for (i = 0; i < numFrames; i++)
    {
        // display current frame number to stderr
        cerr << "Processing Frame Number : " << i << endl;
        // make current frame -- the tail of the queue
        averagingFrames.enqueue(i % 2, i, frameParams);
        // initialize current frame
        averagingFrames.tailPtr->frame->initFrame();
        // average current frame
    for (int r = 0; r < frameParams->m_frameRows; r++)
        for (int c = 0; c < frameParams->m_frameCols; c++)
            {
Node* nodePtr = averagingFrames.headPtr;
double tempPixelValue = 0;
for (int t = 0; t < frameParams->m_avgT; t++)
    {
        tempPixelValue += nodePtr->frame->blockSum(r,c);
        nodePtr = nodePtr->nextPtr;
    }
averagingFrames.tailPtr->frame->m_pixels[r][c] =
    tempPixelValue /
    sqrt( (double)frameParams->m_avgNN * frameParams->m_avgNN *
    frameParams->m_avgT );
            }
// Now add the box
    if (box != 0)
    if((i >= box->m_startTime  &&  (i < box->m_stopTime))
        if (box->m_direction) // horizontal motion
{
    averagingFrames.tailPtr->frame->addBox(box->m_boxH,
        box->m_boxW,
        box->m_startRow,
        box->m_startCol + (i - box->m_startTime)*box->m_stepVal,
        box->m_intensity);
}
        else // box->m_direction == 0, so veritcal motion
{
    averagingFrames.tailPtr->frame->addBox(box->m_boxH,
box->m_boxW,
box->m_startRow + (i - box->m_startTime)*box->m_stepVal,
box->m_startCol,
box->m_intensity);
}
    // find stats for current frame
    averagingFrames.tailPtr->frame->findStats(*(refFrame->frame));
    // write current frame to disk
    averagingFrames.tailPtr->frame->writeMpgData();
    // write pixel data to disk if printPixels == 1
    if (printPixels && i >= 0 && i <= 12)
averagingFrames.tailPtr->frame->writePixelData();
    // write actual motion vectors to disk if Pframe and collectMV != 0
    if ( ((i % 2) == 1)
  && averagingFrames.tailPtr->frame->getCollectMV() )
{
    averagingFrames.tailPtr->frame->writeMotionVectors();
}
    // make the current frame the reference frame
    refFrame = averagingFrames.headPtr;
    while (refFrame->nextPtr != 0)
refFrame = refFrame->nextPtr;
    // delete oldest frame
    averagingFrames.dequeue();
    }
// write the motion vector tally information to disk (if no
// information was collected, this array will be all zeros.
writeMVTally(frameParams);
}

void readFrameParameterFile(const char frameParameterFilename[], int& avgNN,
    int& avgT, int& numNN,
    double& lambda, char baseFrameFile[],
    int& collectMV)
// This function will read in the frame parameter information from
// the frame parameter file. This should be a text file containing
// the following information in this exact order:
//    averaging box height
//    averaging box width
//    number of frames to average back in time
//    number of macroblocks along a side of the frame (square
//    frames have been assumed)
//    lambda value for use in norm function
//    the base filename that frame data will be stored in
//    boolean to indicate whether or not to record actual motion vector data
// Space for the FrameParam data must be allocated outside of this
// function.
{
    ifstream inputFile;
    inputFile.open(frameParameterFilename, ios::in);
    if (inputFile == 0)
        {
            cerr << "Could not open frame parameter file." << endl;
            exit(1);
```

```
}
int tempInt; // temporary variables to read values into
double tempDouble;
char filename[100];
inputFile >> tempInt;
avgHH = tempInt;
inputFile >> tempInt;
avgHW = tempInt;
inputFile >> tempInt;
avgF = tempInt;
inputFile >> tempInt;
numMB = tempInt;
inputFile >> tempDouble;
lambda = tempDouble;
inputFile >> filename;
baseFrameFile = strcpy(baseFrameFile, filename);
inputFile >> tempInt;
collectHW = tempInt;

inputFile.close();

