1993

Image compression using vector quantization and lossless index coding

Mark Dee Hetherington
Iowa State University

Image compression using vector quantization and lossless index coding

by

Mark Dee Hetherington

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Electrical Engineering

Iowa State University
Ames, Iowa

1993

To my wife Susan, whose patience
and support has been invaluable

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

The purpose of data compression is to represent data as efficiently as possible without loss of important information. This generally involves the identification and removal of unnecessary or redundant information contained in the data. Most data are represented in such a way as to make them easily understandable or easy to process. This, however, generally results in significant data redundancy.

The need for data reduction arises whenever there is a limitation on space or time. Although advances in hardware storage capacities and transmission speeds have been made in recent years, the need for data compression has also steadily increased. The demand for storage capacity and transmission speed always seems to be a step ahead of the current technology. Storage devices that would have been viewed as gigantic only a few years ago seem quite restrictive today. There is no evidence that this trend will be soon to pass. With multimedia applications gaining popularity, the demand for transmission speed and storage space is increasing rapidly. The multimedia areas of audio, speech, text, images and video all rely on the availability of good data compression schemes. A good example of this need is full motion digitized video which requires approximately 28 megabytes of data every second. Also, the entertainment industry has recently been interested in the technology of "video-on-demand" as an alternative to the rental of videotapes. Such a system would require the transmission and storage of hundreds of gigabytes of data for a two hour movie. Data compression is required to avoid the need for expensive high speed transmission networks. In general, data compression allows us the maximal use of the resources currently available.

When evaluating the performance of a data compression scheme, there are a number of important factors to consider. The most obvious factor is of course the amount of compression provided by the method. Some applications require very large compression factors while others may be less demanding. Another important factor is the speed of the compression and decompression. In situations where the data is being compressed for faster transmission, it is important that the time required to compress and decompress the data does not greatly reduce the benefits of data compression. Also, in applications such as audio, speech, and video, the compression and especially the decompression must be performed in real time so fast schemes are required. The memory requirement of a compression approach is also very important since large storage hardware is not desirable.

Another factor that must be considered is the use of lossless or lossy data compression methods. In lossless compression, there is a need for the reproduction of the data to be exact. For example, in most computer applications, such as computer software programs or text documents, any amount of error introduced by compression could prove to be disastrous. Ideally, data would always be represented without error, however, only modest amounts of compression are generally achievable with these methods. In applications where there is a need for higher amounts of compression and the interpretation of the data is performed by the human perceptual system, some degree of loss may be tolerable. The advantage of lossy representations is that they generally achieve compressions many times greater than lossless methods. In many applications, a combination of lossy and lossless methods is used in order to achieve the best possible compression performance.

One type of lossy compression method that has gained widespread attention in recent years is *vector quantization* (VQ). VQ has shown the ability to achieve very high compression ratios while maintaining good subjective quality. VQ provides large amounts of compression by removal of redundancies due to statistical dependencies that generally exist between samples. One advantage of VQ is that the decoder is very easy to implement making it attractive for applications where the data are compressed once but decompressed many times, such as archival of medical images. This is also attractive for producing inexpensive decoding hardware for commercial applications. For example, when data are compressed once and transmitted to many users, inexpensive decoding hardware is desired.

VQ provides the best possible performance of any block coding technique for a given block size [1]. However, VQ is often limited to very small block sizes due to storage and computational complexities. The performance of VQ is directly related to the block size used, so this limitation in block size affects the achievable performance. Much of the research being done in the area of VQ is related to the task of reducing the computational burden so that larger block sizes can be used.

Just as the interest in VQ has grown in recent years, interest in the lossless compression technique of *arithmetic coding* has also increased. Although the underlying concepts of arithmetic coding have been known since the late 40's, practical implementations of the technique were not discovered until the mid 70's. Refinements were made in the late 70's to allow efficient implementations, however, the technique still has not seen widespread popularity.

As with most lossless techniques, arithmetic coding removes redundancies due to non-uniform distribution of the message symbols. The basic idea behind all lossless techniques is to assign a variable number of bits to symbols depending on their probabilities of occurrence. Symbols that occur more frequently are assigned a fewer number of bits while less probable symbols are assigned more bits. This has the overall effect of reducing the average data rate.

Arithmetic coding has several advantages over other popular lossless techniques such as Huffman coding. Arithmetic coding is capable of providing near optimal performance since it does not require blocking of the data stream. Shannon's noiseless coding theorem implies that each data symbol can, in theory, be represented by a code length equal to its information content, which is defined as a logarithmic function of the symbol probability. Huffman codes use blocking of the data stream which means that each symbol must be mapped to an integral number of bits. That is, they operate on a symbol by symbol basis. Since, in general, the information content of a data symbol is non-integral, Huffman codes suffer an inefficiency in bit allocations. In fact, Huffman codes can require in the worst case up to an extra bit for each symbol coded. Arithmetic coding, on the other hand, does not suffer from this inefficiency since no data blocking is necessary. This allows fractional bit allocations so each bit in the arithmetic code is fully utilized.

Arithmetic coding is also easily adaptable to the statistics of the data source so coding can be accomplished without previous knowledge of source statistics. As long as the decoder can determine how the encoder arrived at the statistics, it can easily decode the message. Huffman codes, on the other hand, require that a new set of codes be generated every time the symbol probabilities change. This must be done at both the encoder and the decoder which is generally too time consuming for practical implementations.

The use of arithmetic coding also allows the separation of coding and *source modeling*. Source modeling, as the name implies, attempts to model, in a statistical sense, the way in which an information source produces data. By appropriately modeling the source, the compression obtained by the arithmetic coder can be improved by reducing the uncertainty associated with the data. In effect, the goal of the model is to predict future data from past data. The better a model is at predicting the statistics of the source, the less uncertainty exists in its data. The achievable compression of any model is directly related to the degree of uncertainty associated with it, so better model predictions mean better compression performance.

Since arithmetic coding performs near optimally, much of the performance improvement in lossless compression comes from the design of better source models. In general, the more information provided to a model, the better the model will perform. In effect, source modeling removes redundancies due to correlations between source symbols. More information about how the information source operates allows the model to exploit more of the redundancy present in the data.

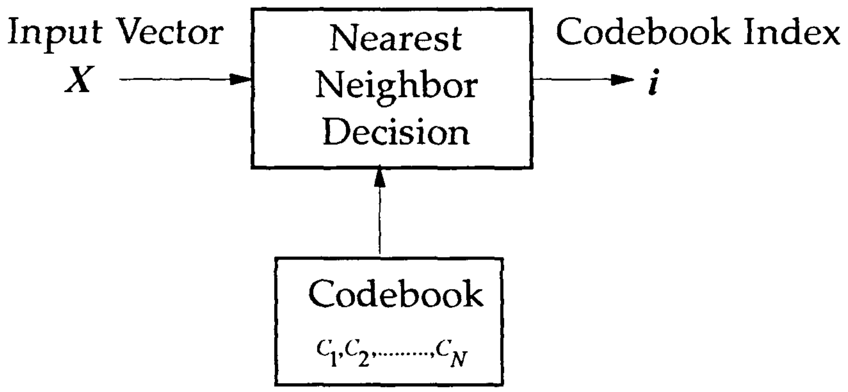
The compression techniques implemented in this thesis involve a combination of both lossless and lossy compression methods. Chapter 2 is concerned with the discussion of the concepts behind vector quantization, including the advantages, disadvantages, and popular alternatives to the standard technique. In Chapter 3, a brief introduction to the ideas behind lossless coding is presented along with a discussion of the arithmetic coding technique. In Chapter 4, a discussion of basic source modeling concepts for use with the arithmetic coder is presented. Chapter 5 describes the overall compression approach as applied to the compression of digital images. Chapter 6 provides the results obtained by implementing the techniques described in Chapter 5. A comparison of the results to other methods is also provided. Finally, Chapter 7 summarizes the conclusions and proposes possible directions for future work in this area.

CHAPTER 2. VECTOR QUANTIZATION

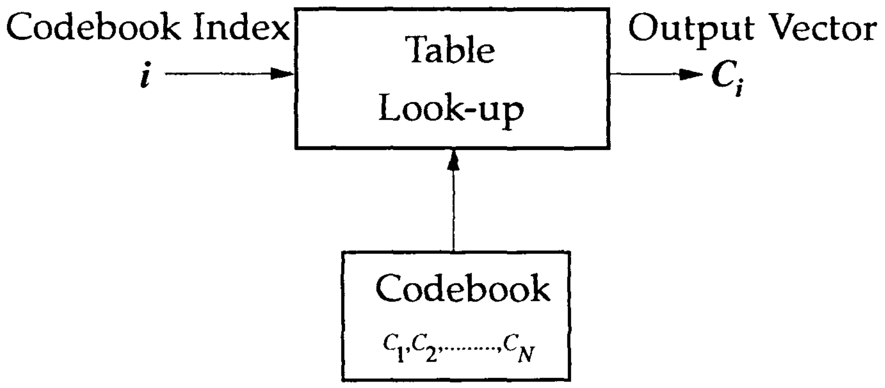
It is a well known fact that in most real world signals there is a large degree of correlation between adjacent samples. Coding systems that can exploit these correlations will in general provide good compression performance. Standard scalar quantization does not take advantage of such correlations since each sample is coded independently. Scalar coding techniques that employ memory, such as differential pulse code modulation (DPCM), are able to exploit these correlations to a degree. Vector quantizers are capable of exploiting these inter-sample correlations to a greater degree by coding groups of samples simultaneously.

The basic encoding and decoding structures for vector quantization (VQ) are shown in Figure 2.1. In VQ, a group of k samples from a signal is considered as a single block or vector, X, which is compared to a collection of N specially designed template vectors called a *codebook*. The individual vectors in the codebook are generally referred to as *codewords* or *codevectors* and each codevector is identified by a unique label or index. A comparison of the signal vector to each of the codevectors is performed with respect to a specified distance measure, usually the average squared error, and the signal vector is quantized to the codevector that is closest to it. This particular codevector is commonly referred to as the *nearest neighbor* of the signal vector. The label that corresponds to this codevector is then transmitted or stored depending on the application. The decoder can then read the index and perform a table look-up using the same codebook to determine the quantized output. Thus, the encoder maps a k dimensional input vector onto one of N possible indices and the decoder maps this index back to a k dimensional approximation to the input vector.

In order to illustrate how VQ can exploit the statistical dependencies within a block of samples, a comparison of how the vector space is partitioned for scalar and vector quantization is considered.



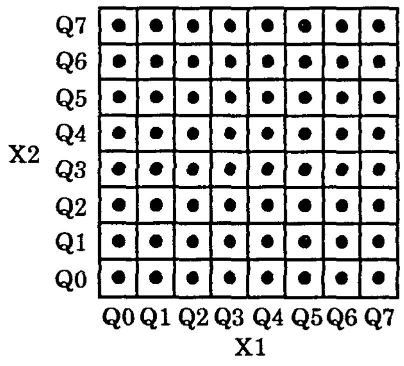
a) VQ encoder



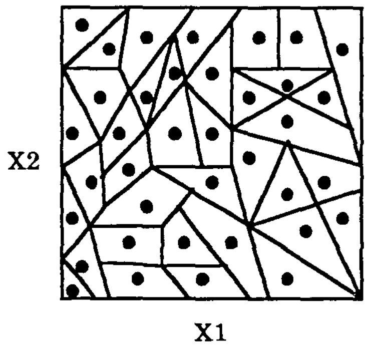
b) VQ decoder

Figure 2.1 Basic encoding and decoding structure for VQ

Examples of possible two dimensional vector space partitions for scalar and vector quantization are shown in Figure 2.2, where X1 and X2 represent the components of the two dimensional vector. The dot within each region represents the quantization point for that partition. In the



a) Scalar quantization



b) Vector quantization

Figure 2.2 Two dimensional vector space partitions for scalar and vector quantization

case of scalar quantization, the samples X1 and X2 are quantized independently but are being observed together in order to make a comparison with the VQ vector space. In this case, it is apparent that the scalar quantizer partitions the vector space in a very structured way. The quantization of X1 is completely independent of the value of X2 and vice versa so statistical dependencies between X1 and X2 cannot be exploited.

The vector quantizer, on the other hand, has more freedom to choose partition shapes and sizes, so that dependencies between samples can be exploited [1]. In the VQ vector space partition, shown in Figure 2.2, the quantization levels of X1 and X2 are now inter-related. This can be shown by holding the value of X1 fixed while varying X2. Partition regions that are intersected while X2 is varied from one end of the vector space to the other represent possible quantization points for the two dimensional vector. It is then easy to see that the quantized value of X1 will change for different values of X2.

Even if the samples within a vector are linearly independent, VQ can perform better than scalar quantization since it has the freedom to choose partition shapes that will more efficiently span the vector space. For example, VQ could partition the vector space with a hexagonal lattice structure as shown in Figure 2.3. This structure would reduce the distortion for the same number of partitions since the worst case distortion for each cell has been reduced.

In addition to the freedom to choose better partition shapes, VQ has the freedom to choose vectors that minimize the quantization error over the entire block of samples. That is, VQ will allow larger distortions for some vector components in exchange for less overall

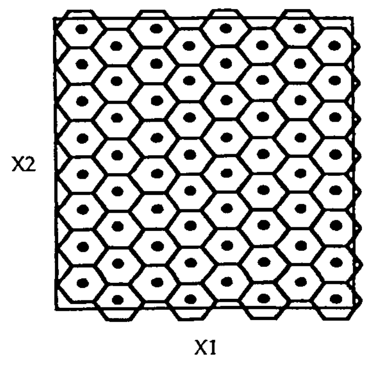


Figure 2.3 Vector space partition for hexagonal lattice

distortion in the vector. This is in contrast to scalar quantization where the distortion is minimized for each individual sample. This can lead to better subjective quality since distortions in individual samples may not be as perceptually important as the overall distortion of the vector. For example, in image coding, isolated pixels with larger distortions will generally not be objectionable to the eye. Similarly, in speech and audio coding, isolated samples with larger distortions will generally be inaudible.

Design

The design of a VQ coding system involves an iterative process of optimizing the encoder for a fixed decoder and optimizing the decoder for a fixed encoder. Given a set of training vectors, it is desired to find a set of codevectors that minimizes the distortion over the entire training set. For a fixed decoder or codebook, the encoder that minimizes the average distortion is simply a nearest neighbor encoder. In other words, the encoder maps the input vector, X, to the index i if and only if

$$d(X,C_i) \leq d(X,C_j) \qquad \text{for all } j$$

where $d(X,C_i)$ is the distance between X and the i^{th} codevector, C_i. For a fixed encoder mapping, the codebook that minimizes the average distortion over the training set contains codevectors that are the centriods of each partition. For example, let R_i be the set of all training vectors that are mapped by the encoder to index i. The codevector that minimizes the average distortion for this partition is simply the centroid of R_i. For the average squared error distance measure given by

$$d(X,C_j) = \|X - C_j\|^2 = \sum_{i=1}^{k}(x_i - c_{j,i})^2 \qquad \text{Equation 2.1}$$

the centroid is simply the statistical average of all the vectors belonging to R_i.

This process leads to the well known iterative improvement algorithm known as the LBG or k-means algorithm shown in Figure 2.4.

```
LBG Algorithm

1. Choose an initial codebook.
2. Optimize Encoder:
     Classify training data using the
     current codebook and the nearest
     neighbor decision rule.
3. Optimize Decoder:
     For each class, find the centriod of
     all the training vectors belonging to
     the class and use this as the new
     codevector.
4. Repeat 2 and 3 with the new
   codebook until convergence.
```

Figure 2.4 LBG algorithm

The first step in this algorithm is to choose an initial codebook, which is usually selected randomly from the training data. This codebook is then improved by first finding the optimal encoder for this codebook. This is simply a classification of the training data with respect to the nearest neighbor decision rule. This step is followed by the task of finding the optimal codebook for the encoder classification. The optimal codevector representing each class is determined by simply finding the centroid of the training vectors belonging to that class. The optimal codebook is then the collection of all of these optimal codevectors. These two steps are then repeated with the new codebook. This process continues until the change in average distortion becomes sufficiently small. The algorithm is guaranteed to converge to a locally optimal solution, however, there is no guarantee that the solution will be globally optimal.

To ensure that a good codebook design is obtained, the algorithm is generally run several times with different initial codebooks.

Complexity

The major drawback to the use of VQ is its computational and storage complexity. When an input vector is coded by a vector quantizer, it must be compared to each codevector in the codebook in order to find the nearest neighbor codevector. Obviously if the codebook size and the cost of these vector comparisons is large, the computation required to code each input vector may be too large for practical use.

The most commonly used distortion measure in VQ is the squared error given by Equation 2.1, where X is the input vector, C_j is the j^{th} codevector in the codebook, and k is the vector dimension. In this case, k multiplications are performed for each codevector. For an exhaustive search of a codebook of size N, kN multiplications are required which results in N multiplications per input sample.

For a codebook of size N, $\log_2(N)$ bits are required to assign a unique index to each codevector. The data rate, r, in bits per sample can then be defined as

$$r = \frac{\log_2(N)}{k} \qquad \text{Equation 2.2}$$

This equation can be rewritten to give the codebook size in terms of the data rate and vector dimension. That is, $N = 2^{rk}$ codevectors. This means that the computational complexity in multiplications per sample for a given data rate is

$$CC = 2^{rk} \qquad \text{Equation 2.3}$$

Similarly, the storage complexity of VQ can be given as

$$SC = kN = k2^{rk}$$

since storage of N codevectors of k dimensions are required. The average number of multiplications can be reduced somewhat by realizing that once the summation in Equation 2.1 is larger than the minimum distance seen so far, the remainder of the summation terms do not have to be calculated since the codevector cannot be the nearest neighbor. The worst case complexity, however, is still given by Equation 2.3.

Since the main performance gain of VQ lies in its ability to exploit inter-block correlations, there is a motivation to increase the vector dimension in order to exploit longer term statistical dependencies. However, increasing the vector dimension for a given data rate leads to an exponential increase in the required codebook size. Table 2.1 illustrates the computational and storage complexity for image VQ at a data rate of 0.5 bits/pixel. It is easy to see that for moderate sizes of

Table 2.1 Storage and computational complexity for various block sizes in image VQ at 0.5 bpp

Block Size	Vector Dimension	Storage Complexity	Computation Complexity (mpp)	Coding Time for 512x512 at 30ns/mult (s)
3x3	9	207	23	0.181
4x4	16	4096	256	2.013
5x5	25	144825	5793	45.560
6x6	36	9437184	262144	2061.584

image blocks, the computational and storage complexities increase very rapidly. The last column in Table 2.1 illustrates the required coding time for an image of resolution 512x512 at 30 ns per multiplication. This is obviously on the optimistic side since many other operations are required to code the image, but it illustrates, in physical terms, how the computational complexity increases with vector dimension. For the case of 6x6 blocks, the required coding time is nearly 35 minutes while the coding time for 5x5 blocks is less than a minute.

Constrained Vector Quantization

A number of techniques have been proposed to overcome the complexity barrier of VQ. These techniques generally impose some constraint on the vector quantizer to reduce the complexity in exchange for sub-optimal performance. A large reduction in complexity is generally obtainable with only a minimal reduction in performance.

Tree Structured Vector Quantization

In tree structured VQ [1], the search for an appropriate codevector is performed in several stages, as shown in Figure 2.5. In this case, a binary tree is shown, but in general any number of branches can be used. At each stage, a portion of the codebook is removed from consideration by comparing the input vector to a number of specially designed test vectors. The results of these comparisons with the test vectors determine which portion of the codebook is searched. For example, at the first stage, the input vector is compared to the test vectors T1 and T2, and the path is selected corresponding to the test vector that gives the smallest distortion. At the second stage, the input vector is compared to either T11 and T12 or T21 and T22, depending on the branch taken at the first stage. Again, the path corresponding to the test vector yielding the smallest distortion is selected. This process is continued until the leaf nodes, which represent the actual codevectors, are reached.

Although the tree structured VQ provides a reduction in the number of vector comparisons, the storage complexity is larger since the test vectors, as well as the original codebook must be stored. The computational complexity in multiplications per vector for a B branch tree structured search of a codebook of size $N = B^h$, where h is the tree height, is simply $B\log_B(N)$. The number of test vectors that need to be stored is $B(B^{h-1}-1)/(B-1)$ so the total number of vectors that need to be stored for the tree structured VQ is $N+(N-B)/(B-1)$. For a codebook of

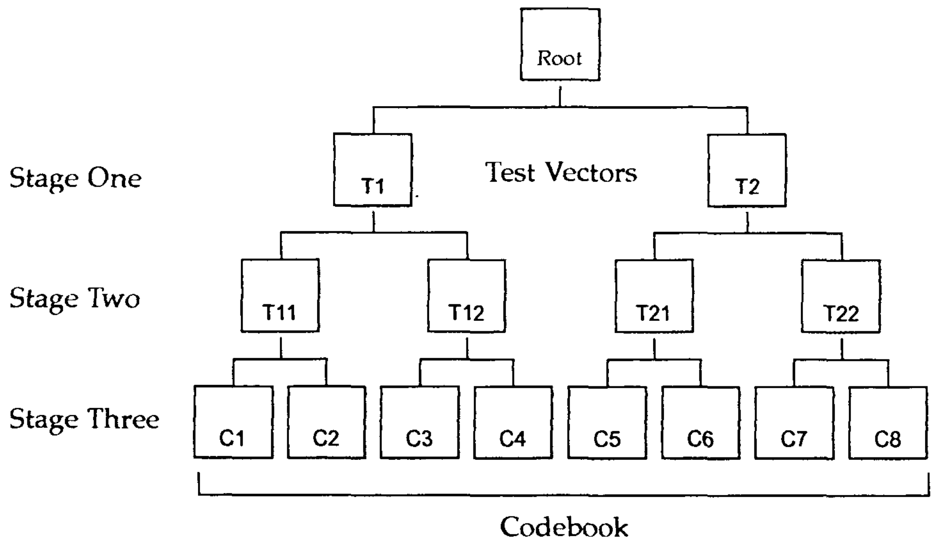


Figure 2.5 Tree structured VQ

size 1024, only 20 vector comparisons are required for a binary tree structure but the storage complexity has nearly doubled to 2046. If a four branch tree structure were used, the number of vector comparisons remains 20 while the storage complexity is 1364.

Transform Vector Quantization

In transform VQ [1-3], the comparison of the input vector to each codevector is performed in the transform domain. Each input vector is transformed using a linear transform with energy compaction properties, such as the discrete cosine transform (DCT). Since most of the energy in the transform domain is contained in relatively few coefficients, only the p most significant coefficients are compared, where $p < k$. Thus, transform VQ has the effect of reducing the vector dimension for comparison purposes.

The computational complexity for transform VQ is given by Equation 2.3, with the substitution of p for k, plus the cost of the linear transform, usually about k multiplications per sample. Since the transform is only performed once for each input vector, the cost of the transformation is usually small compared to the savings due to the reduction in vector dimension. Additionally, the simplicity of the decoder is maintained since the codevectors can be transformed back to the original domain prior to storage so no transformation is necessary at the decoder. Thus, the storage complexity for the encoder is pN while the storage complexity for the decoder is kN.

Classified Vector Quantization

Classified VQ [1][7] uses a codebook switching technique to reduce the search complexity. The codebook in classified VQ consists of several smaller codebooks called sub-codebooks of which only one is searched for each input vector. The basic block diagram for classified VQ is shown in Figure 2.6. Features are extracted from each input vector and the vector is classified into one of M classes. Each class has a sub-codebook, D_i, associated with it which is searched to find the appropriate codevector. If the cost of classification is low, a large reduction in search time can be realized since only a small portion of the entire codebook needs to be searched. For example, if a codebook size of 1024 is used with 16 equal

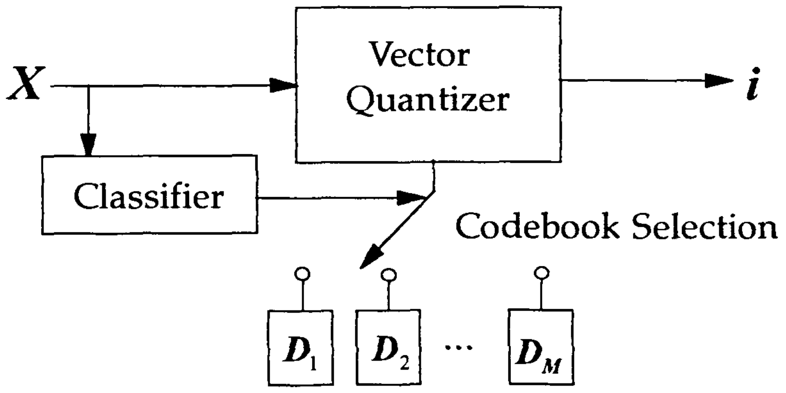


Figure 2.6 Block diagram of classified VQ

sized classes, only 64 codevector comparisons are necessary. In general, however, the classes are not of equal size.

Mean Removed and Gain-Shape Vector Quantization

Mean removed VQ [1][2] can yield a substantial reduction in computation and storage by separately coding the vector mean and its mean removed residual vector. If the size of the mean codebook is N_m and the size of the residual codebook is N_r, there are N_rN_m possible reproduction vectors. However, since the mean and residual codebooks are searched individually, the computational complexity is reduced. In general, the computation required to quantize the mean is negligible compared to the residual comparisons since the mean requires only a scalar comparison. Therefore, the computational complexity of the mean removed VQ is approximately the size of the residual codebook. If, for example, there are 32 possible mean values and 256 residual codevectors, there are 8192 possible reconstruction vectors. Computationally this requires 256 vector comparisons and at most 32 scalar comparisons. The resulting VQ is sub-optimal, however, since many of the 8192 reproduction combinations will not be used.

An approach that is similar to mean removed VQ is gain-shape VQ. In gain-shape VQ [1-3], the gain and residual (shape) vectors are separately coded. However, finding the optimal gain-shape combination is not as straightforward as the mean removed case. If the gain were removed and quantized independently, it is possible that a gain-shape combination that is not the nearest neighbor would be chosen. In order to find the nearest neighbor combination, the shape codevector that is maximally correlated to the input vector is selected as the residual codevector. The gain value that minimizes the distortion between the selected shape codevector and the input vector is then calculated and quantized using the gain codebook. This process can be shown to yield the nearest neighbor codevector [1].

The mean removed VQ and the gain-shape VQ belong to a more general class of vector quantizer called product code VQ [1][2]. Product code techniques divide a large task into several smaller tasks in order to make them more manageable. In general, product code techniques will perform reasonably close to optimal provided that the components of the product code are statistically independent. If the components that are separated contain statistical dependencies, the possibility of exploiting these dependencies is lost when they are coded separately. Therefore, to maintain approximately the same performance level of unconstrained VQ, components of the product code decomposition should be statistically independent.

Vector Quantization with Memory

Since the vector sizes used in unconstrained VQ are limited by the computational complexity, there is usually a substantial amount of correlation remaining between adjacent vectors. It is this fact that has inspired the use of vector quantizers with memory. This type of vector quantizer is capable of improving the performance for a given codebook size by exploiting the statistical dependencies between adjacent blocks.

Predictive Vector Quantization

The most straightforward way of implementing memory into the vector quantizer is to use a vector technique that is a generalization of the scalar DPCM technique called predictive VQ [1-3]. In the case of predictive VQ, an estimate of the current input vector is created using reconstructions of previous vectors. An error vector is then formed by finding the difference between the current input vector and the estimated vector. This error vector is then coded using a codebook of vector errors.

In cases where there is significant correlation between adjacent vector components a large gain in performance over unconstrained VQ can be achieved for a given bit rate and complexity. Unlike unconstrained VQ, however, increasing the block size in predictive VQ does not imply an increase in performance. The reason for this is that using larger block sizes has a tendency to reduce the amount of correlation between adjacent blocks. This reduction in inter-block correlation has the effect of making accurate vector predictions more difficult to obtain. Thus, as the vector dimension is increased, the performance of predictive VQ should be expected to approach the performance for the unconstrained VQ for the same data rate.

Finite State Vector Quantization

Finite state VQ [1][4][5] employs memory into the coding process by using a feedback loop. In finite state VQ, the behavior of the system at any given instant of time can be described by the current output and the state of the system. The state of the system represents a summary of the past operations by the system and is used to determine which state codebook to use with the vector quantizer.

The basic encoding structure of a finite state vector quantizer is shown in Figure 2.7. Finite state VQ, as the name implies, consists of only a finite number, K, of possible states the system can occupy. After each output is emitted from the vector quantizer, the next state S_{n+1} is calculated from the current output index, i, and the current state of the system, S_n, by the next state function. This new state determines which of the K codebooks will be used to code the next vector. Thus, the finite state VQ attempts to predict a good codebook for the current vector based on information from past vectors. The decoder is capable of predicting the same codebook since the state and the current symbol are known. Thus, the next state can be calculated using the same next state function. Note that finite state VQ is similar to the switched codebook structure of classified VQ. The difference here is that the finite state VQ uses the classification of the past vectors to determine the current codebook.

