ADAPTIVE VECTOR QUANTIZATION FOR THE CODING OF NONSTATIONARY SOURCES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

James Edwin Fowler, Jr., B.S., M.S.

* * * * *

The Ohio State University

1996

Dissertation Committee:

Dr. Stanley C. Ahalt, Adviser

Dr. Steven B. Bibyk

Dr. Randolph L. Moses

Approved by

Adviser
Department of Electrical
Engineering

ABSTRACT

Vector quantization (VQ) has long been a popular technique for data compression due in part to results from rate-distortion theory that show that VQ is asymptotically optimal for the coding of a stationary source. The fact most sources of practical interest are, in reality, nonstationary has prompted a search for more general VQ algorithms that are capable of adapting to changing source statistics as the coding progresses. Such algorithms, collectively known as adaptive vector quantization (AVQ), apply often heuristically motivated modifications to VQ to achieve coding schemes that adapt to the changing statistics of nonstationary sources.

The work presented here describes a mathematical model of communication systems using AVQ which we believe to be the first of its kind to appear. This model accurately describes the operation of each of the components of an AVQ communication system, is sufficiently general to apply to all reported AVQ algorithms, and allows the classification of these prior algorithms in a taxonomy, which is also presented. An additional contribution of our work is the development of a new AVQ algorithm, called generalized threshold replenishment (GTR), which is based on the explicit consideration of both rate and distortion measures concurrently with the coding of the source. We show in a series of experimental results obtained for both an artificial nonstationary source as well as for real image-sequence data that the GTR algorithm achieves rate-distortion performance superior to that of other reported AVQ algorithms, particularly for low-rate coding.

To my fiancée, Lisa, and my parents

ACKNOWLEDGMENTS

I thank my adviser, Dr. Stanley C. Ahalt, for the invaluable support and limitless encouragement, both academic and otherwise, which has made this work possible. I also thank Dr. Lee C. Potter, Dr. Steven B. Bibyk, Dr. Randolph L. Moses, and Dr. Urbashi Mitra for helpful discussion and insight, and Mark Hanes for all his computer-related help. I thank AT&T for providing financial support in the form of an AT&T Bell Laboratories Ph.D. Scholarship, for which I am extremely grateful. Finally, I thank my fiancée, Lisa, for her encouragement and patience during the writing of this dissertation.

VITA

January 19, 1967	Born—Huntsville, Alabama, USA
1985 – 1990	Ohio Academic Scholar, The Ohio State University Columbus, Ohio, USA
March 1990	B.S. Computer and Information Science Engineering, Summa Cum Laude The Ohio State University Columbus, Ohio, USA
1990 – Present	University Fellow The Ohio State University Columbus, Ohio, USA
June 1992	M.S. Electrical Engineering, The Ohio State University Columbus, Ohio, USA
1991 – 1993	NASA Space Grant/OAI Fellowship, Ohio Space Grant Consortium
1993 – Present	AT&T Ph.D. Scholarship, AT&T Foundation

PUBLICATIONS

Research Publications

J. E. Fowler, K. C. Adkins, S. B. Bibyk, and S. C. Ahalt, "Real-Time Video Compression Using Differential Vector Quantization," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 14-24, February, 1995.

J. E. Fowler, M. R. Carbonara, and S. C. Ahalt, “ Image Coding Using Differential Vector Quantization,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, no. 5, pp. 350-367, October 1993.

C. Mills, S. C. Ahalt, and J. Fowler, “Compiled Instruction Set Simulation,” *Software–Practice and Experience*, vol. 21, pp. 877-889, August 1991.

FIELDS OF STUDY

Major Field: Electrical Engineering

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Digital systems for signal-processing and communications applications have enjoyed widespread use in recent decades due to their low cost of implementation, ease of programming, and fault tolerance [1]. The fact that most real-world signals are continuous-time, continuous-amplitude waveforms necessitates the conversion to discrete-time, discrete-amplitude representations suitable for digital processing. Although Nyquist sampling theorems argue that converting from continuous time to discrete time can be accomplished without distorting low-pass signals common in real applications, the quantization of continuous amplitudes to discrete values necessarily introduces distortion [1]. Thus the issue of lossy quantization, or more generally that of *source coding*, arises in nearly every application using digital systems.

Over the last 20 years, vector quantization (VQ) has received significant attention as a powerful technique for source coding. As we will see later, VQ is theoretically attractive due to results from rate-distortion theory that show that VQ is asymptotically optimal for the coding of a data source whose statistics are stationary in time. Although VQ has been successfully applied to the coding of speech, audio, image, and video data [2], these sources can rarely be assumed to be stationary in practice, leading to a gap between the performance predicted by theory and that actually obtained in real implementations. Indeed, the nonstationary nature of the sources used by many practical applications has prompted a search for more general VQ algorithms that are capable of adapting to changing source statistics as the coding progresses. Such algorithms use what we call adaptive vector quantization (AVQ).

There have been a number of AVQ algorithms proposed in engineering literature since 1982. Because of the scarcity of rate-distortion results for nonstationary sources, most of these algorithms have been heuristically, rather than analytically, motivated. As a result, the theoretical performance potential of AVQ has been little understood to date. Additionally, there has been little attempt to develop a model of AVQ sufficiently general to encompass the operation of all existing AVQ algorithms or to compare the performances associated with them. The research described in this dissertation is intended to address these shortcomings.