  cerr << "Frame parameters are: " << endl;
cerr << "avgHH " << avgHH << endl;
cerr << "avgHW " << avgHW << endl;
cerr << "avgF " << avgF << endl;
cerr << "numMB " << numMB << endl;
cerr << "lambda " << lambda << endl;
cerr << "baseFrameFile " << baseFrameFile << endl;
cerr << "collectHW " << collectHW << endl;
}

void readBoxInput(const char filename[], int boxH, int boxW, int startRow,
  int startCol, double& intensity, int stepVal,
  int direction,
  int startTime, int stopTime)
// This function reads in the data file that contains the
// information used to add a box to a movie. This file contains the
// following information in this exact order:
//   box height
//   box width
//   row of upper left hand corner to add the first box
//   column of the upper left hand corner to add the first box
//   value to add to each pixel in the box area
//   amount to shift each box each frame
//   direction : non-zero means horizontal, zero means verticle motion
//   first frame to add a box to
//   frame after the last frame to add a box to.
{
  ifstream boxInput;
  int tempInt;
  double tempDouble;

  boxInput.open(filename, ios::in);
  if (boxInput == 0)
    {
      cerr << "Could not open file " << filename << endl;
      exit(1);
    }

  boxInput >> tempInt;
  boxH = tempInt;
  boxInput >> tempInt;
  boxW = tempInt;
  boxInput >> tempInt;
  startRow = tempInt;
  boxInput >> tempInt;
  startCol = tempInt;
  boxInput >> tempDouble;
  intensity = tempDouble;
  boxInput >> tempInt;
  stepVal = tempInt;
  boxInput >> tempInt;
  direction = tempInt;
  boxInput >> tempInt;
  startTime = tempInt;
  boxInput >> tempInt;
  stopTime = tempInt;
  boxInput.close();

  cerr << "Box parameter information: " << endl;
  cerr << "boxW " << boxW << endl;
  cerr << "boxH " << boxH << endl;
  cerr << "startRow " << startRow << endl;
  cerr << "startCol " << startCol << endl;
  cerr << "intensity " << intensity << endl;
  cerr << "stepVal " << stepVal << endl;
  cerr << "direction " << direction << endl;
  cerr << "startTime " << startTime << endl;
  cerr << "stopTime " << stopTime << endl;
}

void readMovieInput(const char filename[], int& numFrames, int& printPixels)
// This function reads in the number of frames in the movie.
// This information is contained in the filename file, which is a
```

```
  // plain text file containing:
  //     the number frames to make in the movie.
  //     the printPixels boolean value; 0 indicates do not write pixel
  //     data to disk, non-zero means data will be written to disk
{
  ifstream movieInput;
  int tempInt;

  movieInput.open(filename, ios::in);
  if (movieInput == 0)
    {
      cerr << "Could not open movie parameter information file." <<
endl;
      exit(1);
    }
  movieInput >> tempInt;
  numFrames = tempInt;
  movieInput >> tempInt;
  printPixels = tempInt;
  movieInput.close();

  cerr << "Movie parameters  :" << endl;
  cerr << "numFrames " << numFrames << endl;
  cerr << "printPixels " << printPixels << endl;
}

int main()
{
  // initialize random seed;
  // Here we use one random number generator to seed the next.
  long int seed1 = time(NULL);
  cerr << "seed1 : " << seed1 << endl;
  srand48(seed1);
  long int seed2 = lrand48();
  cerr << "seed2 : " << seed2 << endl;
  srand(seed2);
  // write seed to file
  fstream seedFile;
  seedFile.open("randomSeed.out", ios::app);
  seedFile << seed2 << endl;
  seedFile.close();

  // initialize static members if Iframe class
  initDCTmatrix();
  // initialize static members of Pframe class
  initStaticMvTally();

  // Frame parameters
  int avgBH, avgBW, avgT, numMB, collectMV;
  double lambda;
  char baseFrameFile[100];
  int numFrames;
  int printPixels; // Non zero value indicates pixel data should be
                   // written to disk
  // Box addition parameters
  int boxH, boxW, startRow, startCol, stepVal, direction, startTime, stopTime;
  double intensity;