Address Vector Quantization

Address VQ [1][6] is another approach that attempts to exploit correlations between adjacent vectors. In address VQ, however, the vectors are initially coded with a memoryless vector quantizer and a type

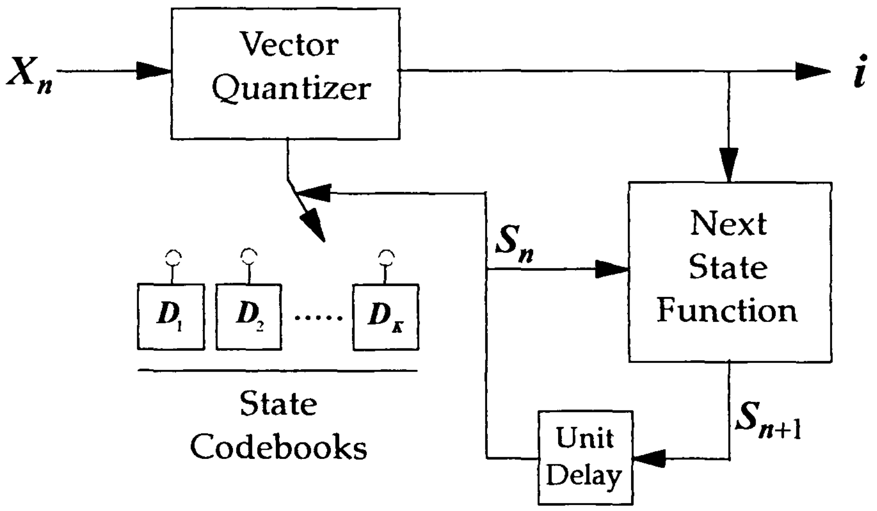


Figure 2.7 Block diagram of finite state VQ encoder

of lossless compression is applied to the resulting indices. The codebook in address VQ is divided into two sections: the VQ codebook and the address codebook. This codebook structure is shown in Figure 2.8. Each entry in the address codebook section consists of a sequence of indices corresponding to locations in the VQ codebook. Each input vector is first coded using the VQ codebook. After a specified number of blocks have been coded, the address codebook is searched for an entry that matches the given sequence of indices. If a match is found, the index corresponding to the matching entry is used to code the group of vectors, otherwise, the index of each individual vector is used. Thus, address VQ is essentially a variable rate coding scheme where more bits are assigned to groups of blocks that are not contained in the address codebook.

One of the drawbacks to the address VQ technique is that the address codebook must be quite large to obtain significant compression improvement. This results in a storage and search complexity problem. In [6], address VQ was used to code images using 4x4 blocks of pixels for

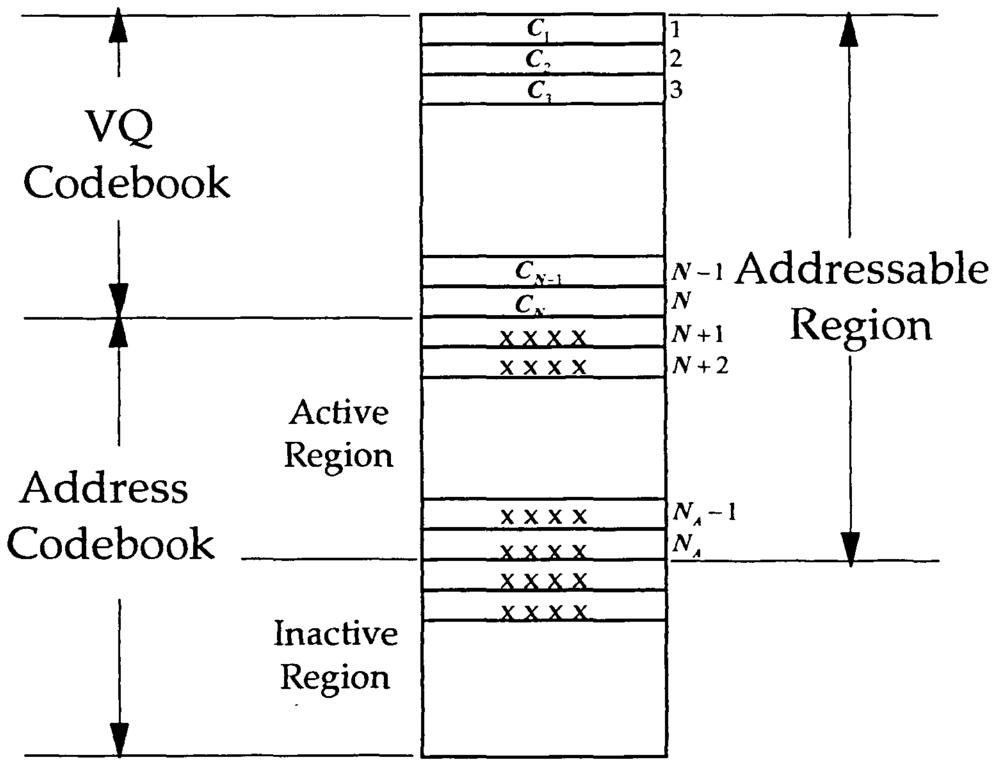


Figure 2.8 Codebook organization for address VQ

the VQ coder while the address codebook was populated with VQ indices from groups of four neighboring blocks. Good compression performance was obtained from this technique since more than 70% of the blocks were coded using the address codebook. However, the address codebook consisted of about 100,000 different index combinations. In order to prevent the need to search the entire address codebook and the need to assign unique indices to each address entry, the address codebook was divided into an active and inactive region of which only the active region was addressable. This is shown in Figure 2.8. The address codebook was then reordered based on pre-computed probabilities to keep the most probable address combinations in the active region. The need to store the 100,000 address combinations and the need to reorder the codebook after coding each group of four blocks makes the complexity prohibitive. The decoder complexity has also increased since it must also store and reorder the large number of address entries.

CHAPTER 3. ARITHMETIC CODING

Many of the binary codes used to represent data on digital computers arise naturally as fixed length codes. That is, each symbol in the code is represented by the same number of bits. For example, text characters are commonly represented on computers by standard eight bit ASCII codes. Similarly, quantization levels for digital audio are generally represented as fixed sixteen bit codes. The use of fixed length codes allows easy manipulation and processing of digital data. However, when efficient storage or transmission of data is necessary, fixed length codes are usually not the most efficient way to represent the data. Fixed length codes are efficient only when each symbol in the code has an equally probable chance of occurring. However, if the code symbols do not occur with equal probabilities, it is reasonable to expect that the average data rate could be reduced by assigning smaller length codes to symbols that occur more often and longer codes to less frequent symbols. This is the underlying concept behind lossless coding. In order to lay the ground work for a discussion of lossless coding, a brief discussion of some important concepts from information theory is provided.

Information

Information can be defined as the degree of surprise communicated by a message. Messages that produce a great deal of surprise contain a large amount of information and conversely, messages that are not surprising contain very little information. For example, the message "it is snowing in Iowa" contains very little information if it is received in January since it snows quite often in Iowa in January. However, if the message is received in April, it contains a relatively large amount of information since it rarely snows in Iowa in April. Thus, information conveyed by a message is related to the probability that the message occurred. Highly probable messages contain less information content than low probability messages.

In a mathematical sense, the information, in bits, conveyed by the symbol z is defined as

$$I(z) = -\log_2(P(z)) \qquad \text{Equation 3.1}$$

where $P(z)$ is the probability of the event z [8]. Thus, events that always occur yield zero bits of information while events that never occur yield infinite information content. The average information, or entropy, conveyed by an information source, Q, with U possible symbols $\{z_1, z_2, \ldots\ldots, z_U\}$ is defined as

$$H(Q) = -\sum_{i=1}^{U} P(z_i)\log_2(P(z_i)) \text{ bits/symbol} \qquad \text{Equation 3.2}$$

The entropy can be viewed as a measure of the uncertainty associated with an information source. According to Shannon's noiseless coding theorem, the entropy of an information source represents the theoretical minimum average codeword length achievable per source symbol. For example, a four symbol source {a,b,c,d} with probabilities of 0.1, 0.3, 0.4 and 0.2 respectively has an entropy of 1.846 bits/symbol which represents the smallest average codeword length for this source. This theoretical compression bound is obtainable if each symbol is coded with a codeword length equal to its information content. This leads to the interpretation of Equation 3.1 as the ideal codeword length for the source symbol z.

Huffman Codes

The most well known and widely used variable length coding technique is Huffman coding. When symbols from an information source are to be coded individually, Huffman coding yields the smallest possible code length. This is not to say, however, that Huffman codes are optimal. Huffman codes are only optimal when the symbol probabilities are integral powers of one half. This is due to the fact that Huffman codes require an integral number of bits for each symbol, where as the ideal length for each symbol, given by Equation 3.1, is only an integer when the symbol probability is an integral power of one half. Huffman codes have an advantage over other variable length codes in that no codeword is a prefix to another codeword. This is useful when decoding a message since information about the length of each symbol is not needed. The data stream can be scanned until a valid codeword is identified. One of the drawbacks of Huffman coding is that new codebook must be generated whenever the source statistics change.

To illustrate how a Huffman code is constructed, consider the example of a four symbol source alphabet, Z = {a,b,c,d}, with the symbol probabilities of 0.500, 0.250, 0.1250, 0.1250 respectively. To construct the code, the source is subjected to a series of alphabet reductions until only two symbols remain as shown in Figure 3.1. At each stage, the source alphabet is reduced by combining the two symbols with the lowest probabilities into a new symbol. For example, in the first alphabet

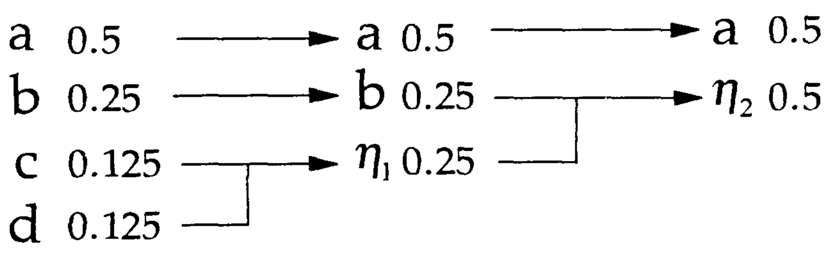


Figure 3.1 Alphabet reductions for Huffman codes

reduction, the two least probable symbols are "c" and "d" so they are grouped into a single new symbol, "η_1" with probability of 0.250. The second alphabet reduction combines "η_1" and "b" into a new source symbol, "η_2" with a probability of 0.500. The process of alphabet reductions continues until only two source symbols remain. In this example, only two alphabet reductions are necessary.

To illustrate how the code alphabet is constructed, the alphabet reductions are shown in Figure 3.2 in the form of a rooted binary tree. The terminal nodes in the tree are the four original source symbols, the next higher level represents the source alphabet after the first reduction and so on. Each branch in the tree is labeled with a bit value, 0 or 1, as shown. To find the code symbol for each symbol in the source alphabet the tree is traversed starting at the root node until the desired terminal node is reached. The combination of branch labels encountered while traversing the tree represents the Huffman codeword for the given source symbol. For example, the terminal node corresponding to the source symbol "c" is reached by taking the branches 0,0, and 1 so the codeword for that symbol is "001". The choice of the branch labels at each stage is arbitrary so there are actually several possible Huffman codes for this source.

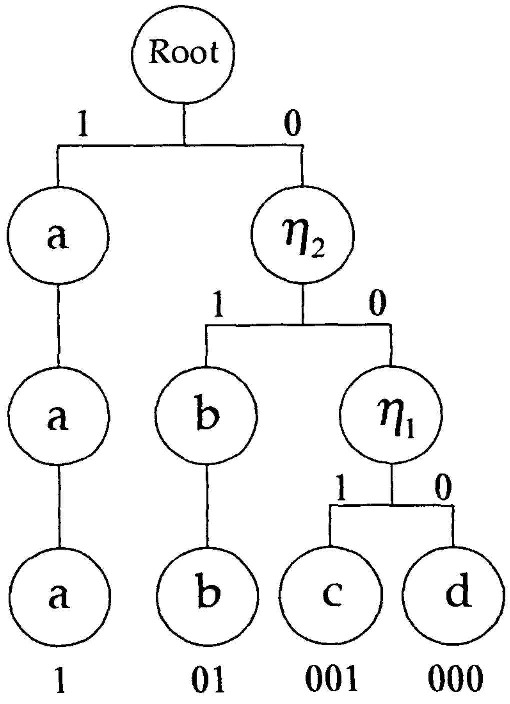


Figure 3.2 Construction of Huffman code

If the probability estimates for the example above are correct, the average number of bits/symbol is

$$(0.500)*1 + (0.250)*2 + (0.125)*3 + (0.125)*3 = 1.750 \text{ bits/symbol}$$

The entropy for this example can be calculated from Equation 3.2 as 1.750 bits/symbol. In this example, the Huffman code performs optimally since the symbol probabilities are integral powers of one half. In general, however, the performance of the Huffman code will be slightly sub-optimal.

Arithmetic Coding

Arithmetic coding offers an attractive alternative to Huffman codes when adaptive compression or source modeling is desired. Unlike Huffman codes, arithmetic coding does not require blocking of the data stream so greater compression efficiency can be realized. In fact, arithmetic coding is capable of performing near the theoretical compression bound, even when the symbol probabilities are not powers of one half [9][10]. Arithmetic coding also handles varying statistics quite easily.

In arithmetic coding, codewords can be viewed as real valued points on the interval [0,1). The goal of the arithmetic coder is to assign to the given message a point within this interval that uniquely distinguishes it from any other message. The interval [0,1) is distributed among the possible symbols according to their probabilities. Using the same source symbols and probabilities as the Huffman coding example, the interval [0,1) can be divided as shown in Figure 3.3. Thus, any point within the

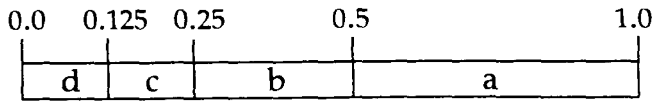


Figure 3.3 Interval division for arithmetic coding

interval [0.5,1.0) can be used to describe the symbol "a", any point within the interval [0.25,0.5) can be used to describe the symbol "b", and so on. As each source symbol in a given message is encoded, the corresponding sub-interval is distributed among the possible source symbols according to their probabilities. For example, Figure 3.4 illustrates how the interval [0,1) is divided as each symbol in the message "aacbd" is encoded. Since the first symbol in the message is "a", the sub-interval corresponding to "a" is divided among the source symbols for use in coding the second

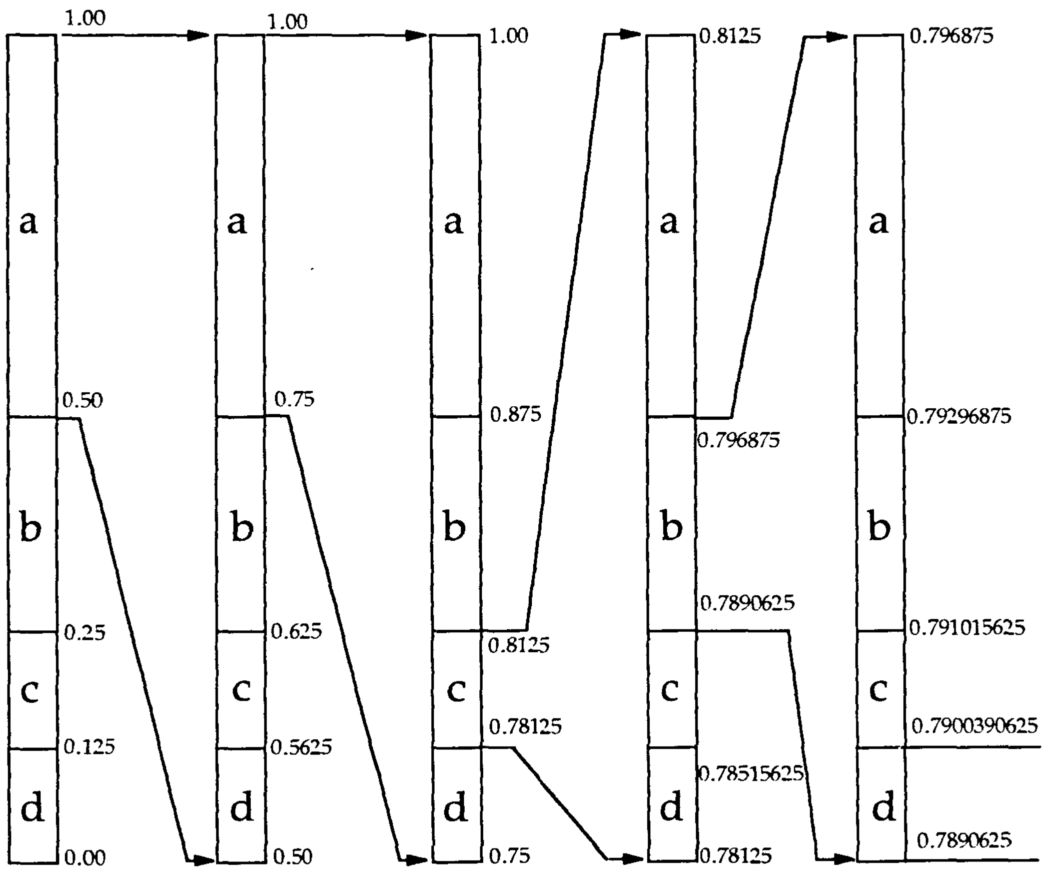


Figure 3.4 Example of arithmetic coding

symbol. Since the second symbol is again "a", its interval [0.75,1) is divided so that the third symbol can be coded. This process is continued until the entire message has been coded. For the message "aacbd", the final interval is [0.7890625,0.7900390625) which uniquely distinguishes the message from any other five symbol message. Any decimal value within this interval can be used to represent this message. When implementing the technique, it is convenient to choose the lower bound of the interval as the code value.

To decode the message from the given decimal representation, the decoder mimics the interval division performed by the encoder. In the previous coding example of Figure 3.4, the decoder can immediately determine that the first symbol is "a" since the coded point falls within this interval, [0.5,1). The interval division is then performed as it was at the encoder. Once again, the decoder can determine that the second symbol is "a" since the coded point lies within this sub-interval, [0.75,1). This is continued until the message is completely decoded. In order for the message to be properly decoded, however, the length of the message must be known. For example, if the point 0.7890625 is used to code the message, the decoder cannot determine the exact message without knowledge of the message length. The messages "aacbd" , "aacbdd" and "aacbddd" could all be represented by this same code value. In cases where the length of the source sequence is not known in advance, a special symbol is reserved as an end-of-message symbol. In the previous example, the source symbol "d" might be reserved as the special end-of-message symbol.

The arithmetic coding technique can be described formally through the use of recursive formulas. Let F be defined as a table containing the cumulative probabilities of the discrete source alphabet, Z, then the probability for the i^{th} source symbol is $P(z_i) = F(i) - F(i-1)$. If $X(\alpha)$ is the start of the interval for source sequence $\alpha=\{a_1,a_2, \ldots \}$ and $W(\alpha)$ represents the width of the interval, the recursive formulas are given by

$$\begin{aligned} &X(\varepsilon)=0 \qquad W(\varepsilon)=1 \\ &X(\alpha\, a_i)=X(\alpha)+W(\alpha)F(i) \\ &W(\alpha\, a_i)=W(\alpha)P(a_i) \end{aligned} \qquad \text{Equation 3.3}$$

where ε is the initial empty sequence and αa_i is the concatenation of the previous source sequence, α, and the current source symbol, a_i.

One of the primary advantages of using arithmetic coding is that it can handle varying statistics very easily. The probability distribution at each stage in the coding process can be changed as desired. For example, if the statistics are to be gathered adaptively from the data being compressed, the probabilities presented to the coder can be updated after each symbol has been coded. The ability to alter the statistics at each stage is also useful when using conditional source models which will be discussed in Chapter 4. As long as the decoder is capable of reproducing the way the encoder produced the probabilities at each stage, the message will be decodable.

Implementation Considerations

Although the concepts behind arithmetic coding are relatively simple, implementation of the technique is a non-trivial task. A number of difficulties are encountered when implementing arithmetic coding with finite precision arithmetic. The most obvious problem is the possibility of underflow since the interval width is repeatedly multiplied by fractional probabilities. To prevent underflow from occurring, the interval is rescaled after each symbol is coded to allow sufficient resolution for any further interval divisions.

Consider, for example, the coding example in Figure 3.4 with the cumulative probabilities replaced by their binary fraction equivalents. This is shown in Figure 3.5. Once the leading bits in the binary representation of the interval boundaries agree, no further division of the interval will change these bits. Every point between the two boundaries must start with this leading digit. Therefore, the bits that agree are no longer needed so they can be transmitted and the interval rescaled. In

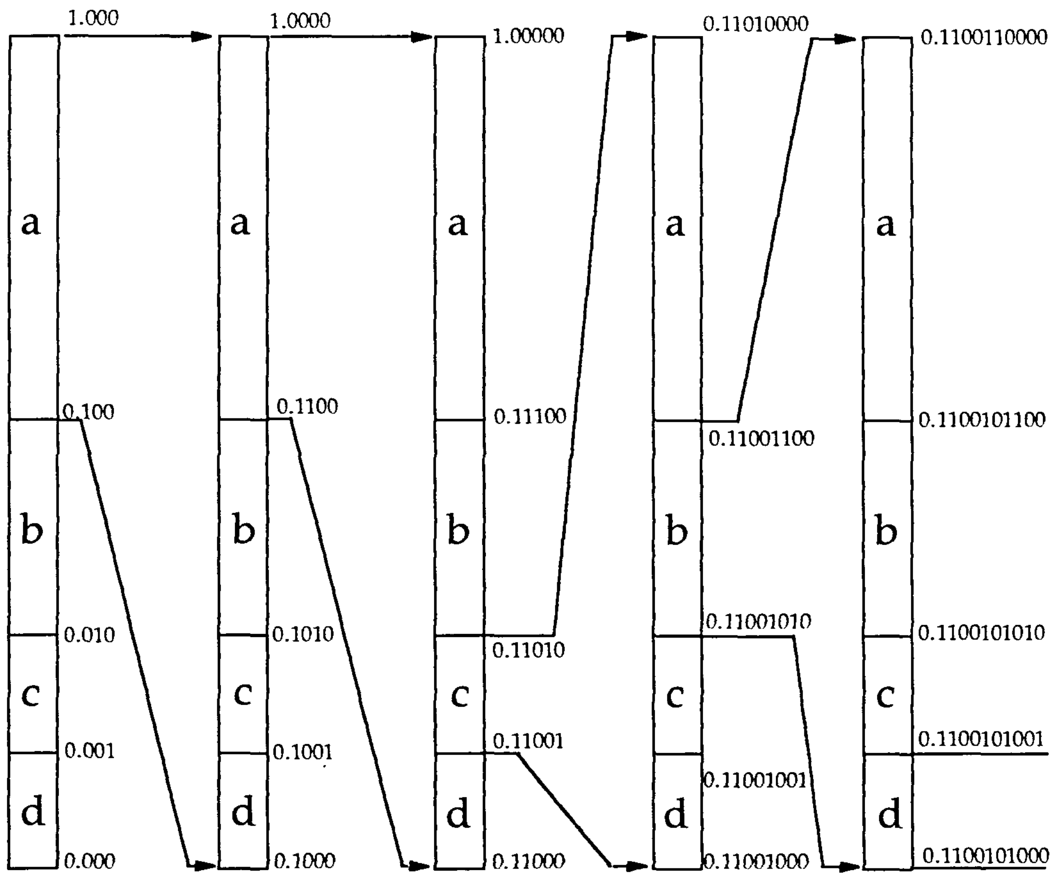


Figure 3.5 Example of arithmetic coding with binary interval representations

other words, leading bits can be shifted out until the leading bits in the boundary representations no longer agree. For example, in Figure 3.5, no bits can be shifted out after the first two divisions since the leading bits in the interval boundaries, [0.100,1.000) and [0.1100,1.000), do not agree. In the third division, however, the interval becomes [0.11001,0.11010) so the first three digits, 110, can be shifted out and transmitted. The interval can then be rescaled to [0.01,0.10) without loss of information. Similarly, in the next division, the bits 01 can be shifted out and the interval rescaled to [0.01,0.10), and so on.

Although this rescaling worked well in this example, there is still no guarantee that an underflow condition will not occur. If, for example, a symbol with small probability is coded, the interval may become too small for further divisions even though the leading bits in the boundary representation do not agree. An example of such an interval could be [0.100000...,0.011111...). In this case, no leading bits agree and further divisions of the interval would definitely produce an underflow condition.

To alleviate this problem, the interval width can be rescaled to a large enough value to prevent underflow. That is, the interval width can be rescaled so that there will be enough precision even if the symbol with the smallest probability occurs. To see how the precision can be guaranteed, the integer implementations of the cumulative probability table, the start of the interval, and the interval width are considered. The symbol probabilities are implemented as a frequency count out of K_m source symbols so that the cumulative probability table represents the accumulation of these probabilities. If $\hat{F}$ is the integer representation of F, then the probability of the i^{th} source symbol is specified as z_i occurring $\hat{F}(i) - \hat{F}(i-1)$ times out of K_m symbols. If $\hat{X}$ and $\hat{W}$ represent the integer implementations of X and W, the recursive formulas given in Equation 3.3 can be written as

$$\begin{aligned}
&\hat{X}(\varepsilon) = 0 \qquad \hat{W}(\varepsilon) = d^{\omega} \\
&\hat{X}(\alpha\, a_i) = \left(\hat{X}(\alpha) + \left\lfloor (\hat{W}(\alpha)\hat{F}(i-1)/K_m) + \tfrac{1}{2} \right\rfloor\right) d^{s} \\
&\hat{W}(\alpha\, a_i) = \left(\left\lfloor (\hat{W}(\alpha)\hat{F}(i)/K_m) + \tfrac{1}{2} \right\rfloor \right. \\
&\qquad\qquad \left. - \left\lfloor (\hat{W}(\alpha)\hat{F}(i-1)/K_m) + \tfrac{1}{2} \right\rfloor\right) d^{s}
\end{aligned} \qquad \text{Equation 3.4}$$

where $\lfloor * \rfloor$ represents the greatest integer less than or equal to the argument. The parameter s is chosen such that

$$d^{\omega} \le \hat{W}(\alpha\, a_i) < d^{\omega+1}$$

where ω is the number of d-ary digits used to represent the interval width. This is equivalent to saying that $\hat{W}$ has ω+1 d-ary digits of precision. An important restriction on the cumulative probabilities is that $\hat{F}(i) \neq \hat{F}(i-1)$, otherwise the interval width could become zero if the i^{th} symbol occurs. In order to ensure that enough precision exists to represent further interval divisions, ω must be chosen such that

$$\frac{\hat{F}(i)-\hat{F}(i-1)}{K_m} d^{\omega} \geq 1 \qquad \textit{for all } i$$

As previously mentioned, it is convenient to choose the lower bound of an interval to represent the coded point for the given message. Equation 3.4 shows that the calculation of $\hat{X}(\alpha a_i)$ at each stage is accomplished by adding a term containing the interval width to the value of $\hat{X}(\alpha)$ and scaling by d^s. The term added to $\hat{X}(\alpha)$ is the interval width scaled by the factor $\hat{F}(i-1)/K_m$ which represents the cumulative probability of the symbol a_{i-1}. By definition, this scaling factor is less than 1.0 since $\hat{F}(i-1) < K_m$ for all i, so that the term added to $\hat{X}(\alpha)$ is at most $\hat{W}(\alpha)$. This means that the addition involves only the ω+1 least significant digits of $\hat{X}(\alpha)$ with the exception of a possible carry over. If $\hat{X}$ is represented by ω+1 d-ary digits, another register $\hat{V}$ of size L can be used as shown in Figure 3.6 to store the digits shifted out of $\hat{X}$ in the event that a carry over occurs. It is possible, however, that the carry out of $\hat{X}$ may propagate all the way to the beginning of the coded message so a large buffer register may be required. This carry over problem represents another difficulty in the implementation of arithmetic coding.

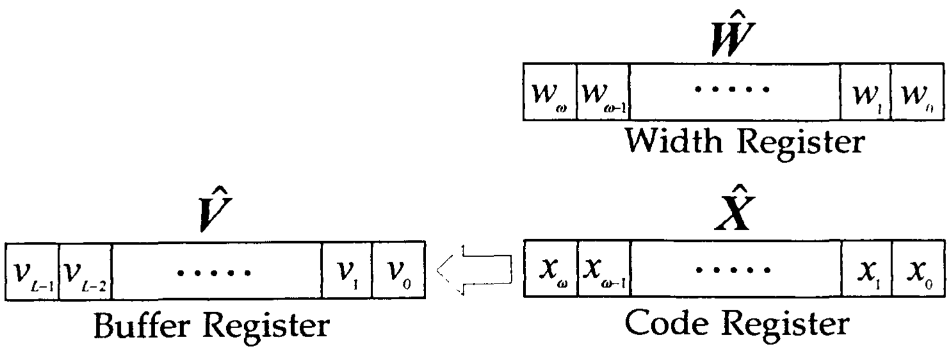
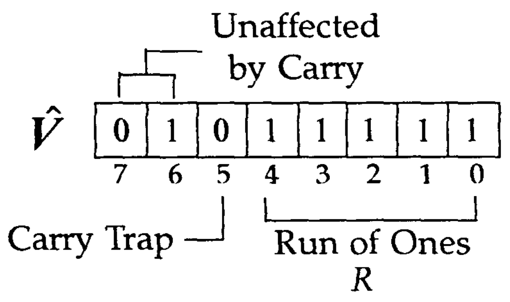
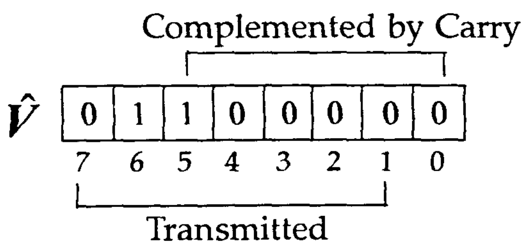

$\hat{W}$
w_ω $w_{\omega-1}$ ..... w_1 w_0
Width Register
$\hat{V}$
v_{L-1} v_{L-2} ..... v_1 v_0
Buffer Register
$\hat{X}$
x_ω $x_{\omega-1}$ ..... x_1 x_0
Code Register

Figure 3.6 Registers for implementation of arithmetic coding

One method for overcoming the need for a large storage buffer uses a run-length representation of the buffer [11]. Consider, for example, the buffer register shown in Figure 3.7a. In this case, the two left-most bits in the register are protected from a carry out of $\hat{X}$ because of the zero in bit five of the register which would terminate the carry propagation. Therefore, these bits can be transmitted since they cannot be altered by a carry over. The remaining bits in the register consist of a zero followed by a series of ones which can be represented by a run counter, R. In this case, the run counter would contain the value of five. If a carry out of $\hat{X}$ occurs, the first six bits in the register would be complemented as shown in Figure 3.7b. In the run-length representation, the carry-bit is transmitted followed by a series of R-1 zeros. The last zero is retained since it might still be affected by a future carry. If no carry occurs the



a) Original buffer register



b) Buffer after carry over

Figure 3.7 Example of run-length buffer representation

bits shifted into the buffer by the rescaling may affect the run counter. If a one is shifted into the buffer, the run counter is simply incremented and if a zero is shifted into the buffer by the rescaling process all of the bits to the left of the new zero are protected from future carries so they can be transmitted. For the run-length representation, a zero is transmitted followed by R ones.