One of the key contributions of the work presented here is the development of a mathematical model for communication systems using AVQ. This model accurately describes the operation of each of the components of an AVQ communication system while being sufficiently general to apply to all reported AVQ algorithms. We discuss the general operation of the components included in the model as well as describe the simplifying assumptions typically used in the practical implementation of those components. The unifying framework resulting from our AVQ model leads naturally to a taxonomy of existing AVQ algorithms. To our knowledge, the mathematical model of AVQ presented here, as well as the taxonomy of existing AVQ algorithms, is the first such discussion to appear, and, as such, represents original work in the field of AVQ.

Another important contribution included here is the development of a new AVQ algorithm called generalized threshold replenishment (GTR). Inspired by the analysis leading to the AVQ taxonomy, the GTR algorithm is one of two AVQ algorithms to feature an explicit consideration of both rate and distortion, the two quantities that measure the performance of a coding algorithm. We show in a series of experimental results that the GTR algorithm achieves rate-distortion performance superior to that of other reported AVQ algorithms, particularly for low-rate coding.

This body of experimental results is the final contribution of the work presented here. In these results, we evaluate the performances of AVQ algorithms against each other as well as make comparisons with nonadaptive VQ. We consider not only an artificially generated nonstationary source, but also real data in the form of an image sequence. We believe that the comparisons presented here between the previously reported AVQ algorithms are unique to this dissertation, as we know of no similar performance evaluation appearing in prior literature. These results show that the GTR algorithm consistently achieves low-rate-coding performance superior to that of other AVQ algorithms. Since a severe rate constraint is implicit in many applications of current interest, such as network and wireless communications, low-rate coding techniques will play an important role in the future of communications. We conclude that algorithms like GTR will be instrumental to the incorporation of AVQ techniques into practical low-rate coding implementations.

The organization of the remainder of this dissertation is as follows. In general, we discuss relevant background material in Chapters 2 through 5, while Chapters 6 through 9 focus on the original contributions of our work. Specifically, in Chapter 2, we review relevant mathematics from probability theory. We then continue the background review in Chapter 3 by giving an overview of information theory, the foundation of the modern mathematical theory of communication. This discussion leads naturally into Chapter 4 which surveys rate-distortion theory, the extension of information theory to the case of lossy communication. The final chapter of background material, Chapter 5, provides an extensive review of nonadaptive VQ. Included in this chapter is the application of source-coding theorems described in Chapter 4 to establish the optimal source-coding capability of VQ for stationary sources.

The presentation of the original contributions of our work begins in Chapter 6. In that chapter, we develop a mathematical theory of AVQ that parallels that of nonadaptive VQ given in Chapter 5. This mathematical theory enables the development of a model of AVQ communication systems, included at the end of Chapter 6. Then, in Chapter 7, the AVQ algorithms that have appeared in prior literature are described in light of our AVQ communication-system model. The source-coding theorems of Chapter 4 coupled with the communication-system model of Chapter 6 inspire a categorization of these prior AVQ algorithms; the presentation of the resulting taxonomy concludes Chapter 7. We detail our proposed GTR algorithm in Chapter 8 and then present extensive experimental results in Chapter 9. Finally, we make some concluding remarks in Chapter 10.

CHAPTER 2

PROBABILITY THEORY AND COMMUNICATION SYSTEMS

In this chapter, we lay some of the groundwork for the discussions to come later. In Sections 2.1 and 2.2, we briefly describe some of the concepts from probability theory that form the mathematical foundation necessary to these later discussions. We conclude this chapter with Section 2.3 in which probability theory is used to construct a mathematical representation of real communication systems.

2.1 Random Variables and Random Vectors

In this and the next section, basic concepts from probability theory are presented. Because the motivation of these two sections is solely to introduce notation and terminology, a basic understanding of probability theory is assumed. Unless otherwise noted, the definitions in these two sections come from Gray [3]; for a more detailed investigation, consult Papoulis [4]. Additionally, a mathematically rigorous development of probability theory in terms of measure theory can be found in Halmos [5].

A *random variable* is a number, X, assigned to every outcome of an experiment [4]. Random variables can be either *continuous* or *discrete*. Continuous random variables have a *probability density function* (pdf), $f_X(x)$, over all $x \in \Re$, where $\Re$ is the set of real numbers. We assume that the pdf of a continuous random variable exists, is finite, and is continuous over all $\Re$, and

$$\int_{\Re} f_X(x)\,dx = 1. \tag{2.1}$$

A discrete random variable has a *probability mass function* (pmf), $p_X(x)$, defined on finite alphabet $\mathcal{X} \subset \Re$ such that

$$\sum_{x \in \mathcal{X}} p_X(x) = 1. \tag{2.2}$$

A *random vector*, $\mathbf{X}$, is a finite collection of random variables [3]. If all the vector components of a random vector are continuous random variables, the random vector is *continuous.* Such a random vector has a pdf, $f_{\mathbf{X}}(\mathbf{x})$, defined for each vector $\mathbf{x}$ in the N-dimensional Euclidean space $\Re^N$, where N is the dimension of the random vector. The pdf must satisfy

$$\int_{\Re^N} f_{\mathbf{X}}(\mathbf{x})\,d\mathbf{x} = 1. \tag{2.3}$$

The pdf of a continuous random vector is the joint probability density of all of the continuous random variables from which the random vector is composed. Similarly, if all the vector components of a random vector are discrete, the random vector is *discrete.* Such a random vector has a pmf, $p_{\mathbf{X}}(\mathbf{x})$, defined for each N-dimensional vector $\mathbf{x}$ in a finite alphabet of vectors, $\mathcal{X}$, so that

$$\sum_{\mathbf{x} \in \mathcal{X}} p_{\mathbf{X}}(\mathbf{x}) = 1. \tag{2.4}$$

The pmf of a discrete random vector is the joint pmf of all of its constituent discrete random variables.