  //get frame parameter information from file
  readFrameParameterFile("frameInit.dat", avgBH, avgBW, avgT,
     numMB, lambda, baseFrameFile, collectMV);
  // define the frame parameter object
  FrameParam frameParameters(avgBH, avgBW, avgT, numMB, lambda,
     baseFrameFile, collectMV);
  //read the box addition information from file
  readBoxInput("boxInit.dat", boxH, boxW, startRow, startCol,
       intensity, stepVal, direction, startTime, stopTime);
  // create box parameter object
  BoxParam boxInfo(boxH, boxW, startRow, startCol, intensity,
   stepVal, direction, startTime, stopTime);

  // get movie information (ie number of frames) from file
  readMovieInput("movieInit.dat", numFrames, printPixels);

  // make the movie
  movie(numFrames, &frameParameters, &boxInfo, printPixels);
  cerr << "Program Finished.\a" << endl;
  return 0;
}
```

//

The following files are used to calculate the p-values from the frame data.

//

```
// This file defines a truncation function needed to create equality
// in the motion vector testing.

#ifndef TRUNCATE_H
#define TRUNCATE_H
#include <math.h>

inline static double truncate(double x)
```

```
{
  double temp;
  if (x < 10)
    {
      temp =  floor(x * 10000);
      return temp/10000;
    }
  else // x >= 10
    {
      temp = floor( x * 1000);
      return temp/1000;
    }
}
#endif TRUNCATE_H

////////////////////////////////////////////////

#ifndef HYPTEST_H
#define HYPTEST_H

class HypothesisTest
{
public:
  HypothesisTest(double dataType);
  double m_inputData;
  double m_pValue;
protected:
  virtual double binarySearch (double key) = 0; // a pure virtual function
  // It will find the pvalue for the frame by looking in the
  // appropriate look-up table.
};

class DCTtest : public HypothesisTest
{
friend void initializeStaticMembersDCT(const char filename[]);
public:
  DCTtest(double input);
private:
  virtual double binarySearch(double key);
  static double dctTable[5000];
};

class mvTest : public HypothesisTest
{
friend void initializeStaticMembersMV(const char filename[]);
public:
  mvTest(double input);
private:
  virtual double binarySearch(double key);
  static double mvTable[2][135];
};
#endif

////////////////////////////////////////////////

// This file defines a truncation function needed to create equality
// in the motion vector testing.

#include "truncate.h"

inline static double truncate(double x)
{
  int temp;
  if (x < 10)
    {
      temp = (int) x * 100000;
      return (double)temp/100000;
    }
  else // x >= 10
    {
      temp = (int) x * 10000;
      return (double)temp/10000;
    }
}

////////////////////////////////////////////////

// This file defines all the member function of the HypothesisTest class

#include <fstream.h>
#include <stdlib.h>
#include "hyptest.h"
#include "truncate.h"

HypothesisTest::HypothesisTest(double input)
{
  m_inputData = input;
  m_pValue = -1; // nonsense value; will be reset
}

// initialize static members
double DCTtest::dctTable[5000];

DCTtest::DCTtest(double input) : HypothesisTest(input)
```

```
{
  m_pValue = binarySearch(m_inputData);
}

double DCTest::binarySearch(double key)
{
  int low = 0;
  int high = 4999;
  int middle;

  if (key < dctTable[0])
    return 1;
  if (key > dctTable[4999])
    return 0;

  while (low <= high)
    {
      middle = (low + high) / 2;
      if ( key >= dctTable[middle]  &&  key < dctTable[middle + 1])
return 1 - middle/(double)5000; // probability of being
        // greater than or equal to this observed  value

      if (key >= dctTable[middle + 1])
low = middle + 1;
      else // key < dctTable[middle]
high = middle - 1;
    }
  // all cases should have been covered by now
  // will exit program if this is not the case
  cerr << "Problem with function binarySearch in dct class." << endl;
  cerr << "No match was found." << endl;
  exit(1);
}