Another approach to controlling the carry over problem is the use of a technique called bit-stuffing [9]. In this approach, if the buffer becomes filled with ones, a zero is inserted into the buffer and the series of ones is transmitted. The inserted zero serves the purpose of terminating any carry that may affect the series of ones. When the decoder detects this series of ones, the following bit is checked to see if a carry occurred. If the bit is a one, a carry occurred and the decoder continues the propagation of the carry.

The advantage of using the bit-stuffing approach is that the encoder does not have to wait for the carry to propagate before transmitting a long series of ones. However, this technique requires an extra bit to be inserted into the data stream whenever a specified number of ones are encountered. Also, the carry must still be propagated at the decoder side.

CHAPTER 4. SOURCE MODELING

In Chapter 3, the arithmetic coding technique was discussed which performs very near the theoretical entropy compression bound established by Shannon. This, however, does not mean that the compression performance of arithmetic coding cannot be improved upon. If the entropy of the information source can be reduced somehow, the compression bound would be lowered and the compression performance of the arithmetic coder could be improved. This process of entropy reduction is exactly what a good source model provides. Source modeling attempts to model, in a statistical sense, the way in which the information source generates its output. Models that are better able to predict the statistical behavior of the source reduce the uncertainty, and therefore the entropy, associated with the source.

In order to demonstrate the effects of source modeling, consider a simple example from English text. The most straightforward way of modeling English text would be to assign unconditional probabilities to each of the possible letters in the alphabet. For example, the letter "u" might be estimated to occur 8% of the time in all English text. This model, however, does not make use of the fact that certain sequences of symbols may be more probable than others. For example, the sequence of letters "qa" is very unlikely to occur while the sequence "qu" is much more probable. Therefore, an alternate model might be to assign probabilities based on previous letters. For example, if the letter "q" had occurred, it would be reasonable to estimate from knowledge of the English language that there is a 98% chance that the next letter will be "u". For the source sequence "qu", the first model assigns 3.64 bits of information to the letter "u" while the second model assigns only 0.029 bits of information. The second model has reduced the uncertainty by effectively predicting the behavior of the source from the previous symbol "q".

The goal of a good model is to accurately predict the probability distribution of the source prior to coding each symbol. The model's predictions are considered correct if the actual statistics agree well with the predicted statistics. In general, the more information that is made available to the model, the better the predictions will be. Therefore, more sophisticated models will provide better compression results. In practice, however, a model's degree of sophistication is limited by its complexity.

Most models use information from past symbols to make a prediction of the statistics for the next event from the source. In this way, the source model is able to exploit statistical dependencies between the previous symbols and the next symbol emitted by the source. The information is provided to the model in the form of conditional probabilities. That is, the symbol probabilities provided to the coder are conditioned on past events. These conditioning events are sometimes referred to as *contexts.*

Markov Source Models

The most common type of source model is a Markov source model. A Markov model [12-14] consists of a finite number of states in which the model can occupy at a given time, along with a set of paths representing the transition from one state to another. Each of the transition paths in the Markov model has a weight associated with it corresponding to the probability of leaving one state via that path. In terms of source modeling, the states represent the conditioning events or contexts and the transition path probabilities correspond to the probability of each symbol occurring with that context. Thus, each context is required to estimate probabilities for each possible symbol in the source alphabet.

Memoryless Model

A memoryless model is Markov model that consists of only a single state or context called a null context. This model is sometimes referred to as a zeroth order Markov model. In a memoryless source model, each of the symbols emitted by a given information source is assumed to be statistically independent of all the other symbols in the sequence. For example, if the sequence $\alpha = \{a_1, a_2, \ldots\ldots, a_U\}$ is emitted by an information source with a V symbol alphabet $Z = \{z_1, z_2, \ldots, z_V\}$, a memoryless model assumes that $P(a_i|a_{i-1}, a_{i-2}, \ldots) = P(a_i)$ for all i. An example of a zeroth order Markov model for a four (V=4) symbol source alphabet $Z = \{a,b,c,d\}$ is given in Figure 4.1. The circle is used to represent the state of the model and the symbol φ is used to represent the null context. Since there is only one context in this model, only the absolute probability of each symbol is stored so the storage complexity is simply V. The memoryless model was used when discussing arithmetic coding in the last chapter and was also used as the first model in the English text example.

The ideal code length for the source sequence α using the memoryless model is given by its information content

$$\begin{aligned} I(\alpha) &= -\log_2[P(\alpha)] = -\sum_{i=1}^{U} \log_2[P(a_i)] \\ &= -\sum_{i=1}^{V} U(z_i)\log_2[P(z_i)] \\ &= -U\sum_{i=1}^{V} P(z_i)\log_2[P(z_i)] \end{aligned}$$

where $U(z_i) = UP(z_i)$ represents the number of times the source symbol z_i occurs in the sequence α. The entropy of the message is defined as the average information content per message symbol so the entropy for the memoryless model is

$$H(Q_0) = \frac{I(\alpha)}{U} = -\sum_{i=1}^{V} P(z_i)\log_2[P(z_i)]$$

where Q_0 is used to indicate the zeroth order model. The entropy, H(Q_0), represents the minimum average codeword length obtainable using the memoryless model.

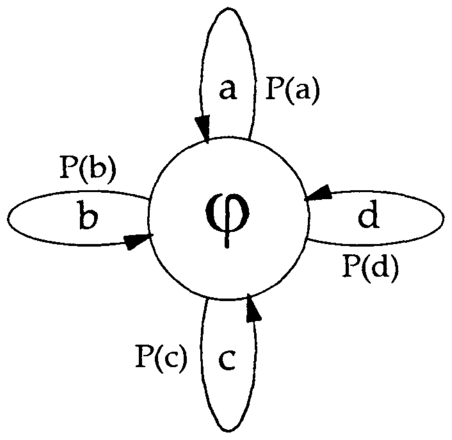


Figure 4.1 Zeroth order Markov model

First Order Model

In a first order source model, each symbol in the source sequence is assumed to be dependent on the previous symbol in the sequence. That is $P(a_i|a_{i-1},a_{i-2},a_{i-3}, \ldots) = P(a_i|a_{i-1})$ for all i. An example of a first order Markov model is given in Figure 4.2 for a four symbol source alphabet. This model will generally perform better than a memoryless model since statistical dependencies between symbols can be exploited. If there are V possible symbols in the source alphabet, then there are V possible first order conditioning events. Therefore, the storage complexity for the first order model is V^2.

The ideal code length for the source sequence α using the first order Markov model is also determined by its information content

$$\begin{aligned} I(\alpha) &= -\log_2[P(\alpha)] = -\sum_{i=1}^{U} \log_2[P(a_i|a_{i-1})] \\ &= -\sum_{j=1}^{V}\sum_{i=1}^{V} U(z_i,z_j)\log_2[P(z_i|z_j)] \\ &= -\sum_{j=1}^{V}\sum_{i=1}^{V} U(z_j)P(z_i|z_j)\log_2[P(z_i|z_j)] \\ &= -U\sum_{j=1}^{V}\sum_{i=1}^{V} P(z_j)P(z_i|z_j)\log_2[P(z_i|z_j)] \end{aligned}$$

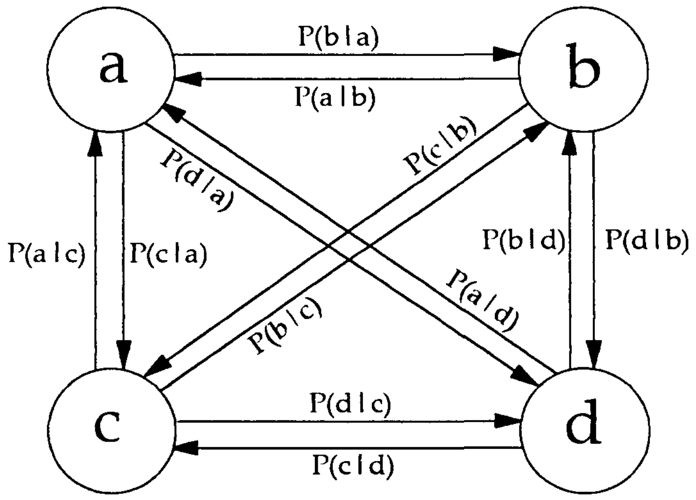


Figure 4.2 First order Markov model

where $U(z_i,z_j)$ is the number of times the two symbol subsequence, z_iz_j, occurs in α. The summation requires knowledge of the symbol a_0 which is not part of the source sequence. Different choices for a_0 will give different values for the ideal code length, however, its effect will be small for long sequences. The first order entropy is simply the average code length per message symbol. That is,

$$H(Q_1)=\frac{I(\alpha)}{U}=-\sum_{j=1}^{V}P(z_j)\sum_{i=1}^{V}P(z_i|z_j)\log_2[P(z_i|z_j)]$$

where Q_1 is used to represent the first order model. As with the memoryless model, H(Q_1) represents the compression bound for the first order model.

Higher Order Models

Higher order Markov models are capable of exploiting more of the statistical dependencies that may exist between source symbols, so they can generally offer better compression performance. However, the number of contexts required becomes prohibitively large for higher order models. The number of contexts for a model of order q and a source alphabet size of V is V^q. Since V probabilities must be estimated for each context, the overall storage complexity for the q^{th} order model is V^{q+1}. Thus, for an alphabet size of $V = 256$, the storage complexity for a first order model is 65536, while the second order model has a complexity of 16,777,216.

Generalized Condition Source Model

The conditioning events that provide the best compression performance vary from application to application. For example, models for compressing text generally use neighboring characters as the conditioning events while speech models might find useful information in neighboring samples and samples at integral number of pitch periods away. To handle the different types of conditioning events, a generalized source model has been developed [14][15].

Consider a source with a V symbol source alphabet $Z = \{z_1, z_2, \dots, z_V\}$ and let $\alpha = \{a_1, a_2, \dots, a_U\}$ be a sequence of symbols emitted from the source. Let $\Lambda = \{\lambda_1, \lambda_2, \dots, \lambda_K\}$ be the set of all possible conditioning events and $\Gamma = \{g_1, g_2, \dots, g_U\}$ be the sequence of conditioning events for α such that g_i is the context for compressing a_i. The generalized condition source model assumes $P(a_i|a_{i-1}, a_{i-2}, \dots) = P(a_i|g_i)$. The generalized model requires storage of V probabilities for each of the K contexts so the storage complexity of such a model is VK.

The set of contexts Λ can be chosen to suit the particular application. For example, in the case of speech signals, the contexts can be chosen to incorporate information about neighboring samples as well as samples that are an integral number of pitch periods away. This generalized model can also be used to represent the Markov models discussed previously. For example, the context set $\Lambda = \{\varphi\}$ yields the memoryless model. In this case, $g_i = \varphi$ for all i. Similarly, the first order model has the context set $\Lambda = Z$ and $g_i = a_{i-1}$ for all i.

The ideal code length and the entropy for the conditioned source model are determined in the same way as the first order model. In this case the ideal code length and the entropy are given by

$$I(\alpha) = -\log_2[P(\alpha)] = -\sum_{i=1}^{U} \log_2[P(a_i|g_i)]$$
$$= -U\sum_{j=1}^{K}\sum_{i=1}^{V} P(\lambda_j)P(z_i|\lambda_j)\log_2[P(z_i|\lambda_j)]$$
$$H(Q_x) = \frac{I(\alpha)}{U} = -\sum_{j=1}^{K} P(\lambda_j)\sum_{i=1}^{V} P(z_i|\lambda_j)\log_2[P(z_i|\lambda_j)]$$

These equations reduce to the memoryless model when $\Lambda = \{\varphi\}$ $(K=1)$ and to the first order model when $\Lambda = Z$ $(K=V)$.

Probability Estimations

Once a particular type of model is chosen, the probability distributions associated with that model must be estimated. A critical requirement for decodability of a message is that the encoder and decoder must have exactly the same probability distributions and context selection for each symbol that is coded. The most obvious way to meet this requirement is to maintain the same fixed symbol probabilities and contexts at both the encoder and decoder. This is often referred to as a *static* model. This method works well when the symbol probabilities for different source sequences are approximately the same. That is, when each message encoded by the model has approximately the same statistical structure. If, however, a message is encoded that does not closely match the statistics of the static model, the compression performance would be poor. In fact, the encoder may actually expand rather than compress the message.

Another approach that may perform better is to actually scan the message before coding to gather the exact statistical distributions for the given message. This would ensure the best compression for the given type of model since the estimations of the symbol probabilities would be exact. This type of model can be referred to as a *semi-adaptive* model.

The problem with this approach is that the decoder cannot estimate the probabilities in the same manner as the encoder so this information must be communicated to the decoder via side-information. The need for side-information will of course affect the overall compression performance. Also, the need to scan the entire message prior to encoding results in undesired coding latency.

An approach that solves the difficulties of the static and semi-adaptive models is known as the *adaptive* model. In this approach, the encoder and decoder both start with the same initial probability distributions and contexts. After each event is encoded, the encoder updates its probability distributions and contexts according to the symbols that have been encoded so far. At the decoder, the probability distributions and contexts can be updated in the same manner after each symbol is decoded. Thus, the encoder and decoder are able to adapt to the statistics of the given message in a synchronous fashion so there is no need for any side-information. Also, the coding latency is not a problem since the compression and the gathering of statistics can be accomplished in the same scan of the message. The disadvantage of this technique, however, is that the compression performance during the beginning of the message may be poor since the model is still learning the statistics at that time.

A practical way to implement this technique with an arithmetic coder is to begin with probability distributions that are uniform. It is convenient to use integer frequency counts to represent the relative frequencies of the source symbols. That is, the probability of a given symbol is represented as occurring an integral number of times out of the total count. Initially, the symbols all have the same probabilities so the frequency counts are all set to one and the total count is set equal to the number of symbols in the source alphabet, V. When a source symbol occurs, the frequency count corresponding to that symbol and the total count are incremented by K_i. The factor K_i controls how quickly the model adapts to the statistics. Larger values of K_i will increase the speed of adaptation, however, if this value is too large the model could have difficulty converging to the true statistics of the message. The value for K_i is generally determined experimentally.

CHAPTER 5. LOSSLESS INDEX COMPRESSION

Chapters 2, 3, and 4 have provided the ground work for a discussion of the techniques used in this chapter. Unlike past chapters which were kept general, this chapter will be concerned with the specific task of compressing digitized images. Although the discussion here will be in terms of image coding, most of the techniques, with some appropriate modifications, are applicable to other signal compression situations.

The underlying concept behind this technique is to use arithmetic coding with appropriate source models to compress the indices of VQ. In general, the indices due to the VQ are not uniformly distributed, so a reduction of the data rate is possible by entropy coding. Also, since the block sizes used in VQ are limited by complexity, the indices usually possess significant correlation between them. Therefore, appropriate source models can be used to reduce the entropy of the source resulting in further compression.

Figure 5.1 shows a block diagram of the coding system used here. The original uncompressed signal is first quantized using a vector quantizer and the resulting indices are presented to the arithmetic coder and the source model. The source model uses the information about past inputs to make a statistical prediction of the current index. The arithmetic coder uses the predictions from the source model to compress the current index.

Classified Vector Quantizer

The first stage shown in Figure 5.1 is the vector quantizer. One of the more popular VQ techniques used for image coding is the classified VQ discussed in Chapter 2. It provides the necessary complexity reduction while allowing classes to be assigned to perceptually significant events. In image coding using VQ, it has been observed that much of the

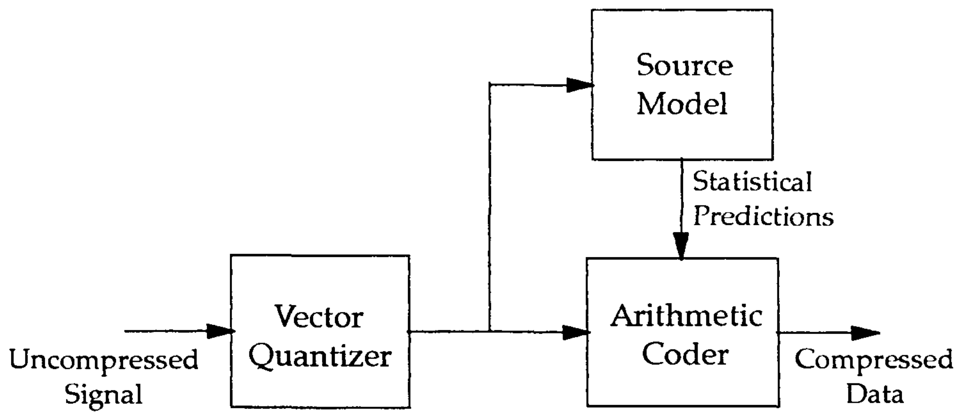


Figure 5.1 Block diagram of coding structure

reduction in subjective quality is due to edge degradation since perceptually, most of the information in an image is due to its edge content. In the classified VQ approach, a number of classes can be reserved for edges in an attempt to preserve the integrity of edges in an image.

The classifier used here is very similar to the techniques described in [7]. Each 4x4 input block is classified as belonging to one of 31 classes based on the edge content of the block. Blocks that contain very little gradient content are classified as *shade* blocks. Blocks containing a moderate amount of gradient, but no definite edge content are classified as *midrange* blocks. Blocks that have no definite single edge, but contain a significant gradient content are classified as *mixed* blocks. The remaining 28 classes are assigned to different edge classes based on edge orientation, location and polarity. A summary of the different edge classes is shown in Figure 5.2.

The classification of each block is performed with respect to a perceptual model of edges. In this model, it is assumed that the edge

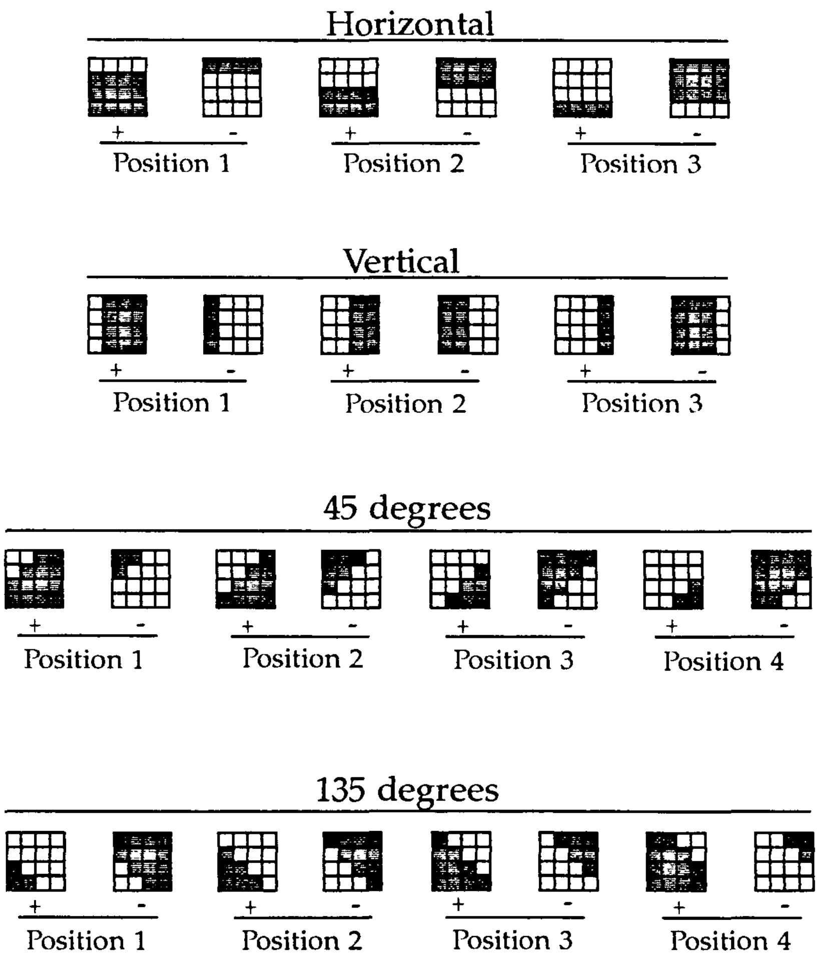


Figure 5.2 Edge classes for 4x4 blocks

perception of the eye is proportional to normalized gradient and not the actual gradient level. The gradient between two adjacent pixels is normalized by the average intensity level of the two pixels. For example, the normalized gradient between pixel $x(i,j)$ and its east neighbor $x(i,j+1)$ is

$$d_h = \frac{2[x(i,j) - x(i,j+1)]}{x(i,j) + x(i,j+1)} \qquad \text{Equation 5.1}$$

Likewise, the normalized gradient between pixel $x(i,j)$ and its south neighbor $x(i+1,j)$ is

$$d_v = \frac{2[x(i,j) - x(i+1,j)]}{x(i,j) + x(i+1,j)} \qquad \text{Equation 5.2}$$

These normalized gradients are compared to two thresholds, T_S and T_e, to determine if the pixel transition is a shade, midrange or edge transition. The shade threshold, T_S, is determined by the Weber fraction as

$$T_s = \begin{cases} 0.1 & \text{if } d_{av} < 30 \text{ or } d_{av} > 225 \\ 0.025 & \text{otherwise} \end{cases}$$

where d_{av} is the average intensity of the two pixels under consideration. The edge threshold, T_e, was determined experimentally by [7], and is given by

$$T_e = \begin{cases} \dfrac{8.0}{d_{av}} & \text{if } d_{av} < 30 \\ 0.2 & \text{otherwise} \end{cases}$$

Six counters are kept to determine the number of edge and shade transitions made in the horizontal and vertical directions. For each transition location in a given block, the counters are incremented according to Figure 5.3. There are a total of 12 possible transition locations for each direction as shown in Figure 5.4. The 12 horizontal gradients are used to increment the counters S_h, H_p, H_n and the 12 vertical gradients are used to increment the counters S_v, V_p, V_n. Once each of the horizontal and vertical transitions has been checked, a classification decision based on the values of these counters is made as shown in Figure 5.5.

In addition to the six transition counters, two tables, G_h, G_v, are used to keep track of gradient locations in the horizontal and vertical directions. These are 3x4 and 4x3 tables that correspond to the transition locations given in Figure 5.4. These tables represent edge enhanced versions of the input block and are used to determine the edge

S_h	=>	Horizontal shade counter.
		Increment when $\lvert d_h \rvert > T_s$
S_v	=>	Vertical shade counter.
		Increment when $\lvert d_v \rvert > T_s$
H_p	=>	Positive horizontal gradient counter.
		Increment when $d_h > T_e$
H_n	=>	Negative horizontal gradient counter.
		Increment when $d_h < -T_e$
V_p	=>	Positive vertical gradient counter.
		Increment when $d_v > T_e$
V_n	=>	Negative vertical gradient counter.
		Increment when $d_v < -T_e$

Figure 5.3 Gradient counters for image classifier

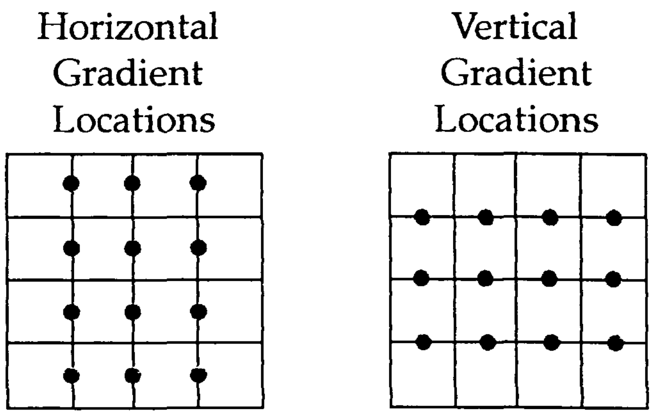


Figure 5.4 Gradient locations for 4x4 blocks

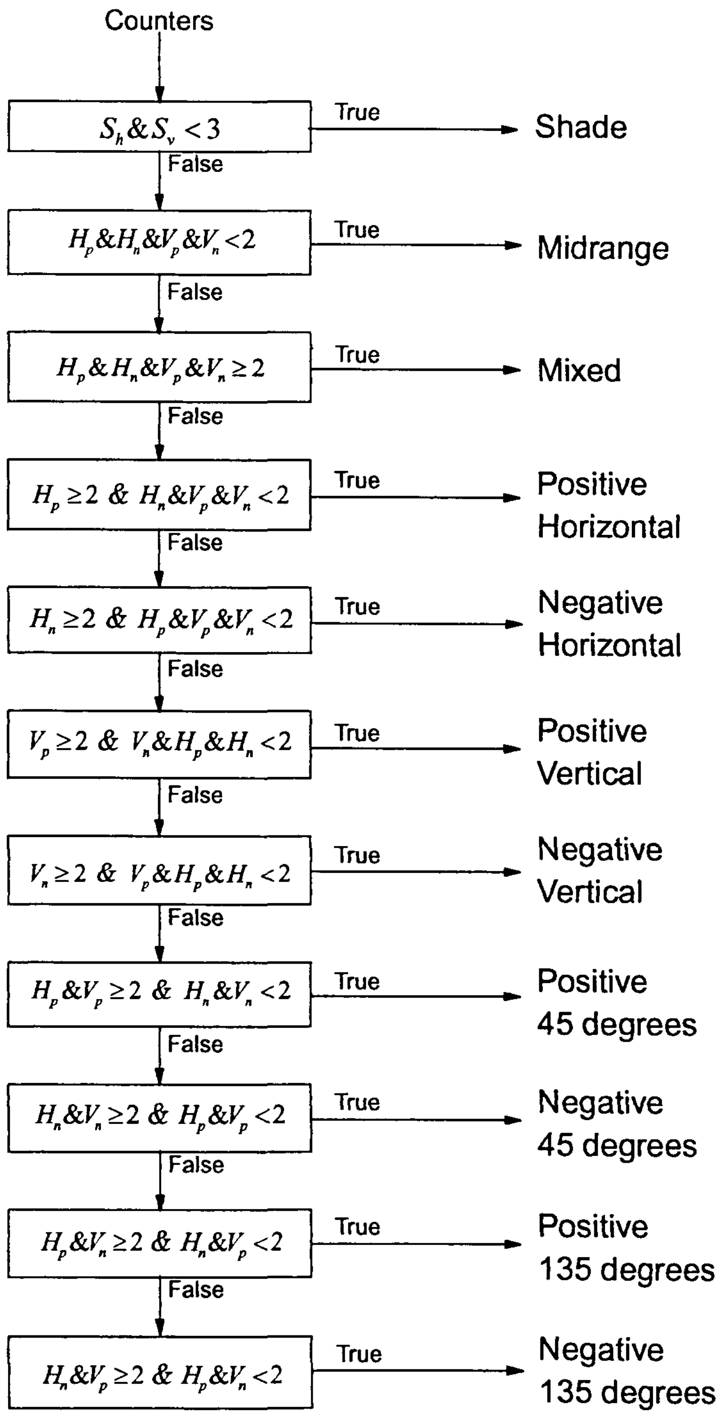

Counters
$S_h \& S_v < 3$
True
Shade
False
$H_p \& H_n \& V_p \& V_n < 2$
True
Midrange
False
$H_p \& H_n \& V_p \& V_n \ge 2$
True
Mixed
False
$H_p \ge 2 \ \& \ H_n \& V_p \& V_n < 2$
True
Positive Horizontal
False
$H_n \ge 2 \ \& \ H_p \& V_p \& V_n < 2$
True
Negative Horizontal
False
$V_p \ge 2 \ \& \ V_n \& H_p \& H_n < 2$
True
Positive Vertical
False
$V_n \ge 2 \ \& \ V_p \& H_p \& H_n < 2$
True
Negative Vertical
False
$H_p \& V_p \ge 2 \ \& \ H_n \& V_n < 2$
True
Positive 45 degrees
False
$H_n \& V_n \ge 2 \ \& \ H_p \& V_p < 2$
True
Negative 45 degrees
False
$H_p \& V_n \ge 2 \ \& \ H_n \& V_p < 2$
True
Positive 135 degrees
False
$H_n \& V_p \ge 2 \ \& \ H_p \& V_n < 2$
True
Negative 135 degrees

Figure 5.5 Decision tree for image classifier

location. If the normalized gradient at a certain location is greater than an edge threshold T_e, the corresponding location in the gradient table is set to +1. If this gradient is less than $-T_e$, the location in the gradient table is set to -1. If the magnitude of the gradient is less than T_e, then the location is set to zero. When the classifier decides that the input block is one of the 28 edge classes, the appropriate table is scanned to find the location of the edge. When determining the edge location for the horizontal class, the vertical gradient table, G_V, is scanned since vertical gradients imply horizontal edges. Similarly, the horizontal gradient table is searched for the vertical edge class. The edge location for the diagonal classes is determined by consulting both the horizontal and vertical gradient tables since a diagonal gradient can be decomposed into horizontal and vertical components.