The next section continues with more definitions from probability theory by considering sequences of random variables and vectors. Such sequences are called random processes.

2.2 Random Processes and Random-Vector Processes

A *random process* is a possibly infinite, indexed collection of random variables [3]. We denote a random process as X_n, where n is the index of the process. Index n is usually considered to be a value of "time." We will consider only the case in which n is discrete; i.e., X_n is a *discrete-time* random process. Occasionally the notation X_n will refer to the entire random process; at other times, it will refer to the value of the process at a specific time n (in which case X_n denotes a single random variable). We will endeavor to carefully distinguish between these two cases in the remainder of this document. A random process can be either *continuously distributed* or *discretely distributed*, depending on whether its constituent random variables are continuous or discrete, respectively. Since we are considering only discrete-time random processes, we will generally refer to a continuously distributed, discrete-time random process as a continuous random process and a discretely distributed, discrete-time random process as a discrete random process.

The statistics of random processes are described by joint pdf's or pmf's. The r^{th}-order pdf of a continuous random process X_n is the joint pdf, $f_{X_n}(x_1, x_2, \ldots, x_r)$, of the continuous random variables X_1, X_2, $\ldots$, X_r of the process. Similarly, the r^{th}-order pmf of a discrete random process is the joint pmf, $p_{X_n}(x_1, x_2, \ldots, x_r)$, of the discrete random variables X_1, X_2, $\ldots$, X_r of the process. We will usually assume that a random process exists over all time $n \geq 1$. Thus, the complete statistics of a random process are specified by the infinite-order pdf or pmf which is the joint pdf or pmf of the random variables over all time indices $n \geq 1$.

A random process is *stationary* if the pdf or pmf is invariant to time shifts [4]. That is, for all shifts $t \geq 0$ and for all orders r,

$$f_{X_n}(x_1, x_2, \ldots, x_r) = f_{X_n}(x_{t+1}, x_{t+2}, \ldots, x_{t+r}), \tag{2.5}$$

for stationary, continuous random processes, or

$$p_{X_n}(x_1, x_2, \ldots, x_r) = p_{X_n}(x_{t+1}, x_{t+2}, \ldots, x_{t+r}), \tag{2.6}$$

for stationary, discrete random processes. Random process that are not stationary are called *nonstationary*. One simple, yet important, class of stationary random processes are the independent and identically distributed (iid) processes. A continuous iid process has r^{th}-order pdf

$$f_{X_n}(x_1, x_2, \ldots, x_r) = \prod_{i=1}^{r} f_{X_1}(x_i); \tag{2.7}$$

that is, each random variable X_i of the process has the same pdf and is independent from all other random variables of the process. Discretely distributed iid processes are defined similarly.

A *random-vector process*, $\mathbf{X}_n$, is an indexed collection of random vectors. Again, a random-vector process can be either *continuously distributed* or *discretely distributed*, depending on whether its constituent random vectors are continuously or discretely distributed, respectively. In the remainder of this document, we will consider only discrete-time random-vector processes. We will generally refer to a continuously distributed, discrete-time random-vector process as a continuous random-vector process and a discretely distributed, discrete-time random-vector process as a discrete random-vector process. The r^{th}-order probability density or mass function of a random-vector process is the joint density or mass function of the r random vectors $\mathbf{X}_1$, $\mathbf{X}_2$, ... , $\mathbf{X}_r$ of the process. Stationarity for random vector processes is defined similarly to Equations 2.5 and 2.6.

In the next section, we will illustrate how the definitions given up to this point are used to model communication systems. Later, this model will allow us to employ certain mathematical results to determine the theoretical performance limits of real communication systems.

2.3 A Model of Communication Systems

Random variables and random processes are the basic building blocks of mathematical models of communication systems. Figure 2.1 depicts the communication-system model used by Berger in his development [6] of rate-distortion theory. Although not a perfect representation of real communication systems, this model captures many of the attributes of physical systems that are of interest to communication-systems designers while providing a convenient basis for the derivation of theoretical performance results.

The *source* shown in Figure 2.1 is the origin of information to be communicated through the system. In the real world, there are many different types of information sources; however, we represent each as follows. At any given time, the source outputs a *symbol* X. The set of all possible output symbols is the *source alphabet*, $\mathcal{X}$. We assume that we do not know in advance what symbol the source will output at a given time; however, we may have some knowledge of the likelihood of the occurrence of the symbols. The natural model of the source in this case is either a random variable