void initializeStaticMemberDCT(const char filename[])
  // This function reads in the table of (previously simulated) data
  // that is used to numerically estimate the p-values of the DCT
  // statistic values.
{
  ifstream tableFile;
  tableFile.open(filename, ios::in);
  if (tableFile == 0)
    {
      cerr << "Could not open DCT table file" << endl;
      exit(1);
    }
  int i = 0;
  while(tableFile >> DCTest::dctTable[i])
    i++; // should read in data to the end
    // of the file

  tableFile.close();
}

// function definitions for mvTest class

// declare static member data
double  mvTest::mvTable[2][136];

mvTest::mvTest(double input) : HypothesisTest (input)
{
  m_pValue =  binarySearch(m_inputData);
}

double mvTest::binarySearch(double key)
  // This function will return the pvalue for the mv hypothesis
  // test. Note that the p-value is the probability of observing a
  // particular value, or anything more extreme.
{
  int low = 0;
  int high = 134;
  int middle;

  while (low <= high)
    {
      middle = (low + high) / 2;

      if (key == mvTable[0][middle])
    {
  if (middle == 0)
    return 1; // probability of observing mv length == 0 or
  // something bigger is 1
  else
    return 1 -  mvTable[1][middle - 1];
  // Note: P[observing a mv length >= observed length]
  //  = 1 - P[ length < observed]
  //  = 1 - P[ length <= next smallest length]
  }
      if (key > mvTable[0][middle])
low = middle + 1;
        else // key < mvTable[0][middle]
high = middle - 1;
    }
```

```
  // all cases should have been covered by now
  // will exit program if this is not the case
  cerr << "Problem with function binarySearch in mv class." << endl;
  cerr << "No match was found." << endl;
  exit(1);
}

void initializeStaticMembersMV(const char filename[])
{
  ifstream tableFile;
  tableFile.open(filename, ios::in);
  if (tableFile == 0)
    {
      cerr << "Could not open mv table file" << endl;
      exit(1);
    }

  int i = 0;
  while (tableFile >> mvTest::mvTable[0][i] >> mvTest::mvTable[1][i])
    i++;
  // this should read in the data to
  // the end of the file
  tableFile.close();
  // Some rounding of the data needs to be done to get it to match the
  // output from the movie program.
  for (i = 0; i < 135; i++)
    mvTest::mvTable[0][i] = truncate(mvTest::mvTable[0][i]);
}

//////////////////////////////////////////////////////

// calcPvalue.cxx
/* This program will read in the frame data -- the motion vectors or
   the DCT coefficients -- and then calculate the p-values of this
   data. */
// The main function is found here.

#include <stdlib.h>
#include <fstream.h>
#include <string.h>
#include <stdio.h>
#include "hyptest.h"
#include "truncate.h"

void findPvalues(char inputFilename[], char outputFilename[], const
 int numFrames)
  // This function finds the p-values for data stored in
  // inputFilename and writes the results to outputFilename.

  // April 24, 2001. New Feature: separating the I- and P-frames. If
  // the separateFrames variable is set to true (non-zero) then the
  // output pvalues for I and P frames will be stored in separate
  // files. This value must be set in the code; as of yet it can't be
  // changed at compile time.

{
  ifstream frameFile;
  ofstream pvalueFile;

  int separateFrames = 1;
  ofstream IframeFile;
  ofstream PframeFile;
  char IframeFileName[100] = "";
  char PframeFileName[100] = "";

  char actualFilename[] = "";
  double data;
  DCTtest* DCTPtr;
  mvTest* mvPtr;

  if (separateFrames == 0)
    {
      pvalueFile.open(outputFilename, ios::out);
      if (pvalueFile == 0)
{
  cerr << "Could not open pvalueOutputFilename file for write" << endl;
  exit(1);
}
    }
  else // separateFrame is true
    {
      snprintf(IframeFileName, 1 + strlen(outputFilename) + 1, "%s%s",
       outputFilename, "I");
      snprintf(PframeFileName, 1 + strlen(outputFilename) + 1, "%s%s",
       outputFilename, "P");

      IframeFile.open(IframeFileName, ios::out);
      if (IframeFile == 0)
 {
  cerr << "Could not open IframeFileName for write " << endl;
  exit(1);
 }
      PframeFile.open(PframeFileName, ios::out);
      if (PframeFile == 0)
```