Once the classification of the training data is performed, a separate VQ codebook is designed for each class using the LBG algorithm discussed in Chapter 2 and the resulting codebooks are merged into one "super codebook". Due to the nature of the mixed class, it is very difficult to obtain a good representative codebook since, unlike the edge classes, no common characteristic is shared by all the training vectors. This class is *essentially a miscellaneous classification.* Instead of generating a separate codebook for the mixed class, it is grouped with the midrange class. This is due to the observation that most (≈70%) of the blocks in an image correspond to a midrange classification while only a small portion (≈5%) correspond to a mixed classification, so the overall effect of the mixed class will be small. Grouping the midrange and mixed classes results in a 30 class VQ. When coding an input block, only the sub-codebook corresponding to the classification is searched, while the decoding of the image is equivalent to the standard VQ table look-up procedure.

Even though the use of the classifier reduces the complexity of the vector quantizer, the sub-codebook sizes required to adequately represent the training data are still large. In order to reduce the complexity further, the mean removed technique described in Chapter 2 was used in [7]. Although sub-optimal, this works well in image VQ since blocks with similar edge content may occur at various mean levels. A modified approach that does not require the separate coding of the mean is described in [6] and is used here. In this technique, the mean is predicted using pixels from neighboring blocks as shown in Figure 5.6. The predicted mean is then calculated as

$$M_p = \frac{1}{9}\sum_{i=1}^{9} m_i$$

This predicted mean is then removed prior to coding. The decoder predicts the mean using the same approach and adds it to the selected codevector. This approach, of course, does not result in a truly zero mean codebook since there will be some mean error that is compensated for by the codebook.

Lossless Coding

As discussed in Chapter 2, larger vector dimensions in VQ will allow the exploitation of longer term statistical dependencies resulting in

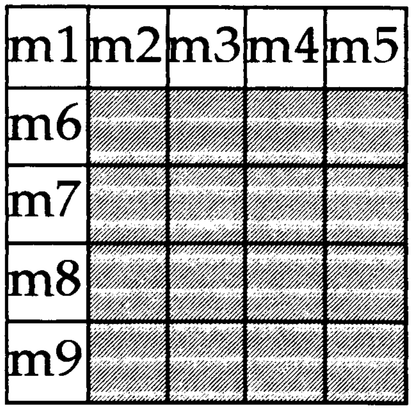


Figure 5.6 Prediction of block mean from previous blocks

improved performance. Since small block sizes are generally used due to the complexity barrier of VQ, it is reasonable to expect that some correlation between neighboring blocks will still exist. This correlation presents itself as correlation between the indices of adjacent blocks. In Chapter 2, the technique of address VQ was discussed which exploits this correlation between indices, however, the storage and computational complexity was quite large. The technique proposed here separates the VQ and lossless compression stages resulting in a large reduction in complexity.

All the models to be described gather their statistics adaptively from the image being compressed using the adaptive technique described in Chapter 3. This allows the compression of the indices to occur in a single scan.

Memoryless Model

Since the distribution of the indices is generally non-uniform, a memoryless model can be used to exploit the coding redundancy present. For most images, a large number of blocks will be classified as shade or midrange blocks, so the number of codebook indices corresponding to these classes will be relatively large compared to the edge classes. Even if another type of VQ were used, the indices will most likely favor certain codewords. This means that the uniform bit allocation of VQ can be improved upon by entropy coding the indices.

First Order Model

The memoryless model does not take advantage of any statistical dependency between adjacent blocks. A first order model, however, is capable of exploiting these correlations by using a previous index as a conditioning event. Figure 5.7 shows an example of a block, *X*, to be coded and its four causal neighbors that are candidates for conditioning events. Only causal neighbors are considered since it is usually desirable to compress the image in a single scan. If, for example, the north neighbor is chosen to be the conditioning event for the first order model,

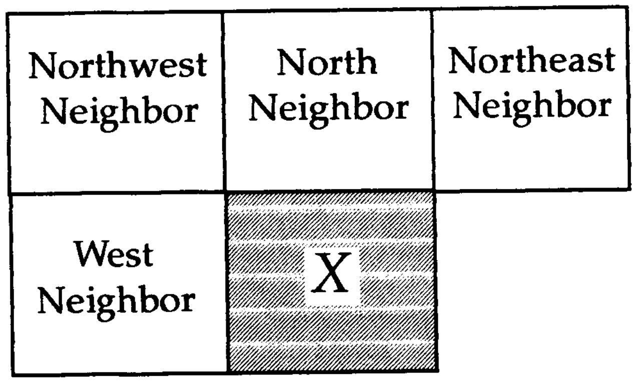


Figure 5.7 Four causal neighbors of X

the symbol probabilities presented to the arithmetic coder would be dependent only upon the index value of the north neighbor.

Two Step Method

One drawback of the first order model is that it is only capable of exploiting correlations in one direction. The correlations between the other three causal neighbors are completely ignored. One approach to this problem might be to use higher order models, such as a second, third or fourth order source model. However, the need to store every possible combination of conditioning event for such models means that the storage requirement becomes prohibitively large. For example, a modest size codebook of 256 requires a probability table for each of its $(256)^4 = 4.29 \times 10^9$ possible conditioning events. Even if storage was not a problem, there would clearly not be enough data to create good probability estimations for each combination.

The approach discussed here takes advantage of the structure of the classified VQ. The indices from the super-codebook are divided into a class component and a sub-codebook component and they are compressed separately. The reasoning here is that the classification of adjacent blocks should be more correlated than the codebook indices, so this portion of the index should compress well. Figure 5.8 illustrates how the codebook index is divided into its class and sub-codebook components. In order to reduce the complexity of the model and to further enhance the compression of the classification, the distinction between different edge locations is ignored resulting in ten classes.

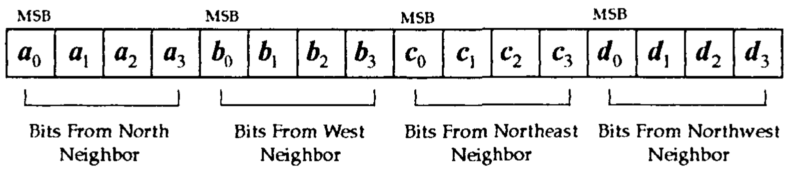
Class	VQ Index	Classification Index	Sub-codebook Index
Shade	0-7	0 Shade	0-7
Midrange/Mixed	8-39	1 Midrange	0-31
+Horizontal 1	40-45		
+Horizontal 2	52-57	2 +Horizontal	0-17
+Horizontal 3	64-69		
-Horizontal 1	46-51		
-Horizontal 2	58-63	3 -Horizontal	0-17
-Horizontal 3	70-75		
+Vertical 1	76-81		
+Vertical 2	88-93	4 +Vertical	0-17
+Vertical 3	100-105		
-Vertical 1	82-87		
-Vertical 2	94-99	5 -Vertical	0-17
-Vertical 3	106-111		
+45 1	112-120		
+45 2	130-138	6 +45 degrees	0-35
+45 3	148-156		
+45 4	166-174		
-45 1	121-129		
-45 2	139-147	7 -45 degrees	0-35
-45 3	157-165		
-45 4	175-183		
+135 1	184-192		
+135 2	202-210	8 +135 degrees	0-35
+135 3	220-228		
+135 4	238-246		
-135 1	193-201		
-135 2	211-219	9 -135 degrees	0-35
-135 3	229-237		
-135 4	247-255		

Figure 5.8 Decomposition of classified VQ index

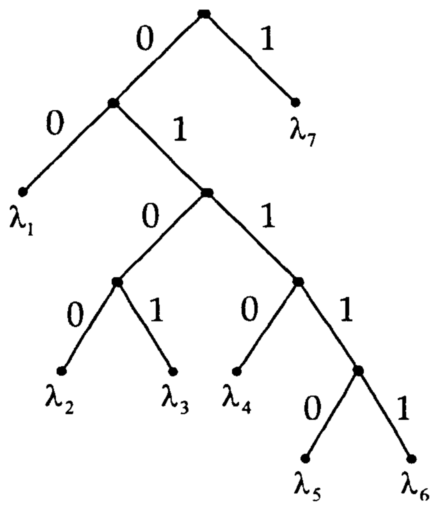
The first two columns in Figure 5.8 represent the original VQ classification and the corresponding VQ indices for an example codebook size of 256. The first 8 codewords (0-7) in the super-codebook are shade vectors, the next 32 codevectors (8-39) are midrange codevectors, etc. The third column shows how the original 30 VQ classes map to the ten classes used for lossless compression purposes. For the purpose of lossless compression, the classification according to edge location is ignored. For example, all of the positive horizontal codevectors are grouped into a single new classification (2). This reduces the number of classes to ten which reduces the complexity of the model. The sub-codebook index is found by combining all of the codevectors belonging to the new classification and renumbering them consecutively starting at zero. For example, the 32 codevectors belonging to the midrange class (8-39) are renumbered from zero to 31. For the positive horizontal class, the codevectors (40-45), (52-57) and (64-69) are grouped together and renumbered from zero to 17. These new index numbers represent the sub-codebook indices.

The first stage in the two step method is to compress the classification index of the block. To determine the context for the current block, a bit string is formed as shown in Figure 5.9. The bit representation of the classification indices of the north, west, northeast, and northwest neighbors are concatenated to form the context string. Since there are ten classes for the purpose of lossless compression, four bits are necessary to represent each index. To find the context for the current block, the context tree shown in Figure 5.9 is used. Starting at the root node, the tree is descended with the context string guiding the path at each node until a terminal node is reached. Each of the terminal nodes represents a possible context for the model. The context tree shown in the figure has seven contexts.

The ordering of the context bit string determines which of the causal neighbors receives the most emphasis when searching for the context. For example, as the tree is traversed, the west neighbor is not


MSB
MSB
MSB
MSB
a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3 c_0 c_1 c_2 c_3 d_0 d_1 d_2 d_3
Bits From North Neighbor
Bits From West Neighbor
Bits From Northeast Neighbor
Bits From Northwest Neighbor

Context String



Contexts: {00, 0100, 0101, 0110, 01110, 011101, 1}

Context Tree

Figure 5.9 Context string and context tree for two step method

considered as part of the context until after the classification of the north neighbor uniquely specified. Table 5.1 shows the conditions for choosing each of the seven contexts in Figure 5.9 where the symbol ∅ represents a "don't care". The first context is chosen whenever the north neighbor is either shade (0), midrange (1) or a horizontal block (2 or 3) regardless of the classifications of the other neighbors. The sixth context is selected

Table 5.1 Conditions for choosing contexts of Figure 5.9

	Contexts						
	1	2	3	4	5	6	7
North	0-3	4	5	6	7	7	8-9
West	∅	∅	∅	∅	0-7	8-9	∅
Northeast	∅	∅	∅	∅	∅	∅	∅
Northwest	∅	∅	∅	∅	∅	∅	∅

whenever the north neighbor is a positive 45 degree block (7) and the west neighbor is either a positive or negative 135 degree block (8 or 9). For this context tree, the classifications of the northeast and northwest neighbors are never considered and the west neighbor is considered only if the classification of the north neighbor is 7. Experiments with the different causal neighbors showed that the north neighbor provided the best compression performance, followed by the west, northeast, and northwest neighbors respectively.

The context tree is grown adaptively from the data to ensure that only the most probable contexts are used out of the large number of possible contexts. Figure 5.10 illustrates the process of growing the context tree. Initially, only a single root node exists which is called the null context. Once this context has been used a specified number of times, K_t, it is split into two contexts by converting it to an internal node and creating two children nodes. These two children now represent the newly created contexts and the number of contexts has increased by one. This process continues until the number of contexts reaches a predetermined limit, N_c. For example, after the null context in Figure

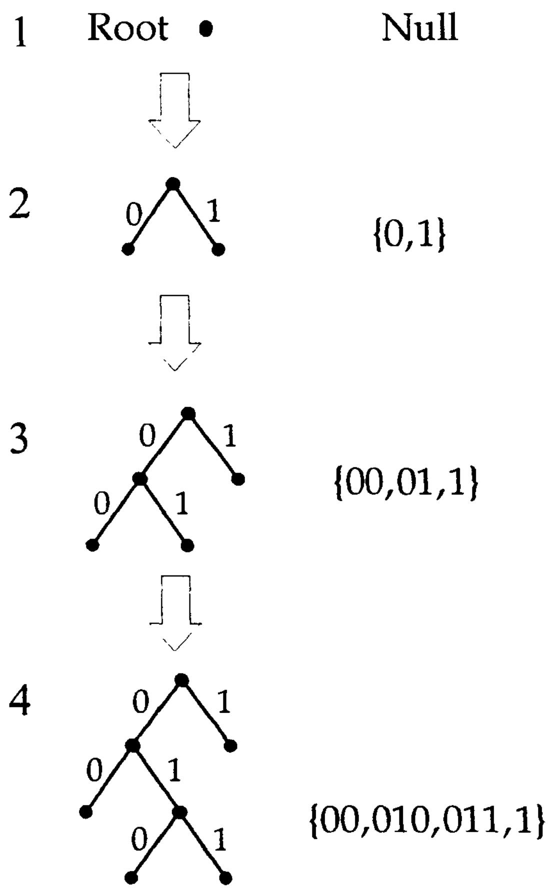


Figure 5.10 Adaptive growth of context tree

5.10 is used K_t times, it is split into the contexts 0 and 1. Once the context 0 is used K_t times, it is split into two contexts 00 and 01, and so on. The storage complexity for this part of the model is $10N_c$ since each context must store a probability estimation for each of the 10 classes.

The second stage in the two step method is the compression of the sub-codebook index. This is done using a first order model, however, the selection of the context is performed with respect to the classification of the neighboring blocks. To find the context for the sub-codebook index, the classifications of the neighboring blocks are consulted to determine if there is a match with the current class. If there is a match, the sub-codebook index of the first matching neighbor of the north, west, northeast, and northwest is used as the context for the current sub-codebook index. If no classification match occurs, a null context is used to compress the sub-codebook index.

Three examples of this context selection process are shown in Figure 5.11. The VQ index for each block is given, as well as the decomposition of the index. The classification indices are shown as the left branch of the tree in each block and the sub-codebook indices are shown as the right branch. The root of the tree represents the actual VQ index. In the first example, the block to be compressed is a midrange block (1) with a sub-codebook index of 5. The context for compressing this sub-codebook index is found by checking the neighbors for a matching class. In this case, the first matching neighbor is the west neighbor. The sub-codebook index of the west neighbor is 28 which is used as the context. In the second example, the first match is the northeast neighbor so its sub-codebook index, 6, is used as the context for compressing the index 22. In the third example, no match is found so a null context is used to compress the sub-codebook index 7.

The complexity of this part of the model is given by

$$\sum_{i=0}^{9} N_s(i)[N_s(i)+1]$$

where $N_s(i)$ is the number of sub-codebook indices for class i. This is due to the fact that $N_s(i)$ probabilities must be estimated for each of the $N_s(i)+1$ possible contexts in a given class. For the example codebook given in Figure 5.8, the storage complexity is 7824 for the sub-codebook index model.

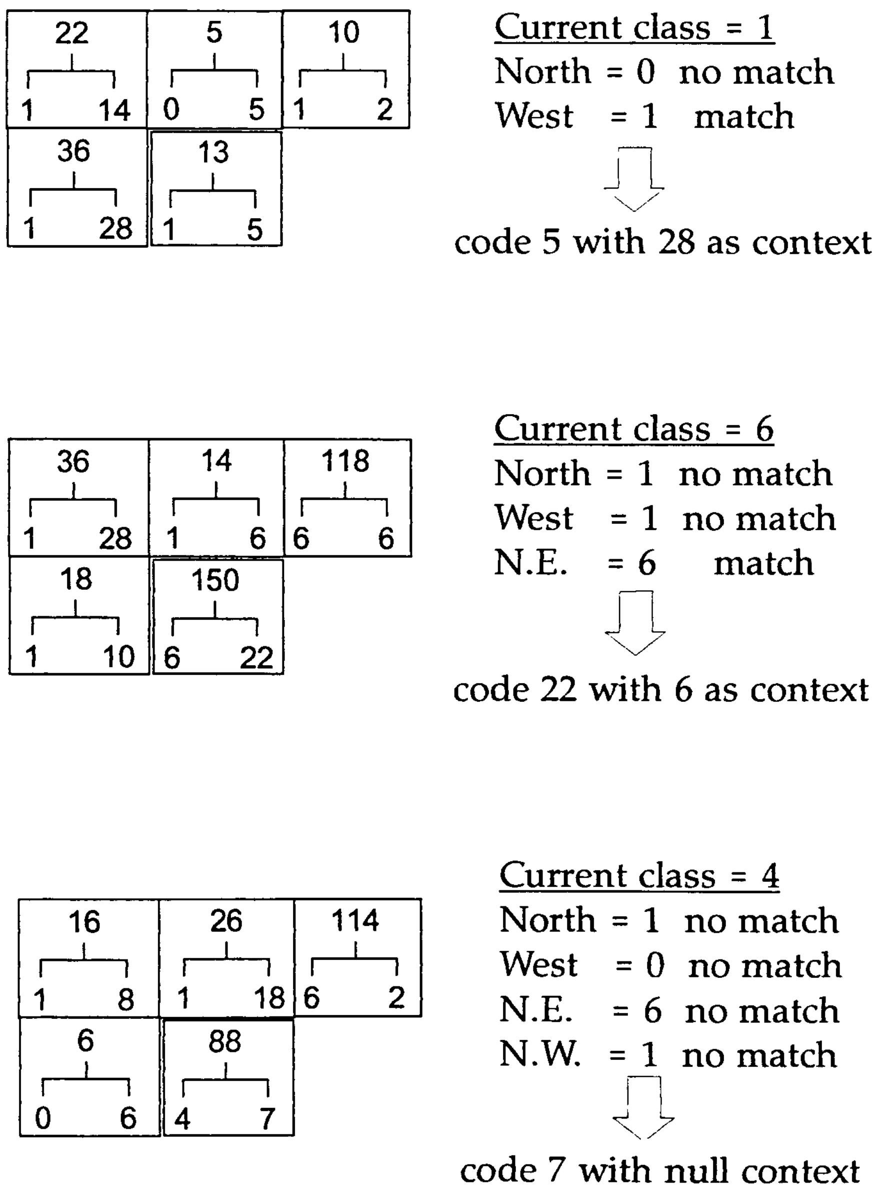


Figure 5.11 Examples of context selection for sub-codebook index

CHAPTER 6. RESULTS

This chapter presents the results obtained using the techniques discussed in Chapter 5. The images used in this experiment consisted of eight images from the USC database. These images have an intensity resolution of eight bits (256 gray values) and a spatial resolution 512x512 pixels. A block size of 4x4 was used resulting in a 128x128 array of indices to be compressed without loss. Of the eight images, five were selected as training images resulting in 81,920 training vectors for the design of the vector quantizer. Codebooks of size 128 and 256 were designed resulting in VQ bit rates of 0.4375 and 0.5000 respectively.

The classification statistics of the of training vectors are given in Table 6.1. It is apparent from this table that the vast majority of the vectors were classified as midrange or shade blocks. However, when designing the codebooks for the training data, it was observed that the number of codevectors necessary to sufficiently represent these classes was relatively low. For example, for the case of N=256, only 40 codevectors were used even though over 80% of the training vectors belonged to these classes. Additionally, the average distortion of the vectors belonging to these classes was much lower than that of the edge classes. For example, the average squared error of the vectors in the shade class was about 6 to 10 and in the midrange class the error was around 60 to 90. On the other hand, the average squared error in the edge classes was usually in the 200 to 300 range.

A separate sub-codebook was generated for each of the 30 classes and the resulting codebooks were merged into a super-codebook. Figure 6.1 shows an example of a super-codebook designed for a codebook size of 256. Numbering the blocks from left to right and top to bottom starting at zero gives the actual VQ index corresponding to each block. This can be compared to the VQ indices listed in Figure 5.8 to determine the classification for each of the blocks. The shade vectors appear to be very uniform as expected. The midrange blocks have some moderate gradient,

Table 6.1 Classification statistics for training data

Classification	Number of Vectors	Percentage
Shade	10367	12.66
Midrange/Mixed	56120	68.51
+Horizontal 1	863	1.05
-Horizontal 1	929	1.13
+Horizontal 2	615	0.75
-Horizontal 2	675	0.82
+Horizontal 3	634	0.77
-Horizontal 3	580	0.71
+Vertical 1	1161	1.42
-Vertical 1	1099	1.34
+Vertical 2	739	0.90
-Vertical 2	743	0.91
+Vertical 3	677	0.83
-Vertical 3	821	1.00
+45 degrees 1	566	0.69
-45 degrees 1	574	0.70
+45 degrees 2	303	0.37
-45 degrees 2	388	0.47
+45 degrees 3	238	0.29
-45 degrees 3	282	0.34
+45 degrees 4	317	0.39
-45 degrees 4	377	0.46
+135 degrees 1	575	0.70
-135 degrees 1	523	0.64
+135 degrees 2	342	0.42
-135 degrees 2	340	0.42
+135 degrees 3	268	0.33
-135 degrees 3	236	0.29
+135 degrees 4	278	0.34
-135 degrees 4	290	0.35

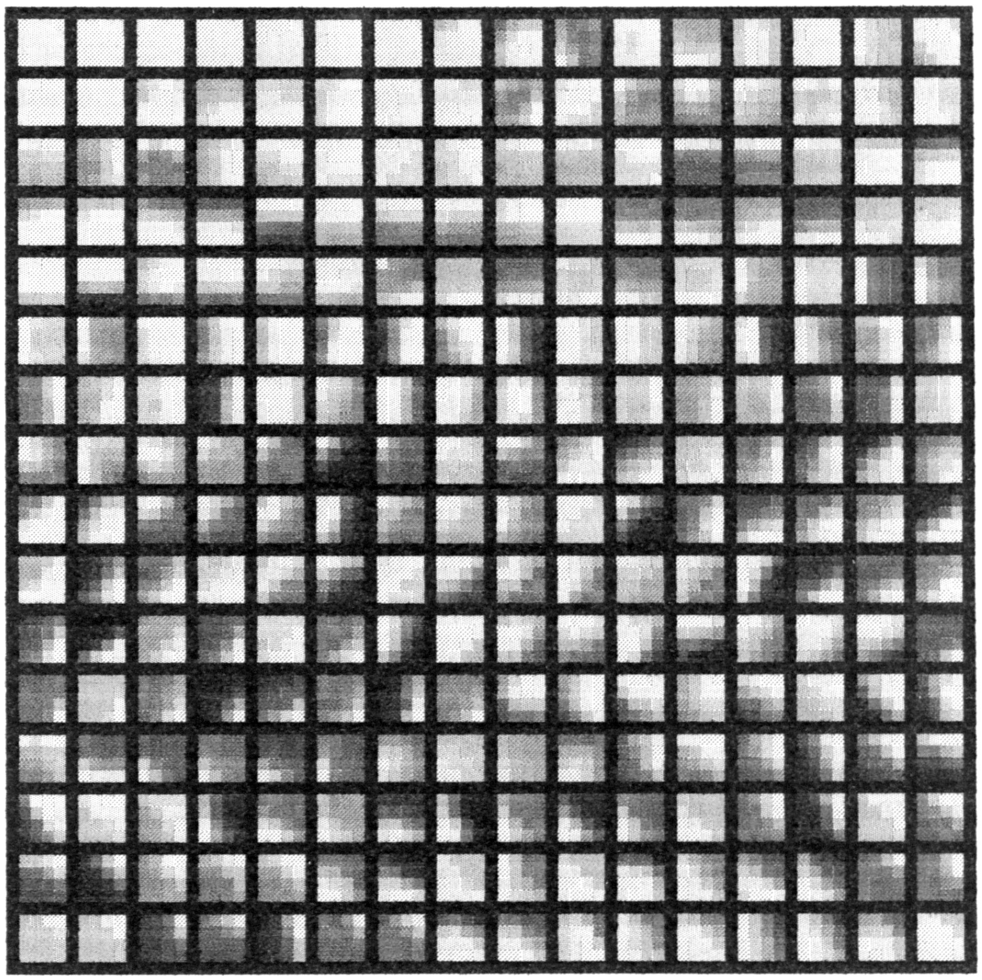


Figure 6.1 Example of classified codebook (N=256)

however, some of the blocks appear to have some moderate edge content. This may be the result of grouping the mixed blocks with the midrange blocks. The edge blocks show edges at appropriate locations and polarities with varying degrees of sharpness. Some of the diagonal blocks, however, appear to have more of vertical or horizontal character than diagonal. However, overall, the codebook represents the edge classes well.

The compression results for the two codebook sizes are summarized in Tables 6.2 and 6.3 for several images inside and outside the training set. The first row in each column represents the peak signal to noise ratio (PSNR) defined as

$$PSNR = 10\log_{10}\left(\frac{P^2}{MSE}\right) \text{ dB} \qquad \text{Equation 6.1}$$

where P represents the peak gray value and MSE is the mean square error of the reconstructed image. For an eight bit intensity resolution, P is 255. The PSNR represents a measure of the quality of the reconstructed image. A PSNR of near 30 dB is generally considered to be of communications quality. The range of PSNR obtained here is typically 28 to 31 dB for N=256 and 27 to 30 dB for N=128. The PSNR for the 128 size codebook is generally about 1 dB less than that of the 256 size codebook.

Figures 6.2 through 6.10 demonstrate the quality obtained for three of the images used in this experiment. Figures 6.2, 6.5 and 6.8 are the original images, Figures 6.3, 6.6 and 6.9 are the images coded at 0.5000 bpp, and Figures 6.4, 6.7 and 6.10 are the images coded at 0.4375 bpp.

Table 6.2 Compression results for N=128

Image	Peak SNR (dB)	Vector Quantization (bpp)	CR	Memoryless (bpp)	CR	First Order (bpp)	CR	Two Step (bpp)	CR
Lena	28.9	0.4375	18.3	0.283	28.3	0.250	32.0	0.241	33.2
Peppers	30.2	0.4375	18.3	0.256	31.3	0.233	34.3	0.221	36.2
Couple	25.9	0.4375	18.3	0.311	25.7	0.298	26.8	0.281	28.5
Sailboat	27.1	0.4375	18.3	0.327	24.5	0.315	25.4	0.299	26.8
Tiffany	27.5	0.4375	18.3	0.273	29.3	0.253	31.6	0.243	32.9
Woman	33.5	0.4375	18.3	0.223	35.9	0.189	42.3	0.177	45.2

Table 6.3 Compression results for N=256

Image	Peak SNR (dB)	Vector Quantization (bpp)	Vector Quantization CR	Memoryless (bpp)	Memoryless CR	First Order (bpp)	First Order CR	Two Step (bpp)	Two Step CR
Lena	30.0	0.5000	16.0	0.344	23.3	0.312	25.6	0.297	26.9
Peppers	31.3	0.5000	16.0	0.315	25.4	0.295	27.1	0.277	28.9
Couple	26.9	0.5000	16.0	0.373	21.4	0.366	21.9	0.340	23.5
Sailboat	28.1	0.5000	16.0	0.386	20.7	0.378	21.2	0.356	22.5
Tiffany	28.4	0.5000	16.0	0.338	23.7	0.323	24.8	0.307	26.1
Woman	35.1	0.5000	16.0	0.279	28.7	0.247	32.4	0.228	35.1

Of these three images, two (Peppers and Lena) are from outside the training set and the other (Sailboat) is from inside the training set. When comparing the original images to the compressed images it can be seen that some moderate amount of degradation has occurred. The relatively smooth areas such as the middle portion of the peppers in Peppers, the facial and back regions of Lena and the water and sky regions of Sailboat were reconstructed well using vector quantization. Much of the perceived degradation in these images is in the edge regions despite the large number of codevectors reserved for edge preservation. This degradation is most noticeable along the edges of the peppers in Peppers, edges along the brim of the hat and the shoulder and hair in Lena, and the edges along the sail in Sailboat. At larger scales, some blocking distortion is apparent that is an artifact of the vector quantization process. There is very little perceptual difference between the reconstructed images for the 128 and 256 codebooks sizes.