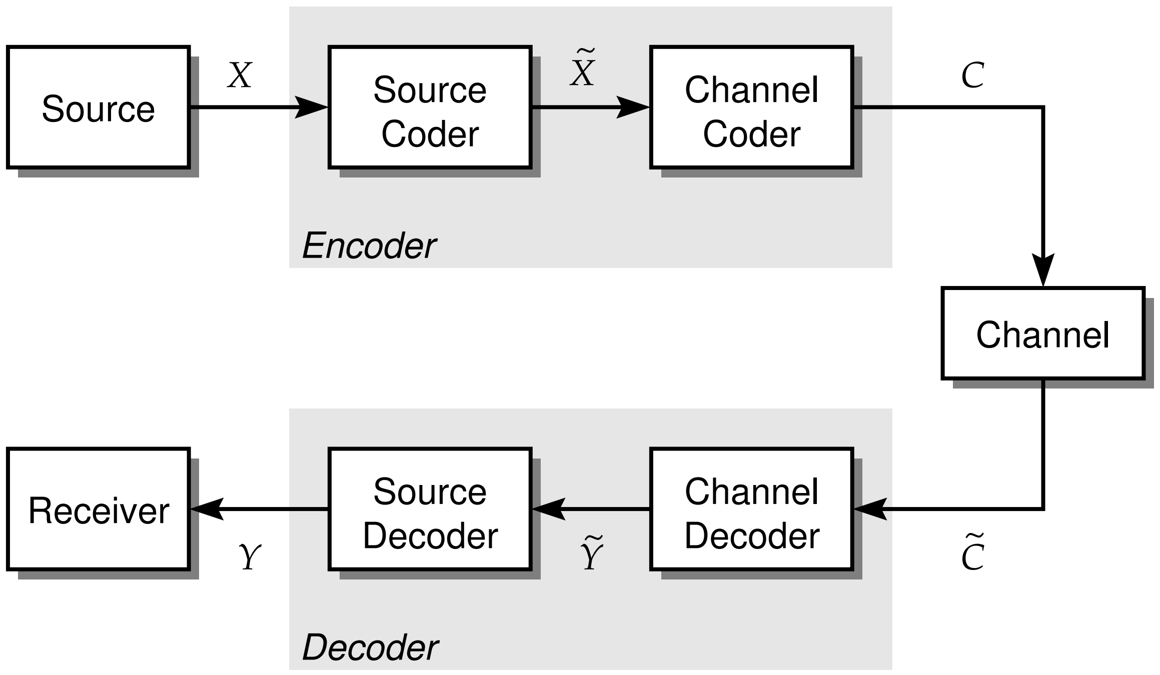


Figure 2.1: A model of a communication system (redrawn and revised from [6])

(if we are interested in the operation of the system at only one time) or a random process (to describe the system's operation over a length of time). The probability density or mass function of the random variable or process describes the imperfect knowledge of the output of the source.

The *receiver* is the destination of the information communicated through the system. In real systems, the receiver is usually a person or a machine. In either case, the *performance* of the communication system is measured by how the receiver interprets the output of the system, symbol Y, when the source is communicating X.

The performance of a communication system will depend on the operation of each of its components in conjunction with the behavior of the medium through which they communicate. As shown in Figure 2.1, the communication system is divided into two major components, an *encoder* and a *decoder*, which communicate through a *channel.* The encoder translates a source symbol X to a channel symbol C which is transmitted through the channel. The decoder perceives a channel symbol $\tilde{C}$ (possibly different from C) which it converts to an estimate Y of the original source symbol X.

The channel is a mathematical model of the medium over which communication occurs. As the channel usually describes a physical medium which is fixed in advance of the design of the communication system, the designer of the system usually has little or no control over the behavior of the channel model. The problem faced by the designer is to develop a system which delivers good performance for the prespecified channel. In order to facilitate this design task, it is common to further partition the encoder and decoder as is shown in Figure 2.1 [6]. The encoder can be considered to be composed of a *source coder*, which manipulates symbols from the source, and a *channel coder*, which prepares symbols for communication over the channel. Typically, the source-coder operation is some form of *data compression*, which results in a more efficient expression of the information from the source. The channel coder usually introduces some form of *error-protection coding* that attempts to ameliorate the effects of the channel on the transmitted symbol. Likewise, the decoder design is partitioned into source-decoder and channel-decoder components which are the inverse operations of those of the encoder.

The partitioning of the encoder allows the design of each component to be conducted independently. The remainder of this text is devoted to the design of the source coder. We make the facilitating, albeit unrealistic, assumption that the channel is *errorless.* That is, we assume that the channel faithfully transfers the channel symbol from the encoder to the decoder, so that $\tilde{C} = C$. Consequently, we assume

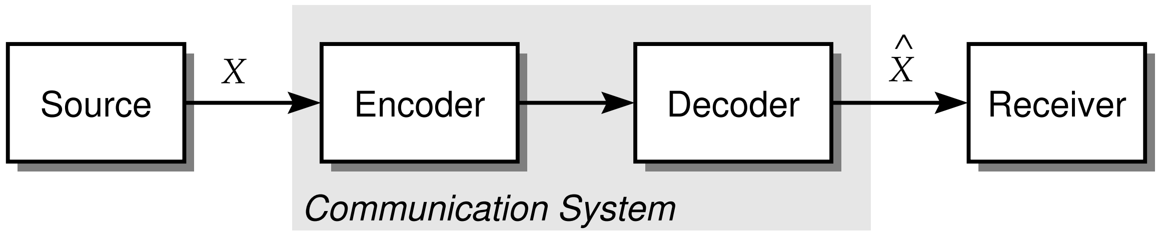


Figure 2.2: Block diagram of the simplified communication-system model

that we can design a perfectly inverting channel coder/decoder pair so that $\tilde{Y} = \tilde{X}$ in Figure 2.1. Furthermore, since Y is an estimate of X that differs only due to effects of the source coder/decoder pair, we will denote Y as $\hat{X}$ in the remainder of this document. These assumptions result in a simplified communication-system model, shown in Figure 2.2, that allows us to focus on the design of the source coder while ignoring channel behavior. The source coder that we design using this simplified model may later be combined with a sophisticated channel-coding scheme as necessary to combat any deleterious effects of the channel.

As stated before, if we are interested in the behavior of the communication system at only one time, we will model the source as a random variable X. Consequently, the output of the communication system is also a random variable, namely $\hat{X}$. However, in most realistic cases, it is the performance of the system over a period of time that is of interest. Thus, we will usually consider the source to be a sequence of random variables; i.e., a random process X_n, where n is a discrete time index. The output of the communication system is also a random process, $\hat{X}_n$. In the next chapter, we present some concepts from information theory that use random processes to measure quantitatively the flow of "information" from the source to the receiver in our communication-system model.