```
{
   cerr << "Could not open iframeFileName for write " << endl;
   exit(1);
}
   }

   for (int i = 0; i < numFrames; i++)
      {
         sprintf(actualFilename, 8 + strlen(inputFilename) + 1, "%s%d",
            inputFilename, i);

         frameFile.open(actualFilename, ios::in);
         if (frameFile == 0)
{
   cerr << "Could not open for read the file " << actualFilename;
   cerr << endl;
   exit(1);
}

         while ( frameFile >> data )
{
         if (i % 2 == 0) // then frame is an I frame
            {
               DCTPtr = new DCTtest(data);
               if (separateFrames)
IframeFile << (DCTPtr->m_pValue) << endl;
               else
pvalueFile <<  (DCTPtr->m_pValue) << endl;
               delete DCTPtr;
            }
         else // frame must be P frame
            {
               mvPtr = new mvTest(truncate(data)); // truncation is
               // necessary to obtain a match in the mv look up tables
               // for calculating the p-value
               if (separateFrames)
PframeFile << mvPtr->m_pValue << endl;
               else
pvalueFile <<  mvPtr->m_pValue << endl;
               delete mvPtr;
            }
      }
         frameFile.close();
         }
      IframeFile.close();
      PframeFile.close();
}

void readInPvalueData(char filename[], char inputFilename[],
        int& numFrames, char dctFilename[],
        char mvFilename[], char outputFilename[])
// This function read in initialization file. This file is called
// "calcPinit.dat" and contains the following
// information, in this order, one piece of information per line
//    path and base filename of frame data
//    number of frames to read in
//    filename of dct table data
//    filename of mv table data
//    the name to call the pvalue output file
{
char tempFilename[100]; // holders for reading in the values
int tempInt;

ifstream initialValuesFile;
initialValuesFile.open(filename, ios::in);
if ( initialValuesFile == 0)
   {
      cerr << filename << " file could not be opened";
      cerr << endl;
      exit(1);
   }
initialValuesFile.width(99);
initialValuesFile >> tempFilename;
strcpy(inputFilename, tempFilename);
initialValuesFile >> tempInt;
numFrames = tempInt;
initialValuesFile.width(99);
initialValuesFile >> tempFilename;
strcpy(dctFilename, tempFilename);
initialValuesFile.width(99);
initialValuesFile >> tempFilename;
strcpy(mvFilename, tempFilename);
initialValuesFile.width(99);
initialValuesFile >> tempFilename;
strcpy(outputFilename, tempFilename);
initialValuesFile.close();

cerr << "Parameters for calculating p-values:" << endl;
cerr << "Input filename " << inputFilename << endl;
cerr << "numFrames " << numFrames << endl;
cerr << "dctFilename " << dctFilename << endl;
cerr << "mvFilename " << mvFilename << endl;
cerr << "Output Filename " << outputFilename << endl;
}
```

```
int main()
{
  char frameFilename[100];
  int numFrames;
  char dctTableFilename[100];
  char mvTableFilename[100];
  char pvalueOutputFilename[100];
  cerr << "Program to find p-values of mv and DCTcoeff" << endl;

  readInPvalueData("calcPInit.dat", frameFilename, numFrames,
   dctTableFilename, mvTableFilename,
   pvalueOutputFilename);

  initializeStaticMembersDCT(dctTableFilename);
  initializeStaticMembersMV(mvTableFilename);

  findPvalues(frameFilename, pvalueOutputFilename, numFrames);
}

////////////////////////////////////////////////
```

The following files are used to perform the cusum.