The remaining columns in Tables 6.2 and 6.3 are the bit rates and compression ratios for the various lossless techniques. The second and third column represent the nominal bit rate due to VQ and the corresponding compression ratio respectively. The bit rate for this case is

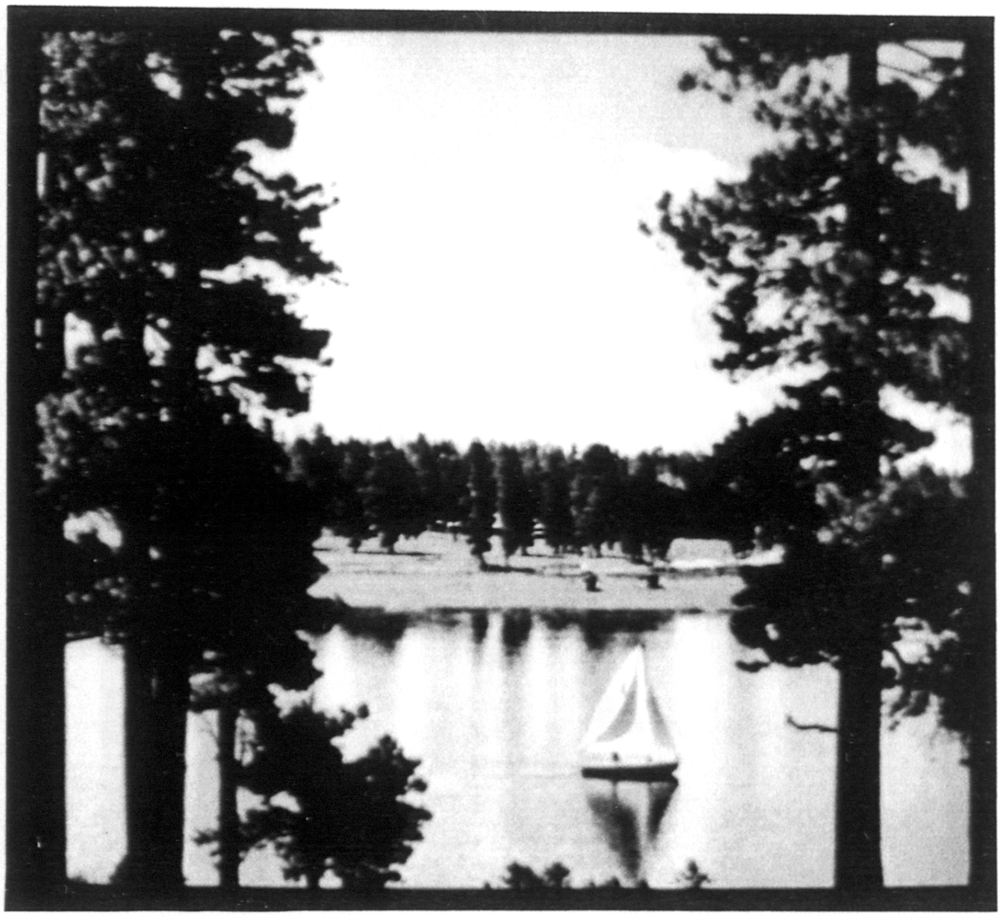


Figure 6.2 Sailboat -- Original image at 8 bpp

determined by Equation 2.2 where N is 128 or 256 and k is 16. The bit rate can be converted to a compression ratio by CR = 8/(bit rate). For the cases where the lossless compression was used, the bit rate was determined by dividing the total number of bits by the total number of pixels to obtain an average bit rate.

In general, the compression achieved by the lossless methods depends on the image being compressed. Most of the compression that is

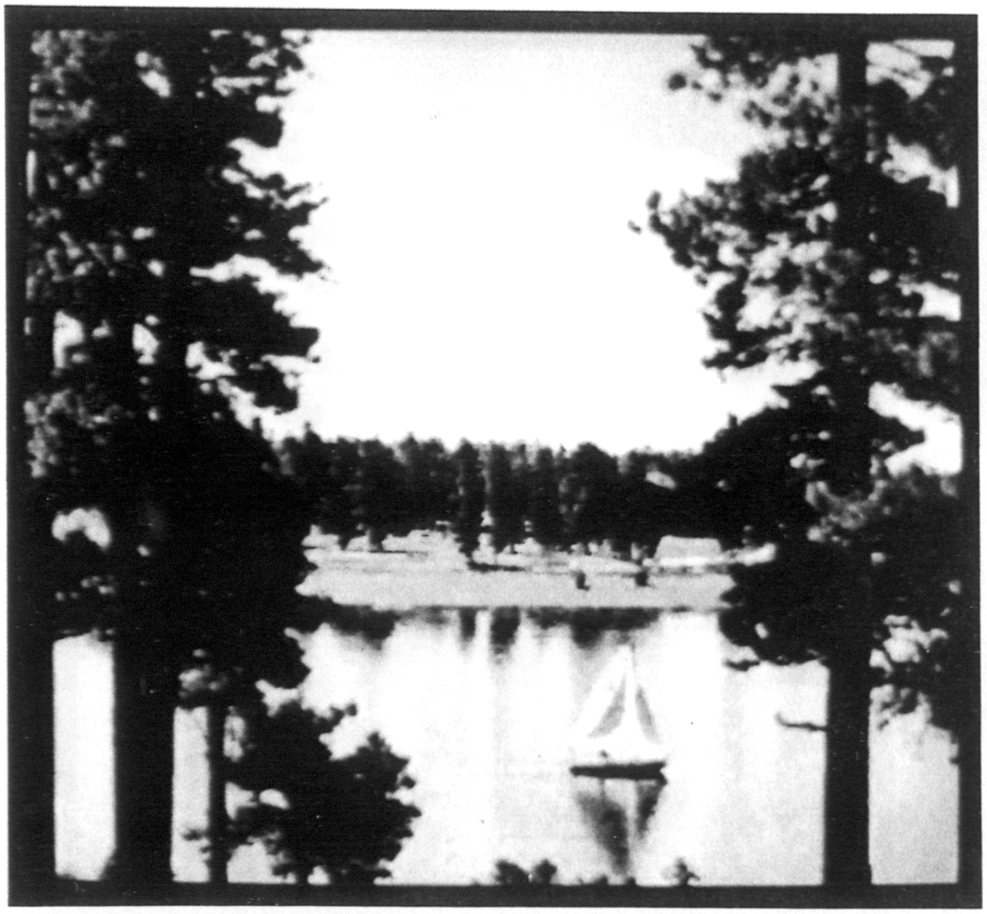


Figure 6.3 Sailboat -- Coded at 0.5000 bpp

obtained is due to the memoryless scheme. The memoryless scheme gathers its statistics adaptively as described in Chapter 4 using a value of K_i=10. The improvement obtained over the nominal VQ bit rate ranges from 20 to 45 percent for N=256 and from 25 to 50 percent for N=128. This improvement is due to the non-uniform distribution of the VQ indices. The first order scheme gives an improvement over the memoryless scheme by exploiting the correlations that exist between a block and its north neighbor. Again, the statistics were gathered

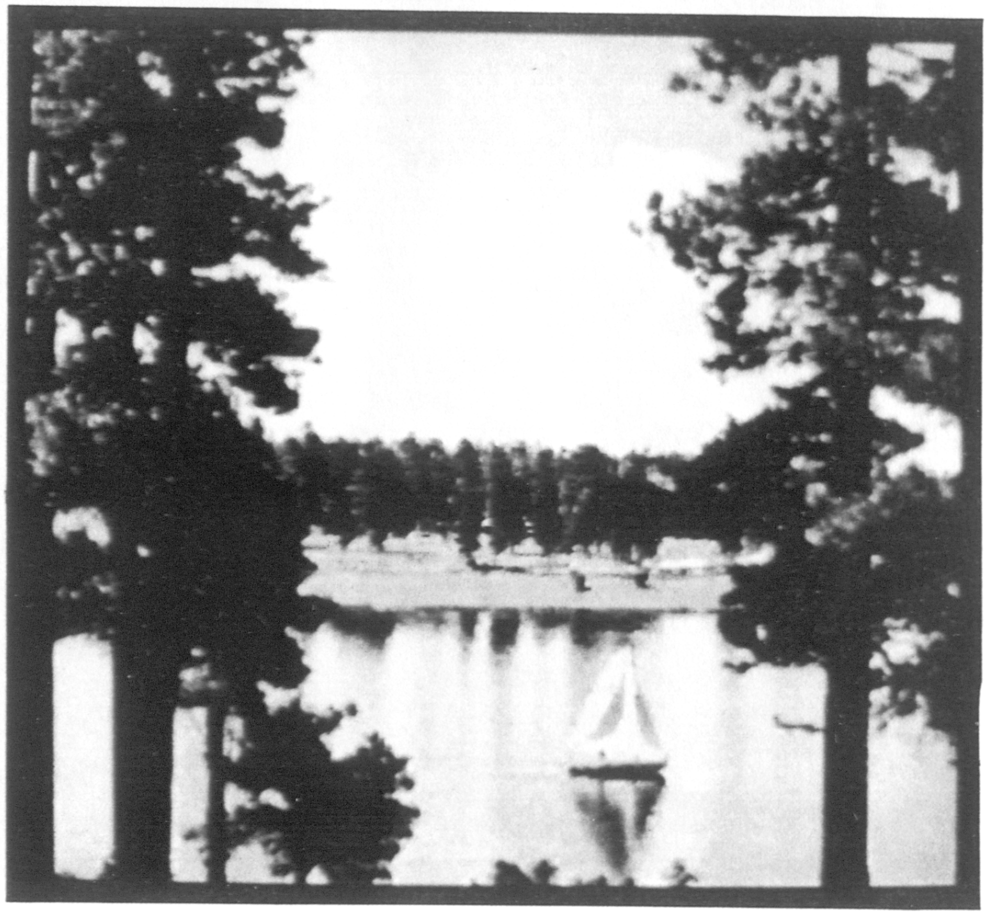


Figure 6.4 Sailboat -- Coded at 0.4375 bpp

adaptively with $K_i = 10$. This scheme improves over the memoryless approach by 2 to 12 percent for N=256 and 4 to 15 percent for N=128. In general, images that are relatively smooth seem to provide a larger performance improvement than more detailed images. The two step approach offers additional improvement over the memoryless approach by exploiting correlations in several of the causal neighbors. For this approach, values of $N_c = 32$, $K_t = 8$ and $K_i = 8$ were used to compress the classification information and $K_i = 2$ was used with the first order

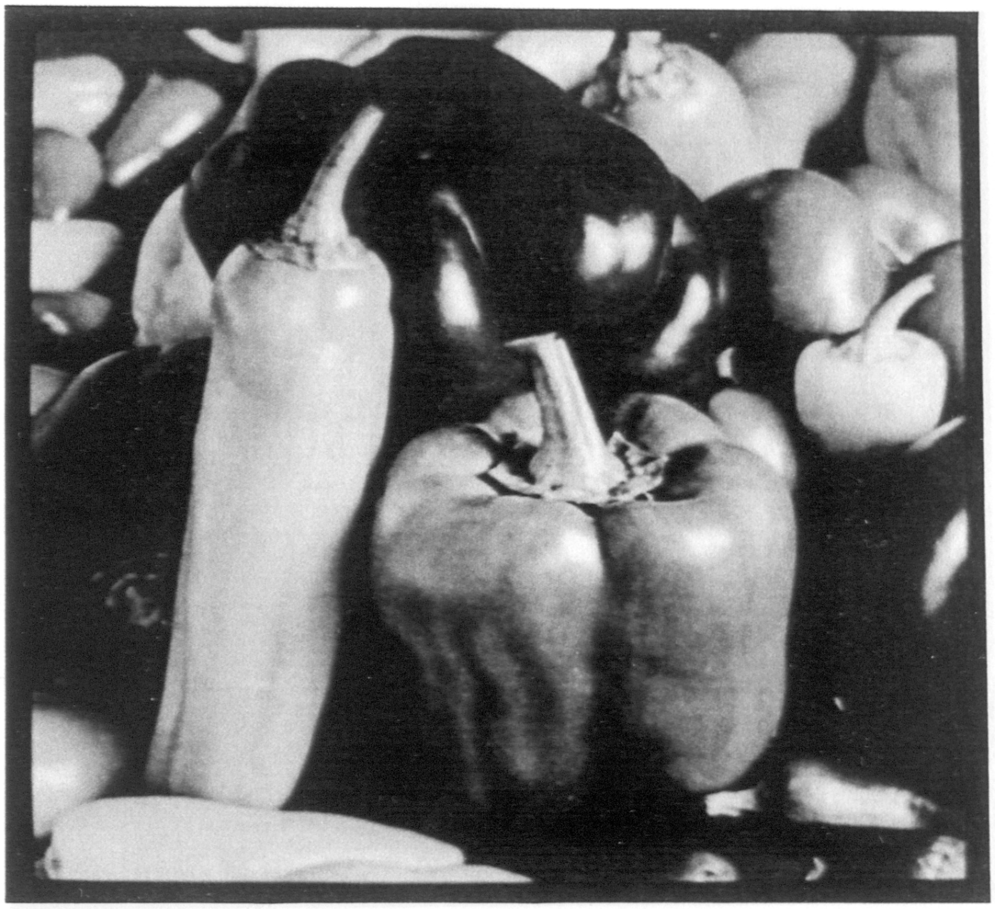


Figure 6.5 Peppers -- Original image at 8 bpp

over the memoryless scheme by 8 to compression of the sub-codebook indices. The two step approach improves 18 percent for N=256 and 9 to 21 percent for N=128.

In terms of physical storage size, the original uncompressed images each require 262,144 bytes of storage space since one byte is stored for each pixel location. The VQ process alone reduces the storage space to 14336 bytes for a codebook size of 128 and 16384 bytes for a codebook size

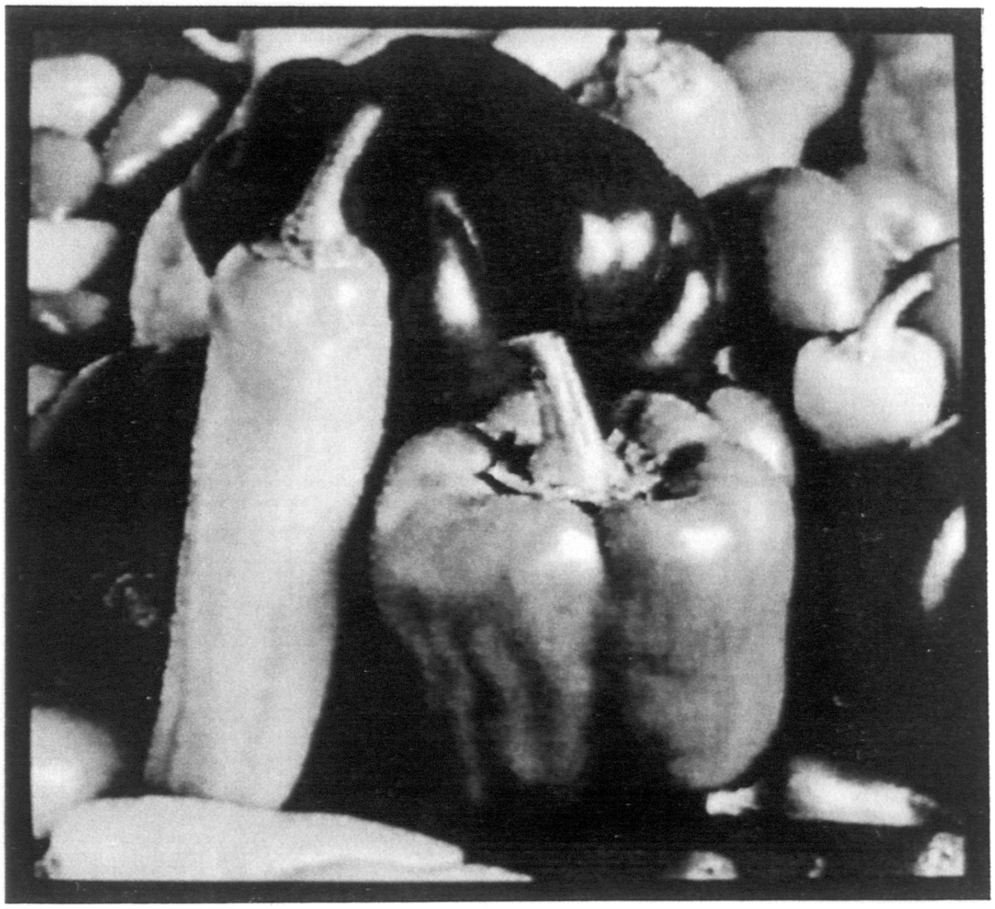


Figure 6.6 Peppers -- Coded at 0.5000 bpp

of 256. After compression using the two step method, the storage space is on average about 8000 bytes for the 128 vector codebook and about 9900 bytes for the 256 size codebook. Overall, the original images have been compressed from 256 kbytes down to a mere 8 or 9 kbytes.

The performance of the lossless methods described here is comparable to the performance of other VQ methods employing memory such as finite state VQ and address VQ. Although a direct relationship

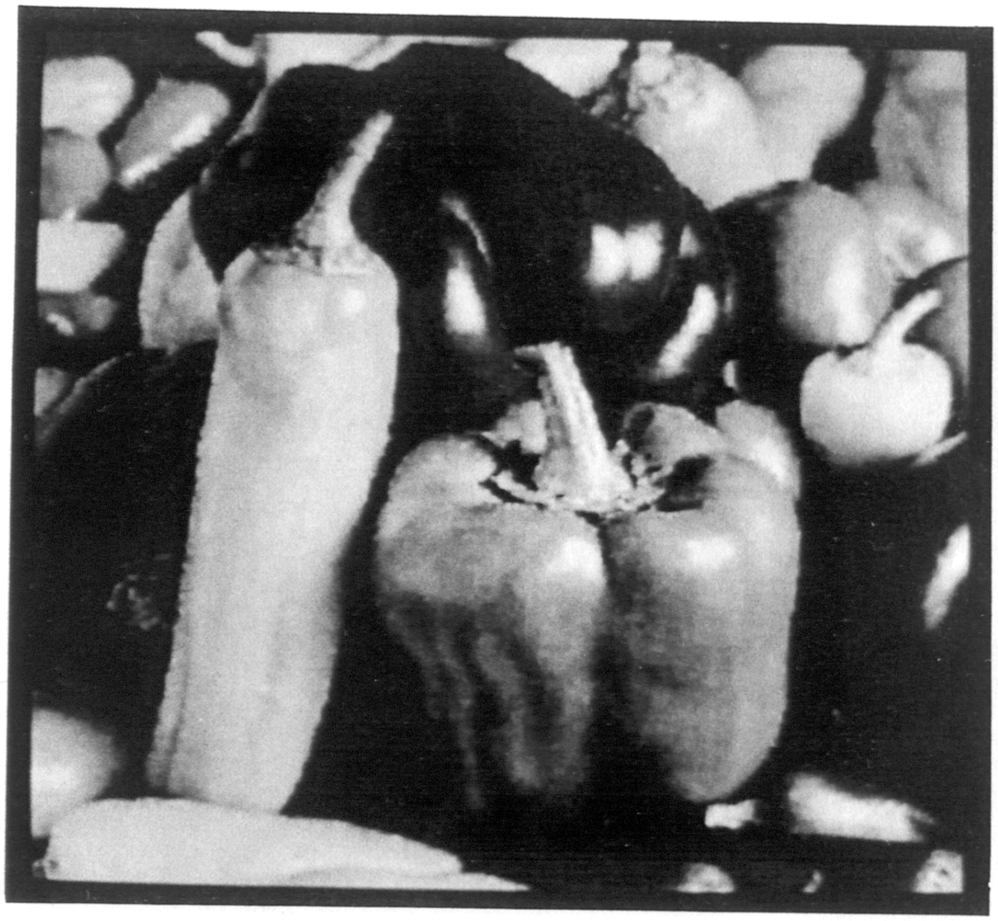


Figure 6.7 Peppers -- Coded at 0.4375 bpp

between the finite state method and the methods presented here is difficult to find, the overall performance of the lossless methods is comparable. In [4], finite state methods were used with 5x5 blocks and a codebook size of 128 to compress "Lena" to 0.24 bpp with a PSNR of 27.5 dB. This can be compared to the two step method where this image was compressed to the same bit rate with a PSNR of 28.9 dB. The image "Couple" was also compressed to 0.32 bpp at a PSNR of about 26 dB where the two step method provided a PSNR of 25.9 dB at a bit rate of

Figure 6.8 Lena -- Original image at 8 bpp

0.281 bpp. The image "Lena" was also compressed in [5] with a PSNR of 30 dB at a rate of 0.25 bpp.

A better comparison of the performance of the lossless methods can be made with respect to address VQ [6]. This is due to the observation that address VQ is essentially a vector quantizer with lossless compression of neighboring indices. In address VQ, images were compressed using 4x4 blocks and a VQ codebook size of 128. The total

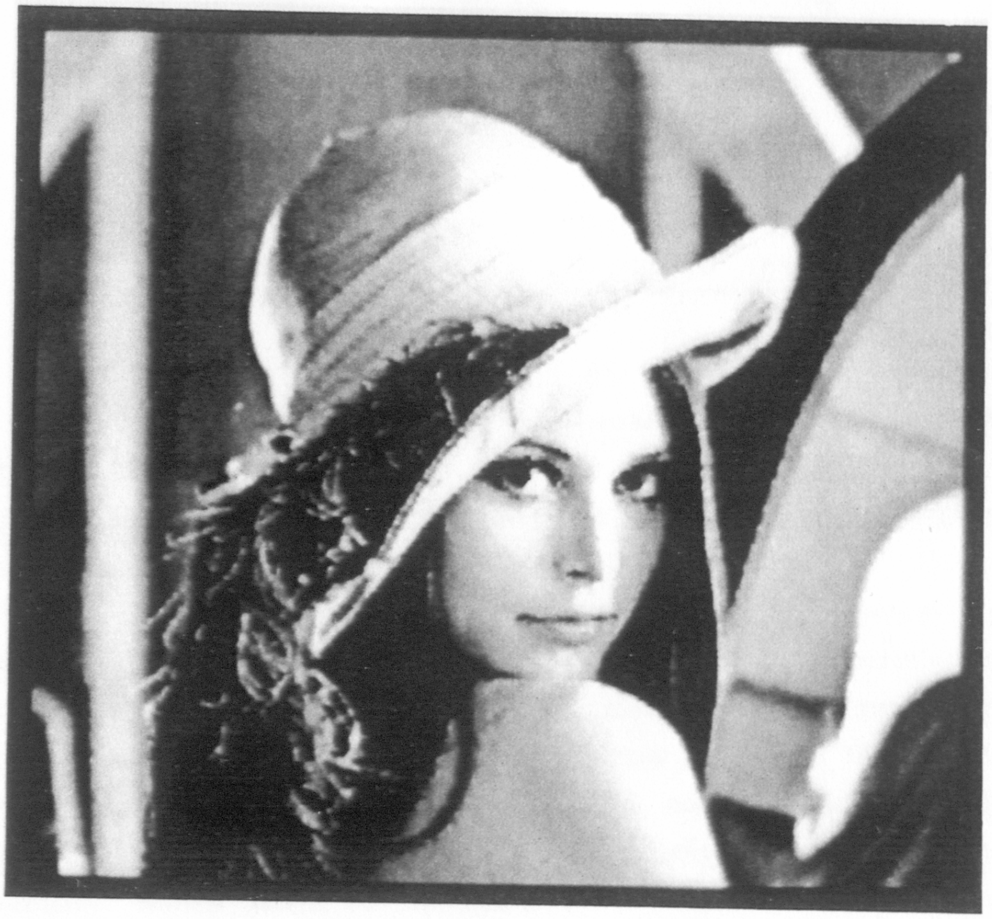


Figure 6.9 Lena -- Coded at 0.5000 bpp

addressable codebook was of size 1024 and about 100,000 address combinations were maintained. A summary of the results for four of the USC images is given in Table 6.4. In their experiments, "Lena" and "Peppers" were outside the training set while the other two images were inside the training set. Comparing the results with those given in Table 6.2 for the same images, it is apparent that the compression performance for images outside of the training set is better for the two step method presented here. Images inside of the training set, however, did not

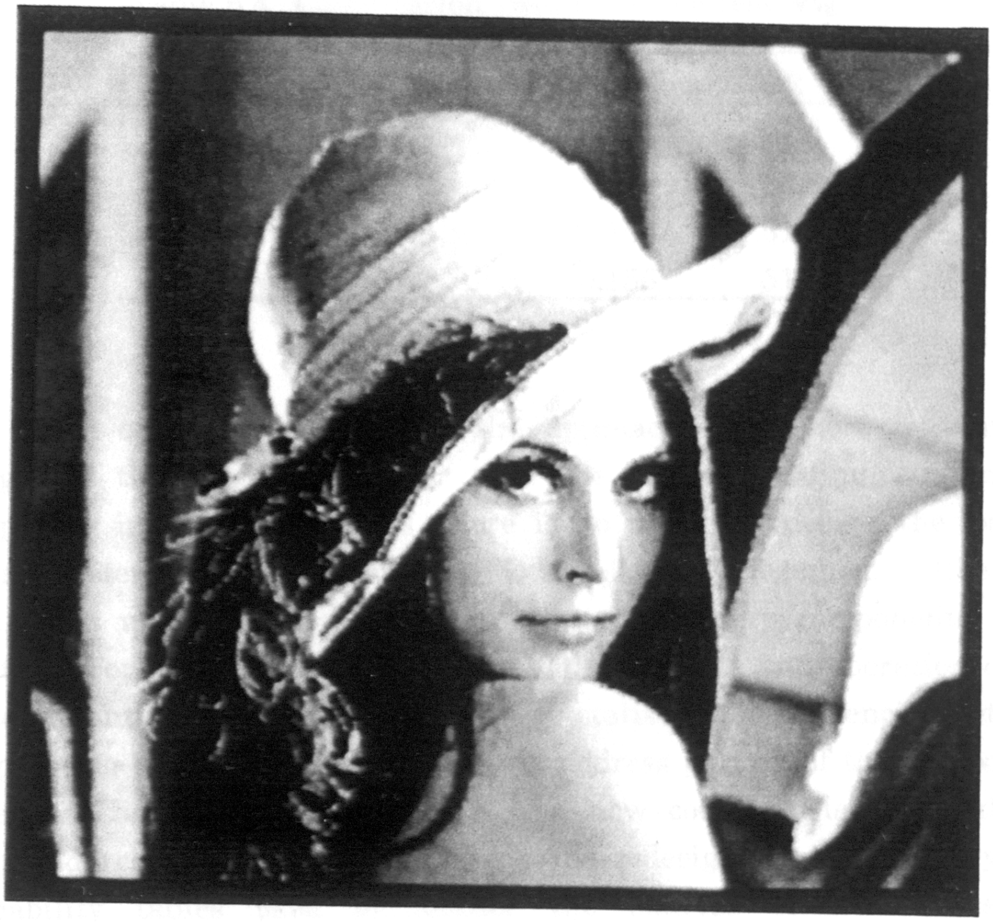


Figure 6.10 Lena -- Coded at 0.4375 bpp

compress as well as in the address VQ method. This is due to the fact that the address codebook and the probability tables used in address VQ are created from the training data. Thus, images inside of the training set will generally compress much better than images outside the training set. The variations in the PSNRs of these images are most likely due to the use of different training vectors or possibly modifications in the classification algorithm.

Table 6.4 Compression results for address VQ

Image	PSNR, dB	Bit Rate, bpp	CR
Lena	30.6	0.256	31.3
Peppers	29.8	0.260	30.8
Sailboat	26.5	0.156	51.3
Tiffany	31.9	0.156	51.3

One of the advantages to the approach given here is that the design of the vector quantizer and the lossless coding system are completely independent. Thus, there is no need to redesign the lossless coding system every time a new VQ codebook is designed. Other VQ techniques using memory have the VQ and memory components inter-related. For example, in finite state VQ the design and operation of the vector quantizer are modified to exploit statistical dependencies between neighboring blocks. In address VQ, the address portion of the codebook is dependent on the VQ codebook so if a new codebook is designed, the entire address codebook would have to be redesigned. Additionally, the probability tables used to reorder the address codebook must be recalculated if new training data are used.

Another advantage to separating the VQ and the lossless compression stages is that the storage and time complexity is small compared to methods such as address VQ. The storage complexities of the lossless methods presented here as well as the address VQ method are given in Table 6.5 for a codebook size of 128. The memoryless scheme offers a large compression performance for only a modest degree of storage space. Only the probability of each index needs to be stored in this case. In the first order approach, a slight compression improvement is obtained for a large increase in storage requirement. In this case, a separate probability table must be maintained for each index in the codebook. Thus, the storage complexity is the square of the codebook

Table 6.5 Storage complexity for lossless approaches

Compression Method	Storage Complexity
Memoryless	128
First Order	16384
Two Step	2262
Address VQ	165536

size. The two step approach offers a larger degree of improvement over the memoryless approach than does the first order case, yet requires much less storage. A comparison with the address VQ in the last row of Table 6.5 highlights the complexity improvement obtained using any of the lossless approaches given here. Address VQ requires storage of 100,000 address combinations as well as four probability tables, each of size 128^2. All of the values in Table 6.5 are concerned with only the lossless portion of the storage complexity. For a codebook size of 128, there are an additional 128*16 = 2048 storage locations needed to store the codebook.

CHAPTER 7. CONCLUSIONS

The need for data compression has lead to the development of a number of promising techniques for both lossless and lossy compression. In Chapter 2, the technique of vector quantization was discussed as a promising approach to lossy signal compression. Very high compression factors can be obtained using VQ while maintaining good reproduction quality. VQ is able to achieve high compression performance by successfully exploiting the statistical dependencies known to exist between samples of most real world signals. This ability to exploit inter-sample correlations comes from the fact that VQ has the freedom to choose the sizes and shapes of the vector space partitions to suit the statistical nature of the signals being compressed. Another advantage to using VQ is that the decoding is simply a table look-up procedure. This makes VQ useful for applications that require data to be decompressed many times but only compressed once.

In general, the performance of VQ will improve as the vector dimension is increased. However, unconstrained VQ is severely limited by the computational complexity of even modest vector sizes. Due to this complexity barrier, a number of techniques have been developed that place some restriction on the vector quantizer in exchange for a reduction in the complexity. Many of these techniques substantially reduce the computational complexity while providing only slightly sub-optimal performance.

Since VQ is limited to small block sizes, there is generally some statistical dependency that remains between adjacent VQ blocks. Methods that are commonly used to exploit these correlations involve introducing memory into the vector quantizer. These methods often provide greater compression efficiency but in some cases, such as address VQ, require substantial storage and computational complexity.

Another approach to removing redundancies that exist between adjacent VQ indices is to separately code the signal using VQ and a lossless coding scheme. In this thesis, the arithmetic coding technique was used with several different source models to exploit inter-block correlations in images coded by VQ. Arithmetic coding offers an efficient coding solution while easily allowing source modeling and adaptive statistic estimations. Arithmetic coding, on its own provides data reduction by exploiting the non-uniform symbol distributions that generally exist in VQ indices. When proper source models are used with arithmetic coding, further data reduction can be realized since correlations between adjacent blocks can be exploited.