CHAPTER 3

INFORMATION THEORY

As mentioned in the previous chapter, random variables and random processes are the foundation of the model we will use in the analysis of real communication systems. This chapter presents several mathematical mechanisms that operate on such random variables and random processes. As we discuss below, these concepts, when applied to communication-system models, provide a concrete means of quantifying "information," the abstract notion of what is transferred from the source to the receiver during communication. Not surprisingly, the field of mathematics from which these ideas spring is known as information theory.

Information theory had its genesis in the classic paper [7] by Shannon. The concepts discussed in this chapter can be traced to that paper; however, the presentation given here follows the somewhat more accessible treatment given by Cover and Thomas [8]. Our discussion will be brief and no proofs will be given; for a more thorough presentation, refer to Cover and Thomas [8] or Gray [9]. In this chapter, we first define the key elements of classic information theory that will be useful later and follow with some of the resulting implications toward communication systems. More precisely, in Sections 3.1 and 3.2, we define, respectively, the entropy of a discrete random variable and the entropy rate of a discrete random process. We conclude this chapter by considering, in Section 3.3, the theoretical performance limits of noiseless coding of discrete random processes as dictated by classic information theory. The coding of continuous random processes will be investigated later in the Chapter 4 when we present rate-distortion theory, an extension of classic information theory to the more general case of lossy coding.

3.1 Discrete Entropy, Differential Entropy, and Mutual Information

The basic measure of the information content of a random variable is called *entropy* [7, 8]. Although it can be argued that a single mathematical expression cannot possibly capture all the meaning of the concept of "information," the definition of entropy, as given below, does have many of the properties that, by intuition, a measure of information should possess [8]. We first consider the case of the entropy of a discrete random variable.

Assume that X is a discrete random variable with finite alphabet $\mathcal{X}$ and pmf $p_X(x)$ for each $x \in \mathcal{X}$. The *discrete entropy* [8] of random variable X is defined as

$$H(X) \triangleq -\sum_{x\in\mathcal{X}} p_X(x) \log_2 p_X(x). \tag{3.1}$$

Since the logarithm in Equation 3.1 is to the base 2, the units of discrete entropy are considered to be "bits" [8]. In order that events of zero probability do not affect the discrete entropy, we follow the usual convention that $0 \log_2 0 \triangleq 0$. The discrete entropy will sometimes be called simply *entropy* when no confusion may result between it and differential entropy, the corresponding quantity for continuous random variables which we will discuss below.

For any discrete random variable X, the following expression holds:

$$0 \leq H(X) \leq \log_2 |\mathcal{X}|, \tag{3.2}$$

where $|\mathcal{X}|$ is the number of symbols contained in finite alphabet $\mathcal{X}$. Equality on the left of Equation 3.2 is achieved if and only if $p_X(x) = 1$ for one $x \in \mathcal{X}$. That is, if a symbol is certain to occur, the entropy, and thus the "information," contained in the random variable is zero. Equality on the right of Equation 3.2 is achieved if and only if random variable X is a uniform random variable, in which case $p_X(x) = \frac{1}{|\mathcal{X}|}$ for all $x \in \mathcal{X}$. We see that the amount of information conveyed by a random variable is tied to its "randomness" or "uncertainty"—the more random the variable is, the more "information" will be required to describe its value. Maximum entropy is achieved when a random variable has maximum "randomness;" i.e., when it is uniformly distributed.

Several definitions involving multiple discrete random variables follow directly from Equation 3.1. The *joint entropy* [8] of two discrete random variables X and Y is

$$H(X,Y) \triangleq -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p_{XY}(x,y)\log_2 p_{XY}(x,y), \tag{3.3}$$

where $p_{XY}(x,y)$ is the joint pmf of the two random variables. The joint entropy is less than or equal to the sum of the entropies of the individual random variables; i.e.,

$$H(X,Y) \leq H(X) + H(Y), \tag{3.4}$$

with equality if and only if X and Y are independent. It follows from Equation 3.3 and the definition of a random vector (Section 2.1) that the entropy of a discrete random vector $\mathbf{X}$ is

$$H(\mathbf{X}) \triangleq -\sum_{\mathbf{x}\in\mathcal{X}} p_{\mathbf{X}}(\mathbf{x}) \log_2 p_{\mathbf{X}}(\mathbf{x}). \tag{3.5}$$

The *conditional entropy* [8] of discrete random variable Y when given discrete random variable X is

$$H(Y \mid X) \triangleq -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p_{XY}(x,y) \log_2 p_{Y\,|\,X}(y \mid X = x), \tag{3.6}$$

where $p_{Y\,|\,X}(y \mid X = x)$ is the conditional pmf of Y given the event that random variable X equals x.

We now turn to the case of a measure of information for continuous random variables. We define the *differential entropy* [8] of continuous random variable X as

$$h(X) \triangleq -\int_{\mathcal{S}_X} f_X(x) \log_2 f_X(x)\, dx, \tag{3.7}$$

where $\mathcal{S}_X$ is the support set of X, the set of points where $f_X(x) > 0$. As in the case of discrete entropy, differential entropy has the units of "bits;" however, differential entropy may be negative or infinite in some cases [3]. Although such values of differential entropy may be counterintuitive, they should not be interpreted as meaning that the random variable has negative or infinite "information" content. Differential entropy is a relative scale of information that has no absolute "zero" value. On the other hand, discrete entropy is an absolute scale; this fact is one of the most important differences between the discrete and differential entropies.