```
////////////////////////////////////////////////

// cusum.h
/* This is the header file for the cusum.cxx program. It defines the
   cusum class which provides a method for implementing the cusum
   procedure suggested by McDonald. */

#ifndef CUSUM_H
#define CUSUM_H

#include <fstream.h>

class Cusum
{
public:
  Cusum( char filename[], char pvalueFilename[], const double delta,
 const double h);
  ~Cusum( void );

private:
  double cusumIncrement(void);
  const char* m_filename;
  const char* m_pvalueFilename;
  const double m_h;
  const double m_delta;
  const double m_k; // a function of delta
  double m_cusum;
  ifstream m_pvalueFile;
  ofstream m_cusumFile;
};

#endif CUSUM_H

////////////////////////////////////////////////

// This file contains the defintions of the member funcation of cusum.h

# include <math.h>
# include <iostream.h>
# include <stdlib.h>
# include "max.h"
# include "cusum.h"

Cusum::Cusum(char filename[], char pvalueFilename[], const double delta,
     const double h)
  : m_filename(filename),
    m_pvalueFilename(pvalueFilename),
    m_h(h), m_delta(delta),
    m_k(-1/delta * log(delta / (exp(delta) - 1)))
{

  m_cusum = 0; // initialized cusum statistic
  // opens the pvalue file so it can be read by the cusumIncrement
  // function
  m_pvalueFile.open(m_pvalueFilename, ios::in);
  if (!m_pvalueFile)
    {
      cerr << m_pvalueFilename << " pvalueFile file could not be opened." ;
      cerr << endl;
      exit(1);
    }

  // opens the file to store the cusum values
  m_cusumFile.open(m_filename, ios::out);
  if (!m_cusumFile)
    {
      cerr << m_filename << "cusumFile file could not be opened." << endl;
```

```
    exit (1);
  }

 // starts the cusum
 while ( !m_pvaluesFile.eof() )
  {
    m_cusum = max( m_cusum + cusumIncrement(), 0 );
    // write m_cusum to file
    m_cusumFile << m_cusum << endl;
    // reset cusum to zero if an alarm has sounded
    if (m_cusum > h)
m_cusum = 0;
  }
 m_cusumFile.close();
 m_pvalueFile.close();
}

Cusum::~Cusum(void)
{
}

double Cusum::cusumIncrement(void)
  /* This function calculates the amount of the increment for the
     cusum statistic using the current p-value. */
{
  double pvalue;
  m_pvalueFile >> pvalue;
  return (1 - pvalue) - m_k;
  // IMPORTANT!! We need to take (1 - pvalue) because we want to test
  // again a change in pvalues that are becoming smaller (which would
  // lead to rejection). But, our hypothesis test will test against
  // pvalues that are changing to larger values.
}

//////////////////////////////////////////////////////

// Here is the driver file for the cusum procedure.

# include <math.h>
# include <iostream.h>
# include <stdlib.h>
# include "max.h"
# include "cusum.h"

void readInCusumData(char cusumF[], char pvalueF[], double& delta,
     double& h)
  // Here we open the file that contains the initial values for the
  // cusum. This file is called "cusumInit.dat" and contains
  // the following information, in this order, one datum per line:
  //     pvalue file name (where data read from)
  //     cusum file name (where data to be stored)
  //     delta value
  //     h value (the cutoff value)
{
  ifstream initialValuesFile;
  char tempFile[100];
  double tempDouble;

  initialValuesFile.open("cusumInit.dat", ios::in);
  if (!initialValuesFile)
   {
     cerr << "cusumInit.dat file could not be opened" << endl;
     exit(1);
   }
  initialValuesFile.width(99);
  initialValuesFile >> tempFile;
  strcpy(pvalueF, tempFile);
  initialValuesFile.width(99);
  initialValuesFile >> tempFile;
  strcpy(cusumF, tempFile);
  initialValuesFile >> tempDouble;
  delta = tempDouble;
  initialValuesFile >> tempDouble;
  h = tempDouble;
  initialValuesFile.close();

  cerr << "Parameter values in cusum: " << endl;
  cerr << "Input filename " << pvalueF << endl;
  cerr << "Output filename " << cusumF << endl;;
  cerr << "delta " <<delta << endl;
  cerr << "h " << h << endl;
}


int main()
{
  char pvalueFilename[100];
  char cusumFilename[100];
  double delta;
  double h;

  cerr << "Cusum of p-values Program" << endl;

  readInCusumData(cusumFilename, pvalueFilename, delta, h);
```

```
  // creating an instance of the Cusum class gets everything going
  Cusum mycusum(cusumFilename, pvalueFilename, delta, h);

  cerr << "End of program. " << endl;
  return 0;
}
```

//

The following files were used to generate dependent p-values.

//