Experiments using the VQ and lossless techniques were performed using standard images from the USC database. Several different source models were used in the lossless stage. Much of the improvement in compression performance was due simply to the non-uniform distribution of the VQ indices. The first order model improved the compression performance to some extent but also increased the complexity substantially. The two step procedure offered even greater compression performance while requiring much less storage space than the first order approach. The first order method requires over seven times the storage space of the two step method for a codebook of size 128. The overall compression ranged from 20 to 35 times for a codebook size of 256 and 25 to 45 times for the codebook size of 128.

A comparison with address VQ shows that the compression obtained for images outside of the training set is better for the two step lossless scheme. Images inside the training set, however, did not compare as well. This is due to the fact that address VQ makes extensive use of the training data to calculate its statistics. Therefore, images that belong to the training data will, in general, compress very well using this approach. The lossless scheme, however, gathers its statistics adaptively from the image being compressed so no *a priori* knowledge of the source statistics is necessary. This means that, in general, images from outside the training set will compress as well as images inside the training set.

A major advantage of using this technique over address VQ is the storage and time complexity improvement. In addition to the VQ codebook, address VQ requires storage of 100,000 address combinations. Additionally, four probability tables must be stored, each of size N^2. On the other hand, the two step lossless scheme using the same size codebook requires storage of only 2262 probability estimations. In terms of time complexity, the address VQ must search a portion of the address codebook, calculate a new probability score for each address entry, and reorder the entire address codebook after every four VQ blocks are coded. The two step method requires a relatively small amount of time to separate the class and sub-codebook indices, determine the contexts and code the block based on the supplied probabilities.

In addition to the storage and time considerations, the two step approach is completely independent of the VQ design so that new codebooks can be generated without the need to redesign the lossless system. In address VQ, however, the design of the address codebook and the probability tables must be performed every time a new VQ codebook is created.

Future Work

There are a number of ways in which the methods discussed in this thesis can be improved upon for future work. Most of the improvements that are suggested here are concerned with the improvement of the image quality for a given bit rate. There are also some suggestions given to improve the compression performance for a given quality level. In general, however, both types of suggestions have the same overall effect since techniques that improve image quality can use reduced bit rates while maintaining the previous level of image quality.

The area that promises the greatest potential for improvement is the classified vector quantizer. One of the difficulties with the classifier used here is in the classification of the edges. The classification process described in Chapter 5 uses simple gradient counters to determine if an edge is present. If the edge counters reach a predetermined threshold, the block is classified as a specific edge type and the gradient table associated with that edge class is searched. When searching the gradient table, the number of gradients exceeding the threshold is counted for each row or column and the location yielding the largest count is selected. The problem here is that the counters used to determine the classification do not consider whether the gradients lie in the same row (or column). For example, a positive vertical edge will be detected if the H_p counter is greater than one. This, however, does not imply that these gradients will lie in the same column of the gradient table. Thus, a decision based solely on the counters does not guarantee proper classification of the edges. Vectors that have very little edge content may be classified as edges. A possible solution to this problem is to maintain a counter for each of the possible edge locations. These counters could then be used to detect when an edge occurs at a specific location. The use of separate counters for each edge location ensures that edge classification only occurs for blocks with definite edge content.

Another problem with the edge classification is that there appears to be some overlap between various edge classes. Many of the edge codevectors appear to have their edges spread over more than one location. These edges do not occur abruptly but are more gradual. Thus, some of the vectors in one location may appear very similar to vectors in an adjacent location. This implies that better performance could be obtained by removing the classification by location since the VQ would be less restricted in choosing codevectors. This would result in the same ten classes used by the two step lossless method. This would of course increase the computational complexity since each edge vector would have to be compared to more codevectors. However, the overall increase in computational complexity would be small since the percentage of edge blocks in an image is generally small.

The classification of the mixed class also presents a problem for the vector quantizer. Designing a separate codebook for the mixed class is difficult since the vectors belonging to that class do not generally possess any common feature. Thus, a large number of vectors must be allocated to this class in order to obtain good reconstruction. The solution proposed in [7] and used in this research was to group the mixed class with the midrange class. The reason for this approach is that the percentage of image block classified as midrange is large and the effect of the mixed class would be small. However, when observing the codebook in Figure 6.1, it is apparent that many of the midrange codevectors contain some degree of edge content. It is also apparent that some of the codevectors in the midrange class appear to be similar to horizontal, vertical and diagonal codevectors as well as mixed edge content. These edges are probably due to the influence of the mixed class.

A possible solution to this problem would be to completely ignore the mixed classification for codebook design purposes. This would allow the midrange class to be designed without the influence of the edges of the mixed class which will have the effect of reducing the number of codevectors necessary to adequately represent this class. The vectors that are no longer used by the midrange class could then be distributed among the various edge classes resulting in better reconstruction of the edges. When coding vectors that belong to the mixed class, the entire codebook could be searched for the best matching codevector. This would allow the VQ encoder to find a relatively good match for the mixed blocks without the need for a separate codebook. Again, this approach would have the effect of increasing the computational complexity since an exhaustive search of the codebook would be required for mixed blocks. However, the number of blocks belonging to the mixed class is relatively small so the overall increase in coding time would be small. Also, the number of codevectors that must be searched for the midrange class is reduced resulting in a reduction of the computational complexity for that class.

This would most likely negate the computational increase due to the mixed class.

Some improvements can also be made to the lossless coding part of the compression system. The arithmetic coding technique provides performance near the theoretical compression bound so most of the gain in compression from the lossless coder will be due to improved source modeling. In general, the more information that is known about how the source creates its data, the better the compression will be. Thus, the compression could be improved by using more sophisticated source models. This, however, generally leads to the need for large amounts of storage and the need for larger data sets for good adaptation. Therefore, there is a need for more sophisticated source models that do not require a large amount of storage. There appears to be no obvious solution to this problem. The best approach would involve a trial and error design of various source models. The two step method presented here shows how the complexity of the model can be reduced while exploiting the redundancies between more neighbors. Only the most probable contexts are used in this case resulting in a large reduction in complexity while retaining some of the benefit of using a higher order model. Also, the decomposition of the VQ index into classification and sub-codebook components reduces the complexity of the model. Some modest improvement in performance may be obtained by trying different decompositions.

Although there appears to be no obvious way to improve the lossless coding by directly modifying the source models, there may be a way of enhancing the performance of the lossless coder by altering the design of the vector quantizer. When experimenting with different codebooks, it was observed that certain codebook designs provided better compression results than others. This suggests that there may be a specific way to design the vector quantizer to enhance the performance of the lossless coder. One possible way that the VQ design can be modified is to select an initial codebook based on how the source model operates.

For example, choosing initial codevectors that somehow promote frequently occurring neighbor combinations. It may also be possible to modify the LBG algorithm so that it considers neighboring vectors during the design process.

REFERENCES

[1] A. Gersho and R.M. Gray, "Vector Quantization and Signal Compression," *Kluwer Academic Publishers*, Boston, MA, 1991.

[2] R.M. Gray, "Vector Quantization," *IEEE ASSP Magazine*, pp. 4-29, April 1984.

[3] N.M. Nasrabadi and R.A. King, "Image Coding Using Vector Quantization: A Review," *IEEE Transactions on Communications*, Vol. 36, pp. 957-971, August 1988.

[4] R. Aravind and A. Gersho, "Image Compression Based on Vector Quantization with Finite Memory," *Optical Engineering*, Vol. 26, pp. 570-580, July 1987.

[5] T. Kim, "New Finite State Vector Quantizers for Images," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 1180-1183, 1988.

[6] N.M. Nasrabadi and Y. Feng, "Image Compression Using Address Vector Quantization," *IEEE Transactions on Communications*, Vol. 38, No. 12, pp. 2166-2173, December 1990.

[7] B. Ramamurthi and A. Gersho, "Classified Vector Quantization of Images," *IEEE Transactions on Communications*, Vol. 34, No. 11, pp. 1105-1115, November 1986.

[8] N. Abramson, "Information Theory and Coding," *McGraw-Hill*, New York, NY, 1963

[9] G.G. Langdon, Jr., "An Introduction to Arithmetic Coding," *IBM Journal of Research and Development*, Vol. 28, No. 2, pp 135-149, March 1984.

[10] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, Vol. 30 No. 6, pp. 520-540, June 1987.

[11] C.B. Jones, "An Efficient Coding System for Long Source Sequences," *IEEE Transactions on Information Theory*, Vol. 27, No. 3, pp 280-291, May 1981.

[12] T.C. Bell, J.G. Cleary, I.H. Witten, "Text Compression," *Prentice Hall*, Englewood Cliffs, NJ, 1990.

[13] R.N. Williams, "Adaptive Data Compression," *Kluwer Academic Publishers*, Boston, MA, 1991.

[14] T.V. Ramabadran, "Adaptive Source Models for Data Compression," Ph.D. Dissertation, University of Notre Dame, 1986

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. T.V. Ramabadran for his guidance and support while overseeing the work presented here. His encouragement and the motivation he inspired proved to be invaluable to the completion of this research. I would also like to thank Dr. Ramabadran for the use of his lossless compression programs that were adapted for use on this project. I would like to thank Dr. Satish Udpa and Dr. Suresh Kothari for serving as members of my committee. A special thanks is due to Dr. Satish Udpa for his help and guidance during the early portion of my graduate study and for helping me with my decision to pursue an advanced degree. I would also like to thank Jorn Lyseggen for the use of the image processing software he developed and for all the helpful hints he provided on the various software programs used to complete this project. Finally, I would like to thank all of the friends and faculty members who have provided words of encouragement and valuable suggestions on different aspects of this project.

APPENDIX

This appendix contains the source listings for the programs used in this project. There are five programs written in C that perform the VQ design and coding. The programs have been written for use on a IBM compatible PC but have been kept as general as possible to allow portability. The first program is the classification program which takes an input image and creates a file containing the classification for each block. The second program is the training set organization program that takes the input images listed in "trimage.fn" and sorts the blocks into appropriate training files. The third program is the codebook design program which reads the training files created by the previous program and designs a sub-codebook for a specified class. The fourth program merges all of the sub-codebooks created by the codebook design program into a single super-codebook. The fifth program is the VQ coding program which compresses the image into a file of VQ indices and creates a reconstructed version of the image.

In addition to the VQ programs, there are four programs that implement the various lossless compression schemes as well as the Jcode subroutine, which is used by all four programs, that implements the arithmetic coding technique described in this thesis. These programs were originally written by Dr. T.V. Ramabadran and were adapted for use here. None of the lossless programs produce output files, rather they simply count the size of the output file.