The entities for multiple continuous random variables corresponding to Equations 3.3 through 3.6 are, as expected, the *joint differential entropy* [8],

$$h(X,Y) \triangleq -\iint\limits_{\mathcal{S}_{XY}} f_{XY}(x,y) \log_2 f_{XY}(x,y)\, dx\, dy, \tag{3.8}$$

the differential entropy of a continuous random vector,

$$h(\mathbf{X}) \triangleq -\int_{\mathcal{S}_{\mathbf{X}}} f_{\mathbf{X}}(\mathbf{x}) \log_2 f_{\mathbf{X}}(\mathbf{x})\, d\mathbf{x}, \tag{3.9}$$

and the *conditional differential entropy* [8],

$$h(Y \mid X) \triangleq -\iint\limits_{\mathcal{S}_{XY}} f_{XY}(x,y) \log_2 f_{Y\,|\,X}(y \mid X = x)\, dx\, dy. \tag{3.10}$$

The entropy (discrete or differential) of a random variable gives a measure of the amount of information possessed by that one random variable; the mutual information is a measure of the amount of information that one random variable has about another random variable. We define the *mutual information* [8] of discrete random variables X and Y as

$$I(X;Y) \triangleq -\sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p_{XY}(x,y) \log_2 \frac{p_{XY}(x,y)}{p_X(x) p_Y(y)}. \tag{3.11}$$

From elementary operations involving Equation 3.1 and the properties of pmf's, we have the following set of relationships among mutual information and discrete entropy:

$$\begin{aligned} I(X;Y) &= I(Y;X) \\ &= H(X) - H(X \mid Y) \\ &= H(Y) - H(Y \mid X) \\ &= H(X) + H(Y) - H(X,Y) \\ &\geq 0. \end{aligned} \tag{3.12}$$

The mutual information between two continuous random variables is

$$I(X;Y) \triangleq -\iint\limits_{\mathcal{S}_{XY}} f_{XY}(x,y) \log_2 \frac{f_{XY}(x,y)}{f_X(x) f_Y(y)}\, dx\, dy. \tag{3.13}$$

The relations between mutual information and differential entropy are the same as those for discrete entropy:

$$\begin{aligned} I(X;Y) &= I(Y;X) \\ &= h(X) - h(X \mid Y) \\ &= h(Y) - h(Y \mid X) \\ &= h(X) + h(Y) - h(X,Y) \\ &\geq 0. \end{aligned} \tag{3.14}$$

We will occasionally need to express the mutual information between a continuous random variable X and a discrete random variable Y. We define the mutual information in this case as

$$I(X;Y) \triangleq H(Y) - H(Y \mid X), \tag{3.15}$$

where

$$H(Y \mid X) \triangleq -\int_{\mathcal{S}_X} f_X(x) \left[\sum_{y \in \mathcal{Y}} p_{Y \mid X}(y \mid X = x) \log_2 p_{Y \mid X}(y \mid X = x) \right] dx \tag{3.16}$$

and $p_{Y \mid X}(y \mid X = x)$ is the conditional pmf of Y given that $X = x$ [3].

The discrete and differential entropies measure the information of random variables. However, as discussed in Section 2.3, we will usually be interested in using random processes in our communication-system model. The next section extends the ideas of this section to define the "information rate" of discrete random processes. The case of information rates for continuous random processes will be treated later in Chapter 4.

3.2 Entropy Rates of Discrete Random Processes

The *entropy rate* of a discrete random process gives an idea of the average rate at which the source produces information. More accurately, the entropy rate describes how fast the entropy of a sequence of discrete random variables grows as the length of the sequence increases [8].

The *entropy rate* [8] of discrete random process X_n is defined as

$$\mathcal{H}(X_n) \triangleq \lim_{N\to\infty} \frac{H(X_1, X_2, \ldots, X_N)}{N}, \tag{3.17}$$

when the limit exists. If the random process is stationary as defined by Equation 2.6, it can be shown that the limit in Equation 3.17 always exists [8, 6]. By Equation 3.4, we then have

$$\mathcal{H}(X_n) \le \lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} H(X_i). \tag{3.18}$$

However, $H(X_i) \le \log_2 |\mathcal{X}|$ by Equation 3.2. Thus, for a stationary process,

$$\mathcal{H}(X_n) \le \log_2 |\mathcal{X}|. \tag{3.19}$$

If a stationary process is iid as described in Section 2.2, then, due to equality in Equation 3.4, the entropy rate becomes

$$\mathcal{H}(X_n) = \lim_{N\to\infty} \frac{H(X_1, X_2, \ldots, X_N)}{N} = H(X_1). \tag{3.20}$$

$H(X_1)$ is commonly called the *first-order entropy* of the random process and is occasionally used as an approximation of the entropy rate of a random process when the entropy rate is unknown or not calculable. This approximation is exact when the process is iid.

The entropy rate of Equation 3.17 is defined for all discrete random processes for which the limit exists. By itself, Equation 3.17 has very little significance; however, in the next section, we will see that the entropy rate of a stationary process provides a lower bound on the average number of bits per symbol achievable by any noiseless coding of the process. It is exactly because of this source-coding property that the entropy rate is considered to be the average rate of information of the source.

3.3 Noiseless Source Coding of Discrete Random Processes

A *source code*, $\mathcal{C}_X = (\mathcal{X}, \mathcal{C}, \zeta)$, for discrete random variable X consists a mapping, ζ, from the finite alphabet $\mathcal{X}$ to a finite set, $\mathcal{C}$, of finite-length strings, c [8]. That is,

$$\mathcal{C} = \{c : c = \zeta(x), x \in \mathcal{X}\}. \tag{3.21}$$

The strings c are called *codewords* and the set $\mathcal{C}$ is called the *codebook*. The codewords are variable-length strings constructed, in general, from a D-ary alphabet, $\mathcal{D}$. For the remainder of this text, we will assume a binary alphabet, $\mathcal{D} = \{0, 1\}$. Additionally, we will consider only the case that source code $\mathcal{C}_X$ is a *prefix code*; i.e., ζ is a one-to-one mapping, and each codeword in the codebook is guaranteed not to be a prefix of another codeword [8]. Prefix codes allow the concatenation of codewords into a stream that can be uniquely decoded into the original stream of source symbols, a property that will be important when we look at the coding of discrete random processes below.