```
// This is the header file for the uniform but dependent p-values.
#ifndef DEPENDENT_H
#define DEPENDENT_H
#include <math.h>

class DependentSequence
{
public:
  DependentSequence(int size, double initSeed);
  DependentSequence(int size, unsigned long int initSeed,
    unsigned long int precision);
  ~DependentSequence(void);

private:
  int m_size;
  double* m_sequence;
  unsigned long int* m_intSequence;
  unsigned long int m_precision;
  int m_period;

  double T(double x) const;
  unsigned long int T(unsigned long int x) const;
  void setDimension(int size);
  void writeSequenceToDisk(void);
};
#endif DEPENDENT_H
```

//

```
// This file defines the functions in the dependent.h header file

#include <assert.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include "dependent.h"

DependentSequence::DependentSequence(int size, double initSeed)
{
  setDimension(size);
  m_sequence[0] = initSeed;
  for (int i = 1; i < size; i++)
    {
      m_sequence[i] = T(m_sequence[i-1]);
    }
  writeSequenceToDisk();
}

DependentSequence::DependentSequence(int size, unsigned long int initSeed,
     unsigned long int precision)
{
  int warmUp = 3000;
  unsigned long int match;

  setDimension(size);
  m_precision = 1;
  for (unsigned int i = 0; i < precision; i++)
    m_precision = m_precision * 10;
  m_intSequence[0] = initSeed;
  m_sequence[0] = (double)initSeed /  m_precision;
  for (int i = 1; i < warmUp; i++)
    {
      m_intSequence[i] = T(m_intSequence[i-1]);
      m_sequence[i] = (double)m_intSequence[i] / m_precision;
    }
  match = m_intSequence[warmUp - 1];
  for (int i = warmUp; i < m_size; i++)
    {
      m_intSequence[i] = T(m_intSequence[i-1]);
      m_sequence[i] = (double)m_intSequence[i] / m_precision;
      if (match == m_intSequence[i])
m_period = i - warmUp;
    }
  writeSequenceToDisk();
}


DependentSequence::~DependentSequence(void)
{
```

```
  if (m_sequence)
    delete m_sequence;
}

void DependentSequence::setDimension(int size)
{
  assert (size >= 0);

  if (m_sequence)
    delete m_sequence;
  if (m_intSequence)
    delete m_intSequence;

  m_size = size;
  if (m_size > 0)
    {
      m_sequence = new double[m_size];
      m_intSequence = new unsigned long int[m_size];
    }
  else // size == 0
    {
      m_sequence = 0;
      m_intSequence = 0;
    }
}

double DependentSequence::T(double x) const
{
  double temp;
  temp = 1 - fabs(2*x - 1);
  /*
  while (temp == 0 || temp == 1)
    temp = (double)rand() / (RAND_MAX + 1.0);
  */
  return temp;
}

unsigned long int DependentSequence::T(unsigned long int x) const
{
  unsigned long int temp;
  temp = 1*m_precision - abs(2*x - 1*m_precision);
  return temp;
}


void DependentSequence::writeSequenceToDisk(void)
{
  ofstream outFile;
  outFile.open("dependentPvalues.out",ios::out);
  if (outFile == 0)
    {
      cerr << "Problems opening dependentPvalues.out for writing" << endl;
      exit(1);
    }
  cerr << "Period of sequence : " << m_period << endl;
  //  outFile << m_period << endl;
  for(int i = 0; i < m_size; i++)
    {
      outFile << m_sequence[i] << endl;
      // outFile << m_sequence[i] << "\t" << m_intSequence[i] << endl;
    }
  outFile.close();
}

////////////////////////////////////////////////

// This program will create a sequence of numbers between 0 and 1
// which has at each point in time a marginal distribution of being
// uniform on (0,1), but are still dependent.