The VQ programs were written with respect to a specific directory structure. All of the executable files are in some root directory. All of the images are stored in a sub-directory "Img" with a ".img" extension for the original images and ".VQ" extension for reconstructed images. All of the classification files are stored in a "Classimg" sub-directory with a ".xx" extension. All of the separated training files are stored in a "Trdata" sub-directory with a ".tr" extension. All of the codebooks are stored in the sub-directory "Codebook" with a ".cb" extension. The codebook used for coding the image is stored in the file "cbook.cb". Finally, the VQ index files are stored in a "Coded" sub-directory with a ".idx" extension.

```
/**********************************************************************

                      VQ CLASSIFICATION PROGRAM

                    Written By: Mark Hetherington
                         Date: August 31, 1993


This program implements the classification algorithm described in the paper "Classified
   Vector Quantization of Images"  by Bhaskar Ramamurthi and Allen Gersho IEEE
               Transactions on Communications, Nov. 1986

     Output File Format:

            BINARY                           CLASS

              0                              shade
              1                               midrange/mixed
              2                              (not used, reserved for mixed)
          3 to (2*BLKSIZE)                   Horzp1, Horzn1, Horzp2, etc...
  (2*BLKSIZE+1) to (4*BLKSIZE-2)             Vertp1, Vertn1, Vertp2, etc...
   (4*BLKSIZE-1) to (8*BLKSIZE-10)           D45p1,  D45n1,  D45p2,  etc...
  (8*BLKSIZE-9) to (12*BLKSIZE-18)           D135p1, D135n1, D135p2, etc...

**********************************************************************/


#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>


#define  MAXBLK     8      /* Maximum Block size */
#define  MAXIMGW  696      /* Maximum Image Width */
```

```
/* Function Prototypes */
void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW);
void getblock(unsigned char A[MAXBLK][MAXIMGW], unsigned char
                                        x[MAXBLK][MAXBLK],int g);
int class (unsigned char x[MAXBLK][MAXBLK]);
int Horz(int Gv[MAXBLK-1][MAXBLK], int x);
int Vert(int Gh[MAXBLK][MAXBLK-1], int x);
int D45(int Gv[MAXBLK-1][MAXBLK], int Gh[MAXBLK][MAXBLK-1], int x);
int D135(int Gv[MAXBLK-1][MAXBLK], int Gh[MAXBLK][MAXBLK-1], int x);
int getmax (int z[2*MAXBLK-4]);
void fnextend(char fname[50], char path[30], char fn[10], char ext[5]);
void errout(char fname[50]);


/* Global Variables */
int BLKSIZE;            /* Block Size      */
int THL;                /* Line Threshold  */
int THS;                /* Shade Threshold */


void main(void)
{ FILE *outfile,*image;
  char fname[50],fn[10],ans[10],ftype[5];
  unsigned char A[MAXBLK][MAXIMGW],x[MAXBLK][MAXBLK];
  int i,j,m,N,M,GL,N1,M1,IDXW,IDXH;


/*   Variables: N       width of input image
                M       height of input image
                GL      Maximum gray level
                N1      adjusted image width
                M1      adjusted image height
                IDXW    width of index image
                IDXH    height of index image
                A[][]   row of blocks from image
                x[][]   current block
 */
```

```
/* Enter Block Size */
 printf("\n***** Classification Program *****\n\n");
 BLKSIZE = MAXBLK + 1;
 while (BLKSIZE>MAXBLK)
   { printf("Blocksize: ");
     scanf("%d",&BLKSIZE);
   }

 /* Set line and shade thresholds */
 THL = (BLKSIZE+1)/2;
 THS = BLKSIZE - 1;

 ans[0] = 'Y';
 while(toupper(ans[0])=='Y')
   {
     /* Open files */
     printf("\nWhat is the input image file? ");
     scanf("%s",fn);
     fnextend(fname,"Img\\",fn,".img");
     if ((image = fopen(fname,"rb"))==NULL) errout(fname);
     fscanf(image,"%s %d %d %d",ftype,&N,&M,&GL); fgetc(image);
     if (N>MAXIMGW)
       { printf("Image Width too large, %d > %d\n",N,MAXIMGW);
         exit(1);
       }
     fnextend(fname,"Classimg\\",fn,".xx");
     if ((outfile = fopen(fname,"wb"))==NULL) errout(fname);
     fputc(BLKSIZE,outfile);

     /* Initialize Parameters */
     IDXW = N/BLKSIZE;
     IDXH = M/BLKSIZE;
     N1 = IDXW*BLKSIZE;
     M1 = IDXH*BLKSIZE;
     if ((M!=M1)||(N!=N1))
       printf("Warning: Non-integral image dimensions (%d,%d) ==> (%d,%d)\n",
              M,N,M1,N1);
```

```
      /* Image classification loop */
      printf("\nClassifying");
      for (i=0; i<IDXH; i++)
        { getrow(image,A,N);     /* read row of blocks    */
          for (j=0; j<IDXW; j++)
            { getblock(A,x,j);   /* extract current block */
              m=class(x);        /* classify block        */
              fputc(m,outfile);  /* write classification  */
            }
        }

      /* Close files & return to top */
      fclose(outfile);
      fclose(image);
      printf("\ndone\n\nAnother file? ");
      scanf("%s",ans);
    }
}

int class (unsigned char x[MAXBLK][MAXBLK])

{  int i,j,sh,sv,Vp,Vn,Hp,Hn,Gv[MAXBLK-1][MAXBLK],
                                Gh[MAXBLK][MAXBLK-1];
   float davh,davv,dh,dv,Tsh,Tsv,Teh,Tev;

/* This subroutine classifies the input block "x" according to
   Ramamurthi and Gersho's Classification algorithm

   Variables:  Hp     Positive Horizontal Gradient counter,
               Hn     Negative Horizontal Gradient counter,
               Vp     Positive Vertical Gradient counter,
               Vn     Negative Vertical Gradient counter,
               sh     Horizontal Shade counter,
               sv     Vertical Shade counter,
               Gh     Horizontal Gradient location table,
               Gv     Vertical Gradient location table,
               davh   Average intensity in Horizontal direction,
               davv   Average intensity in Vertical direction,
               dh     horizontal gradient,
               dv     vertical gradient,
               Tsh    horizontal shade threshold,
                    Tsv     vertical shade threshold,
                    Teh     horizontal edge threshold,
                    Tev     vertical edge threshold.
*/

 Hp = Hn = Vp = Vn = sh = sv = 0;
 for (i=0; i<BLKSIZE; i++)
  for (j=0; j<(BLKSIZE-1); j++) {

   /* calculate average intensities and gradients */
   davh = ((int)x[i][j] + (int)x[i][j+1])/2.0;
   davv = ((int)x[j][i] + (int)x[j+1][i])/2.0;
   if (davh==0.0) davh = (float)0.5;
   if (davv==0.0) davv = (float)0.5;
   dh = ((int)x[i][j] - (int)x[i][j+1])/davh;
   dv = ((int)x[j][i] - (int)x[j+1][i])/davv;

   /* set shade thresholds */
   if ((davh>225.0)||(davh<30.0)) Tsh = (float)0.1;
   else Tsh = (float)0.025;
   if ((davv>225.0)||(davv<30.0)) Tsv = (float)0.1;
   else Tsv = (float)0.025;

   /* increment shade counters */
   if (fabs(dh)>Tsh) sh++;
   if (fabs(dv)>Tsv) sv++;

   /* set edge thresholds */
   if (davh>=30.0) Teh = (float)0.2;
  else Teh = 8.0/davh;
   if (davv>=30.0) Tev = (float)0.2;
   else Tev = 8.0/davv;

   /* fill in gradient tables and increment edge counters */
   if (dh>Teh)
    { Gh[i][j] = 1;
     Hp++;
    }
   else if (dh<-Teh)
    { Gh[i][j] = -1;
      Hn++;
      }
     else Gh[i][j] = 0;
     if (dv>Tev)
      { Gv[j][i] = 1;
        Vp++;
       }
     else if (dv<-Tev)
      { Gv[j][i] = -1;
        Vn++;
      }
     else Gv[j][i] = 0;
    }

/*** Classification ladder ***/
  if ((sh<THS)&&(sv<THS))
    return(0);                              /* shade */
  else if (((Vp>=THL)&&(Vn>=THL))||((Hp>=THL)&&(Hn>=THL)))
    return(2);                              /* mixed */
  else if ((Vp<THL)&&(Vn<THL)&&(Hp<THL)&&(Hn<THL))
    return(1);                              /* midrange */
  else if ((Vp>=THL)&&(Hp<THL)&&(Hn<THL))
    return(3+2*Horz(Gv,1));                 /* positive horizontal */
  else if ((Vn>=THL)&&(Hp<THL)&&(Hn<THL))
    return(4+2*Horz(Gv,-1));                /* negative horizontal */
  else if ((Hp>=THL)&&(Vp<THL)&&(Vn<THL))
    return(2*BLKSIZE+1+2*Vert(Gh,1));       /* positive vertical */
  else if ((Hn>=THL)&&(Vp<THL)&&(Vn<THL))
    return(2*BLKSIZE+2+2*Vert(Gh,-1));      /* negative vertical */
  else if ((Vp>=THL)&&(Hp>=THL))
    return(4*BLKSIZE-1+2*D45(Gv,Gh,1));     /* positive 45 */
  else if ((Vn>=THL)&&(Hn>=THL))
    return(4*BLKSIZE+2*D45(Gv,Gh,-1));      /* negative 45 */
  else if ((Vp>=THL)&&(Hn>=THL))
    return(8*BLKSIZE-9+2*D135(Gv,Gh,1));    /* positive 135 */
  else if ((Vn>=THL)&&(Hp>=THL))
    return(8*BLKSIZE-8+2*D135(Gv,Gh,-1));   /* negative 135 */
  printf("Error: No classification\n");
  exit(1);
}
```

```
void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW)
{ int i,j;

 /* This subroutine gets a row of blocks from image file */

 for (i=0; i<BLKSIZE; i++)
  for (j=0; j<IMGW; j++)
   A[i][j] = fgetc(image);

}

void getblock(unsigned char A[MAXBLK][MAXIMGW],
                    unsigned char x[MAXBLK][MAXBLK],int g)
{  int i,j;

  /* This subroutine gets the gth block from the array A[][] */

  for (i=0; i<BLKSIZE; i++)
   for (j=0; j<BLKSIZE; j++)
    x[i][j] = A[i][g*BLKSIZE+j];
 }

int Horz(int Gv[MAXBLK-1][MAXBLK], int x)
{  /* This subroutine finds the location of the horizontal line */

  int i,j;
  int z[2*MAXBLK-4];

  for (i=0; i<(2*BLKSIZE-4); i++) z[i] = 0;
  for (i=0; i<(BLKSIZE-1); i++)
   for (j=0; j<BLKSIZE; j++)
    if (Gv[i][j] == x) z[i]++;
  return(getmax(z));
 }

int Vert(int Gh[MAXBLK][MAXBLK-1], int x)
{  /* This subroutine finds the location of the vertical line */

  int i,j;
  int z[2*MAXBLK-4];
  for (i=0; i<(2*BLKSIZE-4); i++) z[i] = 0;
  for (i=0; i<(BLKSIZE-1); i++)
   for (j=0; j<BLKSIZE; j++)
    if (Gh[j][i] == x) z[i]++;
   return(getmax(z));
}

int D45(int Gv[MAXBLK-1][MAXBLK], int Gh[MAXBLK][MAXBLK-1], int x)
{  /* This subroutine finds the location of the 45 degree line */

  int i,j,q,p;
  int z[2*MAXBLK - 4];

  p = 1;
  for (i=0; i<(BLKSIZE-2); i++) p *= (i+2);
  for (i=0; i<(2*BLKSIZE-4); i++) z[i] = 0;
  for (i=0; i<BLKSIZE; i++)
   for (j=0; j<(BLKSIZE-1); j++)
    { q = i+j;
     if ((q>0)&&(q<=(BLKSIZE-2)))
      { if ((Gh[i][j] == x)&&(Gv[i][j]==x)) z[q-1]++;}
     else if ((q>(BLKSIZE-2))&&(q<=(2*BLKSIZE-4)))
      { if ((Gh[i][j] == x)&&(Gv[i-1][j+1]==x)) z[q-1]++;}
    }
  for (i=0; i<(BLKSIZE-2); i++)
   { z[i] *= (p/(i+2)); z[2*BLKSIZE-5 - i] *= (p/(i+2)); }
  return (getmax(z));
}

int D135(int Gv[MAXBLK-1][MAXBLK], int Gh[MAXBLK][MAXBLK-1], int x)
{  /* This subroutine finds the location of the 135 degree line */

  int i,j,q,p;
  int z[2*MAXBLK - 4];

  p = 1;
  for (i=0; i<(BLKSIZE-2); i++) p *= (i+2);
  for (i=0; i<(2*BLKSIZE-4); i++) z[i] = 0;
  for (i=0; i<BLKSIZE; i++)
   for (j=0; j<(BLKSIZE-1); j++)
    { q = i-j;
```

```
        if ((q>(2-BLKSIZE))&&(q<1))
          { if ((Gh[i][j] == -x)&&(Gv[i][j+1]==x)) z[BLKSIZE-2-q]++;}
        else if ((q>0)&&(q<(BLKSIZE-1)))
          { if ((Gh[i][j]== -x)&&(Gv[i-1][j]==x)) z[BLKSIZE-2-q]++;}
      }
  for (i=0; i<(BLKSIZE-2); i++)
    { z[i] *= (p/(i+2)); z[2*BLKSIZE-5 - i] *= (p/(i+2)); }
  return(getmax(z));
 }

int getmax (int z[2*MAXBLK-4])
{  /* returns the location of the maximum */

  int i,max;

  max = 0;
  for (i=1; i<(2*BLKSIZE-4); i++)
    if (z[i]>z[max]) max = i;
  return(max);
 }

void fnextend(char fname[50], char path[30], char fn[10], char ext[5])
{ /* Adds directory and extension to filename */

  strcpy(fname,path);
  strcat(fname,fn);
  strcat(fname,ext);
}

void errout(char fname[50])
{ /* Prints error message and terminates execution if file not opened */

  printf("\ncannot open file %s\n",fname);
  exit(1);
}
```

```
/*****************************************************************

                    Training Pattern Organizer
                    Written by Mark Hetherington
                    Date: August 31, 1993

        This program sorts the vectors in the training images into files
                    corresponding to their classification

*****************************************************************/


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define  MAXBLK    12            /* maximum block size */
#define  MAXVECT  144            /* maximum vector size */
#define  MAXIMGW  696            /* maximum image width */
#define  MAXCL    127            /* maximum number of classes */
#define  CFILES   "class4.fn"    /* classification filenames */


/* Function prototypes */
int getmean(unsigned char x[MAXVECT]);
int getprdm(unsigned char A[MAXBLK][MAXIMGW],unsigned char Z[MAXIMGW],
            int i, int j);
void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW);
void getblock(unsigned char A[MAXBLK][MAXIMGW],
            unsigned char x[MAXVECT],int g);
void fnextend(char fname[50], char path[30], char fn[10], char ext[5]);
void errout(char fname[50]);

/* Global Variables */
int     BLKSIZE,
        VSIZE;

void main(void)
{ int i,j,k,v,nclass,ntrain,m,n,f,M,N,GL,M1,N1,IDXW,IDXH,NUMCL,prdm,bsz;
  unsigned char A[MAXBLK][MAXIMGW],Z[MAXIMGW],x[MAXVECT];
  char fname[50],clfn[MAXCL][10],trfn[10][10],ftype[10];
```

```
FILE *outfile[MAXCL],*classfile,*fnfile,*image;

/*      Variables:   nclass       number of classes
                     ntrain       number of training patterns
                     M            image width
                     N            image height
                     GL           maximum gray level
                     M1           adjusted width
                     N1           adjusted height
                     IDXW         index image width
                     IDXH         index image height
                     NUMCL        number of classes
                     A[][]        row of blocks
                     Z[]          previous row of pixels
                     x[]          training vector
                     clfn[][]     array of class filenames
                     trfn[][]     array of training image filenames

*/

 printf("\n*** Training Set Generation ***\n\n");
 BLKSIZE = MAXBLK+1;
 while (BLKSIZE>MAXBLK)
   { printf("What is the block size: ");
     scanf("%d",&BLKSIZE);
   }
 VSIZE = BLKSIZE*BLKSIZE;
 prdm = 2;
 while (prdm>1)
   { printf("\n(0) actual mean removed.\n(1) predictive mean removed.\n\n");
     printf("Enter coding type: ");
     scanf("%d",&prdm);
   }
 NUMCL = 12*BLKSIZE-17;
 f = (NUMCL+8)/10 + 1;

 /* read in class filenames */
 if ((fnfile = fopen(CFILES,"r"))==NULL) errout(CFILES);
 fscanf(fnfile,"%d",&nclass);
 for (i=0; i<nclass; i++)
   fscanf(fnfile,"%s",clfn[i]);
```

```
fclose(fnfile);

/* read in training image filenames */
if ((fnfile=fopen("trimage.fn","r"))==NULL) errout("trimage.fn");
fscanf(fnfile,"%d",&ntrain);
for (i=0; i<ntrain; i++)
  fscanf(fnfile,"%s",trfn[i]);

for (n=1; n<f; n++)
  { /* open class files */
    for (i=(n-1)*10; i<(n*10); i++)
      { if (i<NUMCL)
          { fnextend(fname,"Trdata\\",clfn[i],".tr");
            printf("%d) opening %s\n",i,fname);
            if ((outfile[i]=fopen(fname,"wb"))==NULL) errout(fname);
          }
      }
    /* sort image blocks into class files */
    for (k=0; k<ntrain; k++)
      { /* open training image */
        fnextend(fname,"Img\\",trfn[k],".img");
        if ((image=fopen(fname,"rb"))==NULL) errout(fname);
        printf("\nreading %s\n",fname);
        fscanf(image,"%s %d %d %d",ftype,&M,&N,&GL);
        fgetc(image);
        if (M>MAXIMGW)
          { printf("Image width too large, %d > %d\n",M,MAXIMGW);
            exit(1);
          }

        /* open classification file */
        fnextend(fname,"Classimg\\",trfn[k],".xx");
        if ((classfile=fopen(fname,"rb"))==NULL) errout(fname);
        bsz = fgetc(classfile);
        if (bsz != BLKSIZE)
          { printf("ERROR: Inconsistent Block Sizes\n");
            exit(1);
          }
        /* set parameters */
        IDXW = M/BLKSIZE;
        IDXH = N/BLKSIZE;
        M1 = IDXW*BLKSIZE;
        N1 = IDXH*BLKSIZE;
        if ((M!=M1)||(N!=N1))
         printf("Warning: Non-integral image dimensions (%d,%d) ==> (%d,%d)\n",
                M,N,M1,N1);


        printf("sorting file\n");
        for (i=0; i<IDXH; i++)
         { for (j=0; j<M; j++)
            Z[j] = A[BLKSIZE-1][j]; /* save last row of A[][] */
          getrow(image,A,M);       /* get row of blocks */
          for (j=0; j<IDXW; j++)
           { getblock(A,x,j);      /* get block */
             m=fgetc(classfile);   /* read classification */
             if ((m/10)!=(n-1)) continue;

             /* get mean or predict mean and write to file */
             if (prdm) fputc(getprdm(A,Z,i,j),outfile[m]);
             else fputc(getmean(x),outfile[m]);

             /* write vector to file */
             for (v=0; v<VSIZE; v++)
              fputc(x[v],outfile[m]);
           }
         }
        fclose(image);
        fclose(classfile);
       }

      /* close files */
      for (i=(n-1)*10; i<(n*10); i++)
       if (i<NUMCL) fclose(outfile[i]);
     }
   }
```

```
int getmean(unsigned char x[MAXVECT])
{ int sum,i;

  /* This subroutine returns the mean of the given block */

  sum = 0;
  for (i = 0; i<VSIZE; i++) sum += (int)x[i];
  return((sum + VSIZE/2)/VSIZE);
}


int getprdm(unsigned char A[MAXBLK][MAXIMGW],unsigned char Z[MAXIMGW],
                        int i, int j)
{ int sum,k,v,m,d;

  /* This subroutine returns the predicted mean for current block */

  m = j*BLKSIZE;

/* If first block, return actual mean */
  if (!i && !j)
    { sum = 0;
      for (k=0; k<BLKSIZE; k++)
        for (v=0; v<BLKSIZE; v++)
          sum += A[k][m+v];
      return((sum + VSIZE/2)/VSIZE);
    }

/* If first row or first column, use only four pixels */
  if (!i || !j)
    { d = BLKSIZE;
      sum = 0;
    }
  else
    { d = (2*BLKSIZE+1);
      sum = Z[m-1];
    }
  if (j)
    for (k=0; k<BLKSIZE; k++)
      sum += A[k][m-1];
if (i)
    for (k=0; k<BLKSIZE; k++)
      sum += Z[m+k];
  return((sum + d/2)/d);
}

void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW)
{ int i,j;

/* This subroutine gets a row of blocks from image */

  for (i=0; i<BLKSIZE; i++)
    for (j=0; j<IMGW; j++)
      A[i][j] = fgetc(image);
}

void getblock(unsigned char A[MAXBLK][MAXIMGW],
                    unsigned char x[MAXVECT],int g)
{  int i,j;
/* This subroutine gets the gth block from A[][] */

   for (i=0; i<BLKSIZE; i++)
     for (j=0; j<BLKSIZE; j++)
       x[i*BLKSIZE + j] = A[i][g*BLKSIZE+j];
 }

void fnextend(char fname[50], char path[30], char fn[10], char ext[5])
{
/* This subroutine adds the directory and extension */

  strcpy(fname,path);
  strcat(fname,fn);
  strcat(fname,ext);
}

void errout(char fname[50])
{ /* This subroutine prints error if file not open */

  printf("\ncannot open file %s\n",fname);
  exit(1);
}
```

```
/*****************************************************************

                     Codebook Design Program
                   Written by Mark Hetherington
                       Date:  August 31,1993

        This program is a menu driven program that allows individual design
     of a codebook for each class.  Each sub-codebook is written to its own file.


*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <malloc.h>

#define  MAXVECT   64            /* maximum vector size */
#define  CBSIZE   1024           /* maximum codebook size */
#define  MAXCL     80            /* maximum number of classes */
#define  CFILES  "class4.fn"     /* classification filenames */


/* function prototypes */
float LBG(float __far *cbook[CBSIZE], int nclass, long npat, float tol);
void writecbook(char fname[50], float __far *cbook[CBSIZE], int nclass);
int getic (float __far *cbook[CBSIZE],long npat,int *nclass);
void fnextend(char fname[50], char path[30], char fn[10], char ext[5]);
void errout(char fname[50]);

/* global variables */
int VSIZE;     /* vector size */
float DSTH;    /* distance threshold for initial codebook selection */
```

```
void main (void)
{ int i,m,nclass,nfile,cl,xtra;
  long npat;
  char fname[50],fnarry[MAXCL][10];
  float err,tol, __far *cbook[CBSIZE];
  FILE *classfile,*outfile,*trdata;

/*
Variables:      cbook           codebook
                err             average distortion over training set
                tol             convergence tolerence
                npat            number of training patterns
                nclass          number of classes
*/


  printf("\n\n*** LBG TRAINING ***\n\n");
  VSIZE = MAXVECT+1;
  while (VSIZE>MAXVECT)
    { printf("What is the vector size (blocksize^2): ");
      scanf("%d",&VSIZE);
    }

  /* read classification filenames */
  if ((classfile = fopen(CFILES,"r"))==NULL) errout(CFILES);
  fscanf(classfile,"%d",&nfile);
  for (i=0; i<nfile; i++)
    fscanf(classfile,"%s",fnarry[i]);
  fclose(classfile);
  strcpy(fnarry[nfile],"QUIT");

  while(1)
    {
      /* print menu */
      printf("\n\n          *** CODEBOOK TRAINING ***\n\n");
      for (i=0; i<((nfile+1)/3); i++)
        printf("(%2d) %6s     (%2d) %6s     (%2d) %6s\n",
               i,fnarry[i],i+(nfile+1)/3,fnarry[i+(nfile+1)/3],
               i+2*((nfile+1)/3),fnarry[i+2*((nfile+1)/3)]);
      xtra = nfile + 1 - 3*((nfile+1)/3);
for (i=3*((nfile+1)/3); i<(3*((nfile+1)/3)+xtra); i++)
  printf("                    (%2d) %6s\n",i,fnarry[i]);
printf("\nWhich class do you want to train: ");
scanf("%d",&cl);
if ((cl>=nfile)||(cl<0))
  { printf("\nExit...\n");
    exit(0);
  }

/* copy data to ram drive */
printf("\n\n\n*** %s ***\n",fnarry[cl]);
fnextend(fname,"Trdata\\",fnarry[cl],".tr");
if ((trdata = fopen(fname,"rb"))==NULL) errout(fname);
if ((outfile = fopen("D:\\trdata.tr","wb"))==NULL)
    errout("D:\\trdata.tr");
npat=0; m=fgetc(trdata);
while(!feof(trdata))
  { fputc(m,outfile);
    npat++;
    for (i=0; i<VSIZE; i++) fputc(fgetc(trdata),outfile);
    m = fgetc(trdata);
  }
printf("\n%ld patterns in %s\n",npat,fname);
fclose(trdata);
fclose(outfile);

/* choose initial codebook */
while(getic(cbook,npat,&nclass));

/* LBG */
printf("Enter convergence threshold: ");
scanf("%f",&tol);
err = LBG(cbook,nclass,npat,tol);
printf("\n%s converged.\n",fnarry[cl]);

/* write codebook to file */
fnextend(fname,"Codebook\\",fnarry[cl],".cb");
writecbook(fname,cbook,nclass);
}
}
```

```
float LBG(float __far *cbook[CBSIZE], int nclass, long npat, float tol)
{ float diff,x,__far *oldcbook[CBSIZE],trdata[MAXVECT];
 double err,prerr,dist,mindist;
 int i,j,k,m,iter,done,class;
 long z,__far *nvect;
 FILE *trfile;

/* This subroutine implements the LBG algorithm

   The subroutine iterates until the rate of change
   of the average distortion is less than tol.  The
   average distortion is returned to the calling program.


   Variables:  oldcbook     codebook from the previous iteration
               trdata       current training vector
               err          average error for current iteration
               prerr        average error from previous iteration
               nvect        array containing the number of vectors
                            in each class

*/

 iter = 0; done = 0; prerr = HUGE_VAL;
 for (i=0; i<nclass; i++)
   { if (!(oldcbook[i] = (float __far *)_fcalloc(VSIZE,sizeof(float))))
       { printf("LBG: unable to allocate %dth codeword\n",i);
         exit(1);
       }
   }
 if (!(nvect = (long __far *)_fcalloc(1024,sizeof(long))))
   { printf("LBG: unable to allocate counter\n",i);
     exit(1);
   }

 while (!done)
   { iter++;
     if ((trfile = fopen("D:\\trdata.tr","rb"))==NULL) errout("D:\\trdata.tr");
```

```
/* initialize codebooks and counters */
for (i=0; i<nclass; i++)
  { for (j=0; j<VSIZE; j++)
      { *(oldcbook[i] + j) = *(cbook[i] + j);
        *(cbook[i] + j) = (float)0.0;
      }
    *(nvect + i) = 0;
  }

err = 0.0;
for (z=0; z<npat; z++)
  { mindist = HUGE_VAL;

    /* read in vector */
    m = fgetc(trfile);
    for (j=0; j<VSIZE; j++)
      trdata[j] = (float)(fgetc(trfile) - m);

    /* find closest codevector */
    for (j=0; j<nclass; j++)
      { dist = 0.0;
        for (k=0; k<VSIZE; k++)
          { x = trdata[k] - (*(oldcbook[j] + k));
            dist += (double)(x*x);
            if (dist>mindist) k=VSIZE;
          }
        if (dist<mindist)
          { class = j;
            mindist = dist;
          }
      }
    err += (mindist/((double)VSIZE));

    /* add vector to class sum and increment class counter*/
    for (k=0; k<VSIZE; k++)
      *(cbook[class] + k) += trdata[k];
    *(nvect+class) = *(nvect+class) + 1;
  }
fclose(trfile);
err /=((double)npat);
```

```
    /* print class membership */
    printf("\nclass membership\n");
    for (i=0; i<nclass; i++)
      printf("vector # %d = %ld\n",i,*(nvect+i));
    printf("iteration #%d.....error=%f\n",iter,err);

    /* check for convergence */
    done = 1;
    for (i=0; i<nclass; i++)
      for (j=0; j<VSIZE; j++)
        { *(cbook[i] + j) /= ((float)(*(nvect+i)));
         if ((done)&&(*(cbook[i]+j) != *(oldcbook[i]+j))) done = 0;
        }
    diff = (float)((prerr - err)/(err));
    if (iter>1) printf("rate of convergence = %f%%\n",diff*100.0);
    if (diff<tol) done = 1;
    prerr = err;
   }
  return ((float)err);
}

int getic (float __far *cbook[CBSIZE],long npat,int *nclass)
{ int i,m,n,v,unique;
  long k,j;
  double ex,dist;
  FILE *trdata;

  /* This subroutine selects the initial codebook */

  printf("\nInitial Codebook Selection\n\n");
  printf("How many classes? ");
  scanf("%d",nclass);

  for (i=0; i<(*nclass); i++)
   { if (!(cbook[i] = (float __far *)_fcalloc(VSIZE,sizeof(float))))
      { printf("CB: unable to allocate %dth codeword\n",i);
       exit(1);
      }
   }
```

```
 printf("what is the distance threshold: ");
 scanf("%f",&DSTH);

 k=0;
 if ((trdata = fopen("D:\\trdata.tr","rb"))==NULL)
   errout("D:\\trdata.tr");
 for (i=0; i<(*nclass); i++)
   { unique = 0;
     while(!unique)
       { unique = 1;
         if (k>npat)
           { printf("%d unable to find a unique set of vectors\n",i);
             return(1);
           }
         k++;

         /* read training vector */
         m = fgetc(trdata);
         for (n=0; n<VSIZE; n++)
           *(cbook[i]+n) = (float)(fgetc(trdata)-m);
         if (feof(trdata)) { printf("EOF\n"); exit(1); }

         /* compare distance to current codevectors */
         for (n=0; n<i; n++)
           { dist = 0.0;
             for (v=0; v<VSIZE; v++)
               { ex = *(cbook[n]+v) - *(cbook[i]+v);
                 dist += ex*ex;
               }
             if (dist<DSTH)
               { unique = 0; n = i;}
           }
       }

   }
 fclose(trdata);
 printf("\nScanned %ld of %ld patterns.\n\n",k,npat);
 return(0);
}
```

```
void writecbook(char fname[50], float __far *cbook[CBSIZE], int nclass)
{ FILE *outfile;
  int i,j;

  /* This subroutine writes the codebook to a file */

  if ((outfile = fopen(fname,"w"))==NULL) errout(fname);
  fprintf(outfile,"%d\n",nclass);
  for (i=0; i<nclass; i++)
    for (j=0; j<VSIZE; j++)
      fprintf(outfile,"%f\n",*(cbook[i]+j));
  fclose(outfile);
}

void fnextend(char fname[50], char path[30], char fn[10], char ext[5])
{
/* This subroutine adds the directory and extension */

  strcpy(fname,path);
  strcat(fname,fn);
  strcat(fname,ext);
}

void errout(char fname[50])
{ /* This subroutine prints error if file not open */

  printf("\ncannot open file %s\n",fname);
  exit(1);
}
```

```
/*********************************************************************

                    CODEBOOK MERGING PROGRAM

                  Written By: Mark Hetherington
                     Date: August 31, 1993

       This program merges the subcodebooks created by the LBG program.

*********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define  CBSIZE     300          /* Maximum codebook size */
#define  VSIZE      16           /* Vector size */
#define  CFILES   "class4.fn"    /* class filename file */

void readln (FILE *infile, unsigned char s[31]);
void errout(char fname[50]);
void fnextend(char fname[50], char path[30], char fn[10], char ext[5]);

void main (void)
{ int i,j,k,nfile,ncl,nclass;
  float cbook[CBSIZE][VSIZE],z;
  char fn[10], fname[50];
  FILE *infile,*outfile,*classfile;

/* Variables:   cbook          concatenated codebook
*/

  if ((outfile = fopen("Codebook\\cbook.cb","w"))==NULL)
    errout("Codebook\\cbook.cb");
  if ((classfile = fopen(CFILES,"r"))==NULL) errout(CFILES);
  fscanf(classfile,"%d",&nfile);

  nclass = 0;
  for (i=0; i<nfile; i++)
    { /* open current codebook file and get number of codewords */
      fscanf(classfile,"%s",fn);
      fnextend(fname,"Codebook\\",fn,".cb");
      if ((infile = fopen(fname,"r"))==NULL) errout(fname);
      fscanf(infile,"%d",&ncl);

      /* read current codebook */
      for (j=0; j<ncl; j++)
        for (k=0; k<VSIZE; k++)
          { fscanf(infile,"%f",&z);
            cbook[nclass+j][k] = z;
          }
      nclass += ncl;
      printf("class %6s = %2d\n",fn,ncl);

      /* write codebook division information */
      fprintf(outfile,"%d\n",nclass);
      fclose(infile);
    }
  printf("total codewords = %d\n",nclass);

  /* write merged codebook */
  for (i=0; i<nclass; i++)
    for (j = 0; j<VSIZE; j++)
      fprintf(outfile,"%f\n",cbook[i][j]);
}

void fnextend(char fname[50], char path[30], char fn[10], char ext[5])
{
/* This subroutine adds the directory and extension */

  strcpy(fname,path);
  strcat(fname,fn);
  strcat(fname,ext);
}

void errout(char fname[50])
{ /* This subroutine prints error if file not open */

  printf("\ncannot open file %s\n",fname);
  exit(1);
}
```

```
/*****************************************************************

                         VQ Coding Program

                   Written by: Mark Hetherington
                      Date: August 31, 1993

         This program codes an input image using a CVQ codebook.

         Input Files:   Original Image File, Classification File,
                        Codebook File
         Output Files:  VQ Image, Index File


     Variables:     A      Contains a row of blocks from image
                    Z      Contains the last row of the previous
                           row of blocks
                    x      Block from original image at start of loop,
                           VQ block at the end of loop
                    y      Copy of original block
                    z      Class number for current block
                    cbook  CVQ codebook
                    cbdiv  Class divisions for codebook
                    mean   Mean of current block
                    index  Codebook index for current block

*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define MAXBLK    8     /* Maximum Block Size */
#define MAXVECT   64    /* Maximum vector size = MAXBLK^2 */
#define CBSIZE    256   /* Maximum Codebook Size */
#define MAXCL     80    /* Maximum number of different classes plus one
                               = 12*MAXBLK - 16 */
#define MAXIMGW   696   /* Maximum Image width */
```

```
/* Function prototypes */
int getprdm(unsigned char A[MAXBLK][MAXIMGW],unsigned char Z[MAXIMGW],
                    int i, int j);
int getmean(int x[MAXVECT]);
void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW);
void putrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW);
void getblock(unsigned char A[MAXBLK][MAXIMGW], int x[MAXVECT],int g);
void putblock(unsigned char A[MAXBLK][MAXIMGW], int x[MAXVECT], int g);
void readcbook(float *cbook[CBSIZE], int cbdiv[MAXCL]);
void readln (FILE *infile, unsigned char s[31]);
void errout(char fname[50]);
void fnextend(char fname[50], char path[30], char fn[10], char ext[5]);

/* global variables */
int BLKSIZE, VSIZE;

void main(int argc, char *argv[])
{ char fn[10],fname[50],ftype[5];
 float *cbook[CBSIZE],SNR,ex;
 unsigned char A[MAXBLK][MAXIMGW], Z[MAXIMGW];
 int i,j,k,z,mean,n,index,v,x[MAXVECT],y[MAXVECT],M,N,GL,cbdiv[MAXCL];
 int IDXW,IDXH,M1,N1;
 unsigned prdm;
 double err,mindist,dist;
 FILE *classfile,*codefile,*image,*cdimg;

 if (argc<2)
  { printf("\nWhat is the image filename? ");
   scanf("%s",fn);
  }
 else
  strcpy (fn,argv[1]);

 prdm = 2;
 while (prdm>1)
  { printf("\n(0) actual mean removed.\n(1) predictive mean removed.\n\n");
   printf("Enter coding type: ");
   scanf("%d",&prdm);
  }
 /* open files */
 fnextend(fname,"Classimg/",fn,".xx");
 if ((classfile = fopen(fname,"rb"))==NULL) errout(fname);
 fnextend(fname,"Coded/",fn,".idx");
 if ((codefile = fopen(fname,"wb"))==NULL) errout(fname);
 fnextend(fname,"Img/",fn,".img");
 if ((image = fopen(fname,"rb"))==NULL) errout(fname);
 fnextend(fname,"Img/",fn,".VQ");
 fscanf(image,"%s %d %d %d",ftype,&M,&N,&GL); fgetc(image);
 if (M > MAXIMGW)
   { printf("Image width (%d) > %d\n",M,MAXIMGW);
     exit(1);
   }

/* Initialize parameters */
 BLKSIZE = fgetc(classfile);
 printf("Block Size: %d\n",BLKSIZE);
 VSIZE = BLKSIZE*BLKSIZE;
 IDXW = M/BLKSIZE;
 IDXH = N/BLKSIZE;
 N1 = IDXH*BLKSIZE;
 M1 = IDXW*BLKSIZE;
 if ((M!=M1)||(N!=N1))
   printf("Warning: Reduced image dimensions (%d,%d) ==> (%d,%d)\n",
                              N,M,N1,M1);

 /* read codebook */
 printf("\nreading codebook\n");
 readcbook(cbook,cbdiv);

 if ((cdimg = fopen(fname,"wb"))==NULL) errout(fname);
 fprintf(cdimg,"%s\n%d %d\n%d\n",ftype,M1,N1,GL);

 printf("\nCoding %s\n",fn);
 err = 0.0;
 for (i=0; i<IDXH; i++)
   { /* copy last row to Z */
     for (j=0; j<M; j++) Z[j] = A[BLKSIZE-1][j];

     /* read new row from image */
     getrow(image,A,M);
```

```
for (j=0; j<IDXW; j++)
 { /* read classification of block */
   z = fgetc(classfile);

   /* get new block, get mean, remove mean from vector */
   getblock(A,x,j);
   if (prdm) mean = getprdm(A,Z,i,j);
   else mean = getmean(x);
   for (n=0; n<VSIZE; n++)
     { y[n] = x[n];
       x[n] -= mean;
     }

   /* find closest codevector */
   mindist = HUGE_VAL;
   for (n=cbdiv[z]; n<cbdiv[z+1]; n++)
     { dist = 0.0;
       for (k=0; k<VSIZE; k++)
         { ex = ((float)x[k])-(*(cbook[n] + k));
           dist += (ex*ex);
           if (dist>mindist) k=VSIZE;
         }
       if (dist<mindist)
         { index = n;
           mindist = dist;
         }
     }

   /* replace original vector with coded vector */
   for (n=0; n<VSIZE; n++)
     { v  = (int)(*(cbook[index] + n) + mean + 0.5);
       if (v>GL) v = GL;
       if (v<0) v = 0;
       x[n] = v;
     }

   /* put reconstructed vector into image, output codeword index */
   putblock(A,x,j);
   fputc(index,codefile);
```

```
          /* calculate quantization error */
          dist = 0.0;
          for (n=0; n<VSIZE; n++)
            { ex = x[n] - y[n];
              dist += (ex*ex);
            }
          err += (dist/((float)VSIZE));
         }

      /* write quantized blocks to VQ image */
      putrow(cdimg,A,M1);
    }

  /* Calculate SNR */
  err /= ((float)IDXW*IDXH);
  printf("%d %f\n",GL,err);
  SNR = (float)(10.0*log10(((float)GL*GL)/err));

  printf("\nImage: %s coded with SNR= %f dB\n",fn,SNR);
}

int getmean(int x[MAXVECT])