The average length of the prefix code $\mathcal{C}_X$ for discrete random variable X is

$$L(\mathcal{C}_X) \triangleq E\left[l(\zeta(X)\right] = \sum_{x \in \mathcal{X}} p_X(x) l(\zeta(x)), \tag{3.22}$$

where $l(\zeta(x))$ is the length (in bits) of codeword $c = \zeta(x)$. An optimal prefix code, $\mathcal{C}_X^*$, is one that achieves the minimum average length of all prefix codes for random variable X.

The following well known result, due to Shannon [7], states the performance achievable by an optimal prefix code.

THEOREM 3.3.1 (SHANNON'S SOURCE CODING THEOREM)
The average length of an optimal prefix code for a discrete random variable is within 1 bit of the entropy of the random variable. That is,

$$H(X) \leq L(\mathcal{C}_X^*) \leq H(X) + 1. \tag{3.23}$$

A simple method for constructing an optimal prefix code was developed by Huffman [10]; the Huffman code is one type of coding technique known as *entropy coding.* The Huffman code attempts to assign codewords with short lengths to those symbols in $\mathcal{X}$ with a high probability of occurrence so that the average length of the code is close to the entropy of the random variable. Note that this average length is an *ensemble* average—it is the expected length over many instances of the random variable.

Up to this point, we have discussed the coding of a single discrete random variable. The more interesting and practical case is the coding of a discrete random process. However, entropy-coding techniques, including the Huffman code, can be applied to this case as well [8]. We simply block N random variables of random process X_n

together into a random vector,

$$\mathbf{X} = (X_1, X_2, \ldots, X_N) \in \mathcal{X}^N. \tag{3.24}$$

We then define a new prefix code, $\mathcal{C}'_{\mathbf{X}} = (\mathcal{X}^N, \mathcal{C}', \zeta')$, that maps from alphabet $\mathcal{X}^N$ to new codebook $\mathcal{C}'$ of codewords for the random vector. The average length of this new code, per original source symbol, is defined as

$$L(\mathcal{C}'_{\mathbf{X}}) \triangleq \frac{1}{N} E\left[l(\zeta'(\mathbf{X})\right] = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{X}^N} p_{\mathbf{X}}(\mathbf{x}) l(\zeta'(\mathbf{x})). \tag{3.25}$$

$L(\mathcal{C}'_{\mathbf{X}})$ will be called the *rate* of code $\mathcal{C}'_{\mathbf{X}}$. An optimal prefix code for random vector $\mathbf{X}$ will have, from Theorem 3.3.1, a rate satisfying

$$\frac{1}{N} H(\mathbf{X}) \leq L(\mathcal{C}'^{*}_{\mathbf{X}}) \leq \frac{1}{N} H(\mathbf{X}) + \frac{1}{N}, \tag{3.26}$$

or, equivalently,

$$\frac{H(X_1, X_2, \ldots, X_N)}{N} \leq L(\mathcal{C}'^{*}_{\mathbf{X}}) \leq \frac{H(X_1, X_2, \ldots, X_N)}{N} + \frac{1}{N}. \tag{3.27}$$

If the entropy rate of random process X_n exists, then we may take the limit as $N \to \infty$ of Equation 3.27 to find that the rate of an optimal coding of random process X_n is the entropy rate; i.e.,

$$L(\mathcal{C}^{*}_{X_n}) = \mathcal{H}(X_n). \tag{3.28}$$

The entropy rate will exist if X_n is stationary. Thus, the entropy rate is the lower bound on the rate of any coding of a stationary discrete random process. If, additionally, the process is iid, then, by Equation 3.20,

$$L(\mathcal{C}^*_{X_n}) = H(X_1). \tag{3.29}$$

Thus, the first-order entropy is the lower bound on the rate of any coding of a discrete iid random process.

At this point, one caveat is in order. It is common that the optimal code rate of an iid random process, $L(\mathcal{C}^*_{X_n})$ of Equation 3.29, is confused with the optimal average code length of random variable X_1, $L(\mathcal{C}^*_{X_1})$ of Equation 3.23 [11]. Equation 3.29 states that optimal entropy coding, such as Huffman coding, applied with infinite blocking to the entire random process will yield a rate equal to the entropy. However, in most practical cases, designing a Huffman code becomes prohibitively complicated for even a small block length N. Thus, the usual approach for coding an iid process is to apply a Huffman code for each symbol separately taking advantage of the fact that, because Huffman coding produces prefix codes, the concatenation of the codewords can be decoded to the original stream of source symbols. Although Huffman coding is optimal for the coding of each individual symbol, it does not necessarily produce an optimal coding of the iid random process when applied symbol by symbol. It is for this reason that other entropy-coding techniques, such as arithmetic coding [12] and textural-substitution methods (e.g., [13, 14]), often yield better performance in practice.

In this section, we have been concerned with the *lossless* or *noiseless* coding of a discrete random process. That is, referring to the communication system of Figure 2.2, the encoder produces a stream of codes with average length $L(\mathcal{C}^*_{X_n}) = \mathcal{H}(X_n)$ bits per source symbol. These codes are sent to the decoder over the lossless channel. The decoder perfectly reconstructs the original source process so that $\hat{X}_n = X_n$.