#include "dependent.h"
#include <time.h>
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <fstream.h>

int main()
{
  const int numValues = 10000;

  // seed the random number generator
  long int seedA = time(NULL);
  unsigned short int xsubi[3];
  xsubi[0] = seedA % 10;
  xsubi[1] = (int)((double)seedA / 10) % 10;
  xsubi[2] = (int)((double)seedA / 100) % 10;
  long int seed = jrand48(xsubi);

  // write seed to file
  fstream seedFile;
  seedFile.open("randomSeed.out", ios::app);
  seedFile << seedA << "\t" << seed << endl;
  seedFile.close();
```

```
srand48(seed);
unsigned int intInitSeed = lrand48();
cerr << "Random Seed :" << intInitSeed << endl;
// create the dependent sequence; it will be generated and written
// to disk, in the file "depedentPvalues.out"
DependentSequence pvalues(numValues, intInitSeed, 14);
}

//////////////////////////////////////////////////
```

Bibliography

[1] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures.* Kluwer Academic Publishers, 1995.

[2] Patrick Billingsley. *Probability and Measure.* John Wiley and Sons, Third edition, 1995.

[3] Christopher B. Burckhardt. Speckle in ultrasound B-mode scans. *IEEE Transactions on Sonics and Ultrasonics*, SU-25(1):1–6, January 1978.

[4] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* John Wiley and Sons, 1991.

[5] H. M. Deitel and P. J. Deitel. *C++ How to Program.* Prentice Hall, 1994.

[6] Manfred Denker and Gerhard Keller. Rigorous statistical procedures for data from dynamical systems. *Journal of Statistical Physics*, 44(1/2):67–93, 1986.

[7] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video : An Introduction to MPEG-2.* Chapman and Hall, 1997.

[8] International Electrotechnical Commission International Standards Organization. ISO/IEC 11172–Information technology–Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, August 1993.

[9] G. Lorden. Procedures for reacting to a change in distribution. *The Annals of Mathematical Statistics*, 42(6):1897–1908, 1971.

[10] Patrick Marchand. *Graphics and GUIs with Matlab.* CRC Press, Second edition, 1999.

[11] David McDonald. A cusum procedure based on sequential ranks. In *Naval Research Logistics*, volume 37, pages 627–646. John Wiley and Sons, Inc., 1990.

[12] David Middleton. *An Introduction to Statistical Communication Theory.* McGraw-Hill Book Company, Inc., 1960.

[13] R.C. Molthen, P.M. Shankar, and J.M. Reid. Characterization of ultrasonic B-scans using non-rayleigh statistics. *Ultrasound in Medicine and Biology*, 21(2):161–170, 1995.

[14] George V. Moustakides. Optimal stopping times for detecting changes in distributions. *The Annals of Statistics*, 14(4):1379–1387, 1986.

[15] MPEG Home Page. http://www.cselt.it/mpeg/. The official home page of the Motion Pictures Experts Group, the working committee of the ISO/IEC.

[16] E.S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, June 1954.

[17] Northern Ireland Physics Teachers Panel. Northern Ireland Educational Support Unit Tutorial in Medical Imaging for A-Level Physics. http://www.qub.ac.uk/edu/niesu/physics/Medical_Imaging/index.htm.

[18] Emmanuel P. Papadakis, editor. *Ultrasonic Instruments and Devices: Reference for Modern Instrumentation, Techniques, and Technology.* Academic Press, 1999.

[19] K.R. Rao and P. Yip. *Discrete Cosine Transform: Algorithm, Advantages, Applications.* Academic Press, 1990.

[20] Sheldon Ross. *Applied Probability Models With Optimization Applications.* Holden-Day, 1970.

[21] S.W. Smith, H. Lopez, and Jr. W.J. Bodine. Frequency independent ultrasound contrast-detail analysis. *Ultrasound in Medicine and Biology*, 11(3):467–477, 1985.

[22] Peter D. Symes. *Video Compression.* McGraw-Hill, 1998.

[23] Robert F. Wagner, Stephen W. Smith, John M. Sandrik, and Hector Lopez. Statistics of speckle in ultrasound B-scans. *IEEE Transactions on Sonics and Ultrasonics*, 30(3):156–163, May 1983.

[24] John Watkinson. *MPEG-2.* Focal Press, 1999.