{ int sum,i;

/* This subroutine returns the mean of the given vector */

  sum = 0;
  for (i = 0; i<VSIZE; i++) sum += x[i];
  return((sum + VSIZE/2)/VSIZE);
}

int getprdm(unsigned char A[MAXBLK][MAXIMGW],unsigned char Z[MAXIMGW],
int i, int j)
{ /* This subroutine calculates the predicted mean for block (i,j) */

  float d;
  int sum,k,v,m;
```

```
 /* If first block in image, return actual mean of the block */
 m = j*BLKSIZE;
 if (!i && !j)
   { sum = 0;
     for (k=0; k<BLKSIZE; k++)
       for (v=0; v<BLKSIZE; v++)
         sum += A[k][m+v];
     return((sum+VSIZE/2)/VSIZE);
   }

 /* For other blocks, predict mean */
 if (!i || !j)
   { d = (float)BLKSIZE;
     sum = 0;
   }
 else
   { d = ((float)(2*BLKSIZE + 1));
     sum = Z[m-1];
   }

 /* if not left edge of image */
 if (j)
   for (k=0; k<BLKSIZE; k++)
     sum += A[k][m-1];

 /* if not top edge of image */
 if (i)
   for (k=0; k<BLKSIZE; k++)
     sum += Z[m+k];
 return((int)(sum/d + 0.5));
}

void getrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW)
{ /* This subroutine reads BLKSIZE rows from image */

 int i,j;

 for (i=0; i<BLKSIZE; i++)
   for (j=0; j<IMGW; j++)
     A[i][j] = fgetc(image);
}
```

```
void putrow(FILE *image, unsigned char A[MAXBLK][MAXIMGW], int IMGW)
{ /* This subroutine writes BLKSIZE rows to image */

 int i,j;

 for (i=0; i<BLKSIZE; i++)
   for (j=0; j<IMGW; j++)
     fputc(A[i][j],image);
}

void getblock(unsigned char A[MAXBLK][MAXIMGW], int x[MAXVECT],int g)
{ /* This subroutine copies a block into x */

 int i,j;

 for (i=0; i<BLKSIZE; i++)
   for (j=0; j<BLKSIZE; j++)
     x[i*BLKSIZE+j] = A[i][g*BLKSIZE+j];
}

void putblock(unsigned char A[MAXBLK][MAXIMGW], int x[MAXVECT], int g)
{ /* This subroutine places block x into A */

 int i,j;

 for (i=0; i<BLKSIZE; i++)
   for (j=0; j<BLKSIZE; j++)
     A[i][g*BLKSIZE+j] = x[i*BLKSIZE+j];
}

void readcbook(float *cbook[CBSIZE], int cbdiv[MAXCL])
{ /* This subroutine reads the codebook from a file */

 FILE *infile;
 int i,j,NUMCL;

 NUMCL = 12*BLKSIZE-16;
 if ((infile = fopen("Codebook/cbook.cb","r"))==NULL) errout("cbook.cb");
 cbdiv[0] = 0;
```

```
  for (i=1; i<NUMCL; i++)
    fscanf (infile,"%d",&cbdiv[i]);
  printf("%d %d\n",NUMCL,cbdiv[NUMCL-1]);
  for (i=0; i<cbdiv[NUMCL-1]; i++)
    { if ((cbook[i] = malloc(VSIZE*sizeof(float))) == NULL)
        { printf("Error: memory allocation\n");
          exit(1);
        }

      for (j=0; j<VSIZE; j++)
        fscanf(infile,"%f",(cbook[i] + j));
    }
  fclose(infile);
}

void fnextend(char fname[50], char path[30], char fn[10], char ext[5])
{
/* This subroutine adds the directory and extension */

  strcpy(fname,path);
  strcat(fname,fn);
  strcat(fname,ext);
}

void errout(char fname[50])
{ /* This subroutine prints error if file not open */

  printf("\ncannot open file %s\n",fname);
  exit(1);
}
```

```
/***********************************************************************

                        Arithmetic Coder Subroutine
          Adapted from a subroutine originally written by Dr. T.V. Ramabadran

        This subroutine is used by all of the lossless programs in this thesis.  The
subroutine takes as input the cumulative frequency table to be used, the index of the
current symbol to be coded, the total number of symbols in the table and a initialization
parameter.  The program performs arithmetic coding using the run-length buffer
representation and returns the number of output bytes so far.  The output bytes are not
written to a file.  They are only counted.

***********************************************************************/

#define  WIDTH     (1 << 14)     /* Width to rescale interval width */
#define  BIT14MSK  0x4000        /* Bit mask for most significant bit of
                                    x register */
#define  BIT15MSK  0x8000         /* Bit mask for carry bit */
#define  BYTESIZE  8             /* Number of bits/byte */
#define  EOF       (-1)          /* end-of-message symbol */

int Jcode(int f[], int symbol, int nsymbol,int *start)
{static unsigned short  x;     /* start of interval */
 static unsigned short  w;     /* width of interval */
 static short  code_byte;      /* unfilled byte     */
 static short  bit_count;      /* number of available bits in code_byte */
 static short  next_bit;       /* next bit to be stored */
 static long   run_length;     /* run counter */
 short       z;                /* interval offset */
 short       code_bit;         /* bit shifted from x register */
 short       carry;            /* carry flag */
 short       run_bit;          /* run bit */
 int        nbyte = 0;         /* output byte counter */

  /* Initialize the static variables on first call */
  if (*start)
   {
    x = 0;
    w = WIDTH;
    bit_count = BYTESIZE + 2;
    run_length = 0;
   *start = 0;
  }

/* if not end-of message */
if (symbol != EOF)
  {
  /* if symbol = 0, z = 0 otherwise calculate z */
  z = (symbol) ? (2L * (long) w * (long) f[symbol] + (long) f[nsymbol])
               / (2L * (long) f[nsymbol]) : 0;

  /* New code point = Old code point + offset */
  x += z;

  /* Find new interval width */
  w = (2L * (long) w * (long) f[symbol + 1] + (long) f[nsymbol])
     / (2L * (long) f[nsymbol]) - z;

  /* rescale interval width until it is > width */
  while (w < WIDTH)
    {
    /* Check for carry over */
    carry    = (x & BIT15MSK) ? 1 : 0;

    /* Save most significant bit of x register */
    code_bit = (x & BIT14MSK) ? 1 : 0;

    /* rescale start of interval */
    x <<= 1;

    /* remove code_bit from x register */
    x &= (~BIT15MSK);

    /* rescale interval width */
    w <<= 1;

    /* If there is no carry and the bit shifted out of
       x register is 1, increment run_length.  Go to the
       end of the loop. */
    if ((!carry) && (code_bit))
      {
      run_length++;
```

```
        continue;
       }

     /* If carry occurs, set run bit as 0 */
     if (carry) { next_bit++; run_bit = 0; }

     /* Otherwise, the run bit is 1 */
     else      run_bit = 1;

     /* shift next bit into data stream and decrement bit_count */
     code_byte <<= 1;
     code_byte += next_bit;
     bit_count--;

     /* if code_byte is full, write to file */
     if (!bit_count)
       {
       /* putchar (code_byte); */
       bit_count = BYTESIZE;
       nbyte++;
       }

     /* Set next_bit as bit shifted from x register */
     next_bit = code_bit;

     /* write out run of digits */
     while (run_length)
       {
       run_length--;
       code_byte <<= 1;
       code_byte += run_bit;
       bit_count--;
       if (!bit_count)
         {
         /* putchar (code_byte); */
         bit_count = BYTESIZE;
         nbyte++;
         }  /* if */
       }  /* while */
   }  /* while */
 }  /* if */
```

```
/* If end-of-message symbol occurs, then write out the remaining
  part of the codeword,i.e., next_bit, run, and code_bit,
  to the output file. */
else
 {
  /* check for carry */
  carry    = (x & BIT15MSK) ? 1 : 0;

  /* save most significant bit of x register */
  code_bit = (x & BIT14MSK) ? 1 : 0;

  /* If carry occurs set run bit as 0 */
  if (carry) { next_bit++; run_bit = 0; }

  /* Otherwise set run bit as 1 */
  else      run_bit = 1;

  /* Shift next bit into code_byte and decrement bit_count */
  code_byte <<= 1;
  code_byte += next_bit;
  bit_count--;

  /* if code_byte is full, write to file */
  if (!bit_count)
   {
    /* putchar (code_byte); */
    bit_count = BYTESIZE;
    nbyte++;
   }

  /* Write out run of digits */
  while (run_length)
   {
    run_length--;
    code_byte <<= 1;
    code_byte += run_bit;
    bit_count--;
    if (!bit_count)
     {
      /* putchar (code_byte); */
      bit_count = BYTESIZE;
```

```
         nbyte++;
        }
      }

    /* write out bit shifted from x register */
    code_byte <<= 1;
    code_byte += code_bit;
    bit_count--;

    /* fill rest of last byte with ones */
    while (bit_count)
      {
       code_byte <<= 1;
       code_byte += 1;
       bit_count--;
      }
    /* putchar (code_byte); */
    nbyte++;

  }  /* else */

  /* Return number of bytes written out so far */
  return(nbyte);
}
```

```
/*************************************************************************

                    Memoryless Model Coder
        Adapted from a program originally written by Dr. T.V. Ramabadran

          This program codes the VQ indices using the memoryless model.

*************************************************************************/

#include  <stdio.h>
#include <stdlib.h>

int Jcode(int f[], int symbol, int nsymbol,int *start);

#define   MAXCBSIZE       512         /* Maximum Codebook size */
#define   MAX_COUNT       (1 << 14)   /* Maximum frequency count */
#define   EOF             (-1)        /* end-of-message symbol */

void main (int argc,char *argv[])
{ FILE  *infile;
  int cf_table [MAXCBSIZE],CBSIZE;
  int cur_index,  count, i, j, start, Ki;
  long  insize, outsize;

  /* Read in alphabet size */
  CBSIZE = MAXCBSIZE+1;
  while (CBSIZE>MAXCBSIZE)
    { printf("Alphabet size: ");
      scanf("%d",&CBSIZE);
    }

  /* Read in increment */
  printf("What is the increment: ");
  scanf("%d",&Ki);

  /* open input file */
  if (!(infile = fopen (argv [1],"rb")))
    { fprintf (stderr,"\ncannot open %s\n",argv [1]) ;
      exit (0) ;
    }
```

```
/* Initialize frequency table to uniform distribution */
for (count = 0 ; count < (CBSIZE+1) ; count++)
  cf_table [count] = count;

/* Initialize parameters */
start = 1;
insize = 0;
outsize = 0;

/* Read in first byte */
cur_index = fgetc(infile);

/* while data remain, code */
while (!feof(infile))
  { insize++;

    /* call arithmetic coder with current symbol */
    outsize += Jcode(cf_table,cur_index,CBSIZE,&start);

    /* Update frequency table */
    for (count = cur_index + 1; count < (CBSIZE+1); count++)
      cf_table [count] += Ki;

    /* If total count is too large, rescale */
    if (cf_table[CBSIZE] >= MAX_COUNT)
     {
       for (count = 1; count < (CBSIZE+1); count++)
         { cf_table[count] /= 2;
          if (cf_table[count] - cf_table[count-1] <= 0)
            cf_table[count] = cf_table[count-1] + 1;
         }
     }

    /* Read in next symbol */
    cur_index = fgetc(infile);
    if (cur_index>= CBSIZE)
      { printf("Error: Value out of range (%d)\n",cur_index);
        exit(1);
      }
  }
```

```
 /* Call arithmetic coder with end of message */
 outsize += Jcode(cf_table,EOF,CBSIZE,&start);

 fclose (infile);

 printf ("\nInput symbols   :%ld",insize);
 printf ("\nOutput bytes  :%ld",outsize);
}
```

```
/************************************************************************

                        First Order Model Coder
         Adapted from a program originally written by Dr. T.V. Ramabadran

 This program codes VQ indices using a first order model with a north neighbor context

************************************************************************/
#include  <stdio.h>
#include  <stdlib.h>

#define   NROWS          128          /* Number of rows */
#define   NCOLS          128          /* Number of columns */
#define   MaxCBSIZE      256          /* Maximum codebook size */
#define   MAX_COUNT      (1 << 14)    /* Maximum frequency count */
#define   EOF            (-1)         /* end-of-message symbol */

int Jcode(int f[], int symbol, int nsymbol,int *start);

main (int argc, char *argv[])
{ FILE  *infile;
  int prow[NCOLS];                    /* previous row of indices */
  int crow[NCOLS];                    /* current row of indices */
  int *cf_table;                      /* current cumulative frequency table */
  int table[MaxCBSIZE][MaxCBSIZE];    /* collection of first order tables */
  int context;                        /* current context */
  int i, j, count, start;
  long  insize, outsize;

  /* Open input file */
  if (!(infile = fopen (argv [1],"rb")))
   { fprintf (stderr,"\ncannot open %s\n",argv [1]) ;
     exit (0) ;
   }

  /* Read in alphabet size */
  CBSIZE = MAXCBSIZE+1;
  while (CBSIZE>MAXCBSIZE)
    { printf("Alphabet size: ");
      scanf("%d",&CBSIZE);
    }
 TABLE_SIZE = CBSIZE+1;

 /* Read in increment */
 printf("What is the increment: ");
 scanf("%d",&Ki);

/* Initialize tables */
 for (i = 0; i < CBSIZE; i++)
   for (count = 0; count < TABLE_SIZE; count++)
     table[i][count] = count;

/* Initialize parameters */
 start = 1;
 insize = 0;
 outsize = 0;

 for (i = 0; i < NROWS; i++)
   {
     /* copy current row and read in new row of indices */
     for (j=0; j<NCOLS; j++)
       { prow[j] = crow[j];
         crow[j] = fgetc(infile);
       }

     for (j = 0; j < NCOLS; j++)
       { insize++;

         /* Context selection */
         /* if first block, context=0 */
         if ((i == 0) && (j == 0)) context = 0;

         /* if left column, use west neighbor */
         else if (i == 0) context = crow[j-1];

         /* Otherwise use north neighbor */
         else context = prow[j];

         /* use appropriate table */
         cf_table = table[context];

         /* Code symbol */
```

```
          outsize += Jcode(cf_table,crow[j],CBSIZE,&start);

          /* Update table */
          for (count = crow[j] + 1; count < TABLE_SIZE; count++)
            cf_table [count] += Ki;

          /* If total count is too large, rescale */
          if (cf_table[CBSIZE] >= MAX_COUNT)
            { for (count = 1; count < TABLE_SIZE; count++)
                { cf_table[count] /= 2;
                  if (cf_table[count] - cf_table[count-1] <= 0)
                    cf_table[count] = cf_table[count-1] + 1;
                }
            }
      } /*for j*/
    } /*for i*/

  /* Call coder with end-of-message */
  outsize += Jcode(cf_table,EOF,CBSIZE,&start);

  fclose (infile);

  printf ("\nInput symbols: %ld",insize);
  printf ("\nOutput bytes : %ld",outsize);

} /*main*/
```

```
/************************************************************************
                Classification Index Compression Program
       Adapted from a program originally written by Dr. T.V. Ramabadran

    The program compresses the classification index for the two step method.

************************************************************************/
#include  <stdio.h>
#include  <stdlib.h>

#define   NROWS           128               /*Number of rows*/
#define   NCOLS           128               /*Number of columns*/
#define   NCLASS          10                /* Number of Lossless classes */
#define   IDX_SIZE        4                 /* Number of bits for class index */
#define   TABLE_SIZE      (NCLASS + 1)      /* Table size */
#define   MAX_COUNT       (1 << 14)         /* Maximum Frequency Count */
#define   TERMINAL        1                 /* terminal node */
#define   INTERNAL        0                 /* internal node */
#define   STRG_SIZE       (4*IDX_SIZE)      /* Size of sting */
#define   MAX_DEPTH       STRG_SIZE         /* Maximum tree depth */
#define   MSB_MASK        0x08              /* Most significant bit mask */
#define   EOF             (-1)              /* End-of-message symbol */
#define   NULL            0

main (int argc,char *argv[])
{ FILE  *infile;
  int prow[NCOLS];               /* previous row of indices */
  int crow[NCOLS];               /* current row of indices */
  int Ki;                        /* Increment */
  int Kt;                        /* Context threshold */
  int Nc;                        /* Maximum number of contexts */
  int cntxt_strg[STRG_SIZE];     /* Context string */
  int *cf_table, new_index;
  int cur_index, num_index, tmp_index;
  int start, ncntx;
  int i, j, count, depth;
  int insize, outsize;
```

```
/* Define the node structure */
  struct  n_struct
   { int     ntype;              /* Node Type: INTERNAL or TERMINAL */
     int     ctx_count;          /* Number of times node is visited */
     struct  n_struct *c0_ptr;   /* 0 child pointer */
     struct  n_struct *c1_ptr;   /* 1 child pointer */
     int     *tbl_ptr;           /* Pointer to context table */
     int     max_prob;           /* Maximum probability in table */
   };
  typedef  struct  n_struct  NODE, *NodePtr;
  NodePtr   root_ptr, cur_ptr, tmp_ptr;

/* Open input file (classification file)*/
  if (!(infile = fopen (argv [1],"rb")))
   { fprintf (stderr,"\ncannot open %s\n",argv [1]) ;
     exit (0) ;
   }

  fgetc(infile);  /* Remove blocksize byte */

/* Enter parameters */
  printf("What is the increment (Ki): ");
  scanf("%d",&Ki);

  printf("What is the count threshold (Kt): ");
  scanf("%d",&Kt);

  printf("How many contexts (Nc): ");
  scanf("%d",&Nc);

/* Tree initialization. Root Node */
  root_ptr = (NodePtr) malloc (sizeof(NODE));
  root_ptr -> ntype = TERMINAL;
  root_ptr -> ctx_count = 0;
  root_ptr -> c0_ptr = NULL;
  root_ptr -> c1_ptr = NULL;
  root_ptr -> tbl_ptr = (int *) malloc (sizeof(int)*TABLE_SIZE);
  root_ptr -> max_prob = 0;
```

```
 /* Initialize table for root node */
 cf_table = root_ptr -> tbl_ptr;
 for (count = 0; count < TABLE_SIZE; count++)
   cf_table[count] = count;

/* Initialize parameters */
 start = 1;
 num_index = NCLASS;
 ncntx = 1;
 insize = 0;
 outsize = 0;

 for (i=0; i<NROWS; i++)
   { /* Copy current row and read in new row of indices */
    for (j=0; j<NCOLS; j++)
      { prow[j]=crow[j];
       crow[j]=fgetc(infile);
      }

    for (j=0; j<NCOLS; j++)
      {cur_index = crow[j];
       insize++;
       if (cur_index > 1) cur_index--;

      /* Find new class index */
       if ((cur_index == 0) || (cur_index == 1))
         new_index = cur_index;        /* Shade or Midrange */
       else if ((cur_index == 2) || (cur_index == 4) || (cur_index == 6))
         new_index = 2;                /* Positive Horizontal */
       else if ((cur_index == 3) || (cur_index == 5) || (cur_index == 7))
         new_index = 3;                /* Negative Horizontal */
       else if ((cur_index == 8) || (cur_index == 10) || (cur_index == 12))
         new_index = 4;                /* Positive Vertical */
       else if ((cur_index == 9) || (cur_index == 11) || (cur_index == 13))
         new_index = 5;                /* Negative Vertical */
       else if ((cur_index == 14) || (cur_index == 16) ||
                (cur_index == 18) || (cur_index == 20))
         new_index = 6;                /* Positive 45 degrees */
       else if ((cur_index == 15) || (cur_index == 17) ||
            (cur_index == 19) || (cur_index == 21))
         new_index = 7;                /* Negative 45 degrees */
```

```
        else if ((cur_index == 22) || (cur_index == 24) ||
              (cur_index == 26) || (cur_index == 28))
          new_index = 8;                /* Positive 135 degrees */
        else
          new_index = 9;                /* Negative 135 degrees */

      crow[j] = new_index;
      cur_index = new_index;

  /* Fill Context String */
    if ((i == 0) && (j == 0))
      { /* If upper left corner block, fill string with zeros */
        for (count = 0; count < STRG_SIZE; count++)
          cntxt_strg[count] = 0;
      }
    else if (i == 0)
      { /* if top row */
        /* clear string */
        for (count = 0; count < STRG_SIZE; count++)
          cntxt_strg[count] = 0;

        /* Place West neighbor into string */
        tmp_index = crow[j-1];
        for (count = 0; count < IDX_SIZE; count++)
          { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
            tmp_index <<= 1;
          }

        /* if not second column, add second West neighbor to string */
        if (j > 1)
          { tmp_index = crow[j-2];
            for (count = IDX_SIZE; count < 2*IDX_SIZE; count++)
              { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
                tmp_index <<= 1;
              }
          }
      }
    else if (j == 0)
      { /* If left column */
        /* clear string */
      for (count = 0; count < STRG_SIZE; count++)
        cntxt_strg[count] = 0;

      /* Place north neighbor into string */
      tmp_index = prow[j];
      for (count = 0; count < IDX_SIZE; count++)
        { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1: 0;
          tmp_index <<= 1;
        }

      /* If not second row, add second North neighbor */
      if (i > 1)
        { tmp_index = vqimage[i-2][j];
          for (count = IDX_SIZE; count < 2*IDX_SIZE; count++)
            { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
              tmp_index <<= 1;
            }
        }
    }
  else if (j < 127)
    { /* If not right row */
      /* clear string */
      for (count = 0; count < STRG_SIZE; count++)
        cntxt_strg[count] = 0;

      /* Place North neighbor into string */
      tmp_index = prow[j];
      for (count = 0; count < IDX_SIZE; count++)
        { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
          tmp_index <<= 1;
        }

      /* Place West neighbor into string */
      tmp_index = crow[j-1];
      for (count = IDX_SIZE; count < 2*IDX_SIZE; count++)
        { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
          tmp_index <<= 1;
        }
    /* Place Northeast neighbor into string */
    tmp_index = prow[j+1];
    for (count = 2*IDX_SIZE; count < 3*IDX_SIZE; count++)
      { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
        tmp_index <<= 1;
      }

    /* Place Northwest neighbor into string */
    tmp_index = prow[j-1];
    for (count = 3*IDX_SIZE; count < 4*IDX_SIZE; count++)
      { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
        tmp_index <<= 1;
      }
  }
else
  { /* If right column */
    /* clear string */
    for (count = 0; count < STRG_SIZE; count++)
      cntxt_strg[count] = 0;

    /* Place North neighbor into string */
    tmp_index = prow[j];
    for (count = 0; count < IDX_SIZE; count++)
      { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
        tmp_index <<= 1;
      }

    /* Place West neighbor into string */
    tmp_index = crow[j-1];
    for (count = IDX_SIZE; count < 2*IDX_SIZE; count++)
      { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
        tmp_index <<= 1;
      }

    /* Place Northwest neighbor into string */
    tmp_index = prow[j-1];
    for (count = 2*IDX_SIZE; count < 3*IDX_SIZE; count++)
      { cntxt_strg[count] = (tmp_index & MSB_MASK)? 1 : 0;
        tmp_index <<= 1;
      }
}
```

```
/* Find the context and code the index.  Update the statistics */

    /* Start at top of tree */
    cur_ptr = root_ptr; depth = 0;

    /* Decend tree to a terminal node */
    while ((cur_ptr -> ntype == INTERNAL) && (depth < MAX_DEPTH))
      cur_ptr = (cntxt_strg[depth++])? cur_ptr -> c1_ptr: cur_ptr -> c0_ptr;

    /* Use table from selected context */
    cf_table = cur_ptr -> tbl_ptr;

    /* Code classification index */
    outsize += Jcode(cf_table,cur_index,num_index,&start);

    /* Update statistics */
    for (count = cur_index + 1; count < TABLE_SIZE; count++)
      cf_table [count] += Ki;

    /* Update max_prob */
    if (cf_table[cur_index+1]-cf_table[cur_index] > cur_ptr -> max_prob)
      cur_ptr -> max_prob = cf_table[cur_index+1]-cf_table[cur_index];

    /* If total count is too large, rescale */
    if (cf_table[TABLE_SIZE - 1] >= MAX_COUNT)
      { for (count = 1; count < TABLE_SIZE; count++)
          { cf_table[count] /= 2;
           if (cf_table[count] - cf_table[count-1] <= 0)
             cf_table[count] = cf_table[count-1] + 1;
          }
      }

/*If appropriate conditions are satisfied, generate new contexts*/

    /* Increment count in current context */
    (cur_ptr -> ctx_count)++;
    /* If count>Kt and Number of contexts < Nc and
       not full depth, create new context */
    if ((cur_ptr -> ctx_count >= Kt) && (depth < MAX_DEPTH) &&
        (ncntx < Nc))
     { /* convert to internal node and create children */
      cur_ptr -> ntype = INTERNAL;
      cur_ptr -> c0_ptr = (NodePtr) malloc (sizeof(NODE));
      cur_ptr -> c1_ptr = (NodePtr) malloc (sizeof(NODE));

      /* Initialize 0 child */
      tmp_ptr = cur_ptr;
      cur_ptr = tmp_ptr -> c0_ptr;
      cur_ptr -> ntype = TERMINAL;
      cur_ptr -> ctx_count = 0;
      cur_ptr -> c0_ptr = NULL;
      cur_ptr -> c1_ptr = NULL;
      cur_ptr -> tbl_ptr = tmp_ptr -> tbl_ptr;
      cf_table = cur_ptr -> tbl_ptr;
      for (count = 0; count < TABLE_SIZE; count++)
        cf_table[count] = count;

      /* Initialize 1 child */
      cur_ptr = tmp_ptr -> c1_ptr;
      cur_ptr -> ntype = TERMINAL;
      cur_ptr -> ctx_count = 0;
      cur_ptr -> c0_ptr = NULL;
      cur_ptr -> c1_ptr = NULL;
      cur_ptr -> tbl_ptr = (int *) malloc (sizeof(int)*TABLE_SIZE);
      cf_table = cur_ptr -> tbl_ptr;
      for (count = 0; count < TABLE_SIZE; count++)
        cf_table[count] = count;

      ncntx++;

     } /*if*/
    } /*for j*/
   } /*for i*/

 /* Code end-of-message */
 outsize += Jcode(cf_table,EOF,num_index,&start);
```

```
/*********************************************************************
                 Sub-codebook Index Compression Program
        Adapted from a program originally written by Dr. T.V. Ramabadran

      The program compresses the sub-codebook index for the two step method

*********************************************************************/
#include <stdio.h>
#include <stdlib.h>

#define  NROWS          128    /*Number of rows*/
#define  NCOLS          128    /*Number of columns*/
#define  CLASS_SIZE     30     /* Original number of classes */
#define  NCLASS         10     /* Number of new classes */
#define  CLASS0_SIZE    4      /* Number of codevectors in each class */
#define  CLASS1_SIZE    18
#define  CLASS2_SIZE    3
#define  CLASS3_SIZE    3
#define  CLASS4_SIZE    3
#define  CLASS5_SIZE    3
#define  CLASS6_SIZE    3
#define  CLASS7_SIZE    3
#define  CLASS8_SIZE    4
#define  CLASS9_SIZE    4
#define  CLASS10_SIZE   4
#define  CLASS11_SIZE   4
#define  CLASS12_SIZE   4
#define  CLASS13_SIZE   4
#define  CLASS14_SIZE   4
#define  CLASS15_SIZE   4
#define  CLASS16_SIZE   4
#define  CLASS17_SIZE   4
#define  CLASS18_SIZE   4
#define  CLASS19_SIZE   4
#define  CLASS20_SIZE   4
#define  CLASS21_SIZE   4
#define  CLASS22_SIZE   4
#define  CLASS23_SIZE   4
#define  CLASS24_SIZE   4
#define  CLASS25_SIZE   4
#define  CLASS26_SIZE   4
```

```
#define  CLASS27_SIZE     4
#define  CLASS28_SIZE     4
#define  CLASS29_SIZE     4
#define  EDG_SIZE         16     /*maximum size of edge classes*/
#define  MAX_COUNT        1023   /* maximum frequency count */
#define  EOF  (-1)

main (int argc,char *argv[])
{ FILE  *infile;
 int cl_crow[NCOLS];                      /* current classification index row */
 int cl_prow[NCOLS];                      /* previous classification index row */
 int scb_crow[NCOLS];                     /* current sub-codebook index row */
 int scb_crow[NCOLS];                     /* previous sub-codebook index row */
 int cbdiv[CLASS_SIZE+1];                 /* codebook division array */
 int cur_class;                           /* current classification index */
 int new_class;                           /* new classification */
 int cur_index;                           /* current index */
 int class_size;                          /* size of current class */
 int context;                             /* context */
 int *cf_table;                           /* cum frequency table */
 int shdtable[CLASS0_SIZE+1][CLASS0_SIZE+1];         /* shade table */
 int midtable[CLASS1_SIZE+1][CLASS1_SIZE+1];         /* midrange table */
 int edgetable[NCLASS][EDG_SIZE+1][EDG_SIZE+1];      /* edge tables */
 int i, j, count, start;
 long  insize, outsize;

 /* Open input file */
 if (!(infile = fopen (argv [1],"rb")))
   { fprintf (stderr,"\ncannot open %s\n",argv [1]) ;
     exit (0) ;
   }

/* Enter parameters */
 printf("What is the increment (Ki): ");
 scanf("%d",&Ki);

/* Set up codebook division array */
 cbdiv[0] = 0;
 cbdiv[1] = cbdiv[0] + CLASS0_SIZE;
 cbdiv[2] = cbdiv[1] + CLASS1_SIZE;
 cbdiv[3] = cbdiv[2] + CLASS2_SIZE;
```

```
  cbdiv[4] = cbdiv[3] + CLASS3_SIZE;
  cbdiv[5] = cbdiv[4] + CLASS4_SIZE;
  cbdiv[6] = cbdiv[5] + CLASS5_SIZE;
  cbdiv[7] = cbdiv[6] + CLASS6_SIZE;
  cbdiv[8] = cbdiv[7] + CLASS7_SIZE;
  cbdiv[9] = cbdiv[8] + CLASS8_SIZE;
  cbdiv[10] = cbdiv[9] + CLASS9_SIZE;
  cbdiv[11] = cbdiv[10] + CLASS10_SIZE;
  cbdiv[12] = cbdiv[11] + CLASS11_SIZE;
  cbdiv[13] = cbdiv[12] + CLASS12_SIZE;
  cbdiv[14] = cbdiv[13] + CLASS13_SIZE;
  cbdiv[15] = cbdiv[14] + CLASS14_SIZE;
  cbdiv[16] = cbdiv[15] + CLASS15_SIZE;
  cbdiv[17] = cbdiv[16] + CLASS16_SIZE;
  cbdiv[18] = cbdiv[17] + CLASS17_SIZE;
  cbdiv[19] = cbdiv[18] + CLASS18_SIZE;
  cbdiv[20] = cbdiv[19] + CLASS19_SIZE;
  cbdiv[21] = cbdiv[20] + CLASS20_SIZE;
  cbdiv[22] = cbdiv[21] + CLASS21_SIZE;
  cbdiv[23] = cbdiv[22] + CLASS22_SIZE;
  cbdiv[24] = cbdiv[23] + CLASS23_SIZE;
  cbdiv[25] = cbdiv[24] + CLASS24_SIZE;
  cbdiv[26] = cbdiv[25] + CLASS25_SIZE;
  cbdiv[27] = cbdiv[26] + CLASS26_SIZE;
  cbdiv[28] = cbdiv[27] + CLASS27_SIZE;
  cbdiv[29] = cbdiv[28] + CLASS28_SIZE;
  cbdiv[30] = cbdiv[29] + CLASS29_SIZE;

/* Initialize tables */
  for (i = 0; i <= CLASS0_SIZE; i++)
    for (count = 0; count <= CLASS0_SIZE; count++)
      shdtable[i][count] = count;

  for (i = 0; i <= CLASS1_SIZE; i++)
    for (count = 0; count <= CLASS1_SIZE; count++)
      midtable[i][count] = count;

  for (i = 0; i < NCLASS; i++)
    for (j = 0; j <= EDG_SIZE; j++)
      for (count = 0; count <= EDG_SIZE; count++)
        edgetable[i][j][count] = count;
```

```
/* Initialize Parameters */
 start = 1;
 insize = 0;
 outsize = 0;

 for (i=0; i<NROWS; i++)
  { /* Copy current row and read in new row of indices */
   for (j=0; j<NCOLS; j++)
     { cl_prow[j]=crow[j];
      cl_crow[j]=fgetc(infile);
      scb_prow[j]=scb_crow[j];
     }

    for (j = 0; j < NCOLS; j++)
     { cur_index = cl_crow[j];
      insize++;

      /* Find current class */
      for (count=0; count<CLASS_SIZE; count++)
       if ((cur_index >= cbdiv[count]) && (cur_index < cbdiv[count+1]))
         { cur_class = count;
          class_size = cbdiv[count+1] - cbdiv[count];
          break;
         }

      /* renumber starting at zero */
      cur_index = cur_index - cbdiv[cur_class];

      /* Find current classification and adjust sub-codebook index */
      if ((cur_class==0)||(cur_class==1))
        new_class = cur_class;          /* Shade or Midrange */
      else if ((cur_class==2)||(cur_class==4)||(cur_class==6))
        { new_class = 2;                /* Positive Horizontal */
         class_size = CLASS2_SIZE+CLASS4_SIZE+CLASS6_SIZE;
         if (cur_class==4) cur_index += CLASS2_SIZE;
         if (cur_class==6) cur_index += CLASS2_SIZE + CLASS4_SIZE;
        }
      else if ((cur_class==3)||(cur_class==5)||(cur_class==7))
        { new_class = 3;                /* Negative Horizontal */
         class_size = CLASS3_SIZE+CLASS5_SIZE+CLASS7_SIZE;
         if (cur_class==5) cur_index += CLASS3_SIZE;
  if (cur_class==7) cur_index += CLASS3_SIZE + CLASS5_SIZE;
 }
else if ((cur_class==8)||(cur_class==10)||(cur_class==12))
 { new_class = 4;              /* Positive Vertical */
  class_size = CLASS8_SIZE+CLASS10_SIZE+CLASS12_SIZE;
  if (cur_class==10) cur_index += CLASS8_SIZE;
  if (cur_class==12) cur_index += CLASS8_SIZE + CLASS10_SIZE;
 }
else if ((cur_class==9)||(cur_class==11)||(cur_class==13))
 { new_class = 5;              /* Negative Vertical */
  class_size = CLASS9_SIZE+CLASS11_SIZE+CLASS13_SIZE;
  if (cur_class==11) cur_index += CLASS9_SIZE;
  if (cur_class==13) cur_index += CLASS9_SIZE + CLASS11_SIZE;
 }
else if ((cur_class==14)||(cur_class==16)||
     (cur_class==18)||(cur_class==20))
 { new_class = 6;              /* Positive 45 degrees */
  class_size = CLASS14_SIZE+CLASS16_SIZE+
          CLASS18_SIZE+CLASS20_SIZE;
  if (cur_class==16) cur_index += CLASS14_SIZE;
  if (cur_class==18) cur_index += CLASS14_SIZE+CLASS16_SIZE;
  if (cur_class==20) cur_index += CLASS14_SIZE+
                     CLASS16_SIZE+CLASS18_SIZE;
 }
else if ((cur_class==15)||(cur_class==17)||
     (cur_class==19)||(cur_class==21))
 { new_class = 7;              /* Negative 45 degrees */
  class_size = CLASS15_SIZE+CLASS17_SIZE+
          CLASS19_SIZE+CLASS21_SIZE;
  if (cur_class==17) cur_index += CLASS15_SIZE;
  if (cur_class==19) cur_index += CLASS15_SIZE+CLASS17_SIZE;
  if (cur_class==21) cur_index += CLASS15_SIZE+
                     CLASS17_SIZE+CLASS19_SIZE;
 }
else if ((cur_class==22)||(cur_class==24)||
     (cur_class==26)||(cur_class==28))
 { new_class = 8;              /* Positive 135 degrees */
  class_size = CLASS22_SIZE+CLASS24_SIZE+
          CLASS26_SIZE+CLASS28_SIZE;
  if (cur_class==24) cur_index += CLASS22_SIZE;
  if (cur_class==26) cur_index += CLASS22_SIZE+CLASS24_SIZE;
```

```
    if (cur_class==28) cur_index += CLASS22_SIZE+
                                          CLASS24_SIZE+CLASS26_SIZE;
   }
  else
   { new_class = 9;              /* Negative 135 degrees */
    class_size = CLASS23_SIZE+CLASS25_SIZE+
                     CLASS27_SIZE+CLASS29_SIZE;
    if (cur_class==25) cur_index += CLASS23_SIZE;
    if (cur_class==27) cur_index += CLASS23_SIZE+CLASS25_SIZE;
    if (cur_class==29) cur_index += CLASS23_SIZE+
                                          CLASS25_SIZE+CLASS27_SIZE;
   }

  cl_crow[j] = new_class;
  scb_crow[j] = cur_index;

/* Context Selection */
  /* Check North neighbor */
  if ((i > 0) && (new_class == cl_prow[j]))
    context = scb_prow[j];

  /* Check West neighbor */
  else if ((j > 0) && (new_class == cl_crow[j-1]))
    context = scb_crow[j-1];

  /* Check Northeast neighbor */
  else if ((i > 0) && (j < 127) && (new_class == cl_prow[j+1]))
    context = scb_prow[j+1];

  /* Check Northwest neighbor */
  else if ((i > 0) && (j > 0) && (new_class == cl_prow[j-1]))
    context = scb_prow[j-1];

  /* If no match, choose null context */
  else
    context = class_size;

/* Choose appropriate table */
  if (new_class == 0)
    cf_table = shdtable[context];
```

```
          else if (new_class == 1)
            cf_table = midtable[context];
          else
            cf_table = edgetable[new_class][context];

        /* Code sub-codebook index */
          outsize += Jcode(cf_table,cur_index,class_size,&start);

        /* Update statistics */
          for (count = cur_index + 1; count <= class_size; count++)
            cf_table[count] += Ki;

        /* If total count is too large, rescale */
          if (cf_table[class_size] >= MAX_COUNT)
            for (count = 1; count <= class_size; count++)
              { cf_table[count] /= 2;
                if ((cf_table[count] - cf_table[count-1]) <= 0)
                  cf_table[count] = cf_table[count-1] + 1;
              }
        } /*for j*/
      } /*for i*/

  /* Code end-of-message */
    outsize += Jcode(cf_table,EOF,class_size,&start);

    fclose (infile);

    printf ("\nInput symbols   :%ld",insize);
    printf ("\nOutput bytes  :%ld",outsize);

  } /* main */
```