Thus, the entropy rate gives a measure of the amount of information that must be communicated between the encoder and the decoder for lossless coding. The following theorem expresses this idea more explicitly.

THEOREM 3.3.2

In order to noiselessly communicate stationary, discrete random process X_n with entropy rate $\mathcal{H}(X_n)$ through the communication system of Figure 2.2, the encoder must send at least an average of $\mathcal{H}(X_n) = \mathcal{H}(\hat{X}_n)$ bits per source symbol to the decoder. It is assumed that the communication occurs over a lossless channel. The rate *of the communication system is then defined to be $\mathcal{H}(X_n)$.*

Theorem 3.3.2 is the special case of the well known joint source-channel theorem [8] applied to a lossless channel.

In the next chapter, we consider the case of continuous random processes. Although one may define a *differential entropy rate* for continuous random processes (see Cover and Thomas [8]), such a definition lacks a source-coding implication (Equation 3.28) similar to that established here for discrete random processes. Since an infinite number of bits is needed to specify the value of a continuous random variable (i.e., the discrete entropy of a continuous random variable can usually be considered to be infinite [3]), it is not possible, in general, to code a continuous random process without introducing some distortion. In the next chapter, we will investigate the theoretical tradeoffs between distortion and code rate that arise from allowing imperfect reconstruction at the decoder.

CHAPTER 4

RATE-DISTORTION THEORY

One of the key results of Shannon's development of information theory as presented in Chapter 3 is the fact that a stationary, discrete random process may be coded with a finite number of bits per source symbol in such a way that the original process may be exactly reconstructed from the code. As we saw in Chapter 3, the lower bound on this code rate is the entropy rate, or, if the source is iid, the discrete entropy. In this chapter, we turn to the case of continuous random processes. Since it takes an infinite number of bits to describe exactly a continuous random variable, noiselessly coding a continuous random process requires, in general, an infinite rate. However, as we will see below, it is possible to code a continuous random process with a finite rate if one allows some amount of distortion to be introduced between the original source and the output of the decoder. *Rate-distortion theory* provides a body of mathematics that describes this tradeoff between rate and distortion in the coding of a continuous random process.

Rate-distortion theory was originally developed by Shannon in the second of his classic papers [15]. Although Shannon's investigation was confined to iid sources, his results have been extended by others to more general cases. These more general results may be found in the book by Berger [6], one of the most complete references on rate-distortion theory. Indeed, most of the discussion found here (Sections 4.1 through 4.4) merely summarizes the key results found in Berger's book. The reader should consult the book itself for proofs and other details missing here.

The organization of this chapter is as follows. In Section 4.1, we discuss distortion measures, mechanisms that allow us to quantify the amount of distortion introduced by a coding of a continuous random process. Then in Sections 4.2 and 4.3, we introduce the concepts of the rate-distortion and distortion-rate functions, the heart of rate-distortion theory. In Section 4.4, we present theorems that provide the rate-distortion and distortion-rate functions operational significance toward the coding of stationary, continuous random processes. We conclude the chapter with Section 4.5 which reviews the few rate-distortion results and source-coding theorems available for nonstationary processes.

4.1 Distortion Measures

To code a continuous random process, we allow the decoder to make an inexact approximation to it. In order to judge the "closeness" of this approximation, we introduce a quantitative rule that measures the amount of difference between the original source process and the process that is output from the decoder. This rule is known as a *distortion measure.*

More specifically, suppose that the encoder input is a sequence of N values; in other words, a vector in N-dimensional Euclidean space,

$$\mathbf{x} = (x_1, x_2, \ldots, x_N). \tag{4.1}$$

The decoder outputs an N-dimensional approximation vector, $\hat{\mathbf{x}}$. The nonnegative *word distortion measure* $d(\mathbf{x}, \hat{\mathbf{x}})$ measures the "difference" between the two vectors.

Many word distortion measures of interest can be expressed as

$$d(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{N} \sum_{i=1}^{N} \rho(x_i, \hat{x}_i), \tag{4.2}$$

where $\rho(x, \hat{x})$ is a scalar distortion measure called the *per-letter distortion.* A word distortion measure of the form of Equation 4.2 is known as a *single-letter distortion measure* [2] since it measures the average distortion per component between two vectors.

The most commonly used distortion measure is the *squared error*, defined as

$$d(\mathbf{x}, \hat{\mathbf{x}}) \triangleq ||\mathbf{x} - \hat{\mathbf{x}}||^2 = (\mathbf{x} - \hat{\mathbf{x}})^T (\mathbf{x} - \hat{\mathbf{x}}) = \sum_{i=1}^{N} (x_i - \hat{x}_i)^2. \tag{4.3}$$

The squared-error distortion measure can be made into a single-letter distortion measure by dividing it by N, in which case it becomes the *mean squared-error distortion* (MSE).

4.2 The Rate-Distortion Function

The rate-distortion and distortion-rate functions for a given continuous random process describe the tradeoff between the rate needed to code the random process and the distortion incurred by the coding. In this section, we define the rate-distortion function for a continuous random process following the discussion given in Chapters 4 and 7 of the book by Berger [6]. We delay the definition of the distortion-rate function to Section 4.3 and the more important source-coding properties implied by the rate-distortion function to the theorems in Sections 4.4 and 4.5.

We define the rate-distortion function as the minimal mutual information achieved by a mapping subject to a constrained distortion. More precisely, suppose we have

N successive values from continuous random process X_n which we group into the N-dimensional continuous random vector $\mathbf{X} = (X_1, X_2, \ldots, X_N)$. We assume that the decoder outputs N values, vector $\mathbf{Y}$, from another continuous random process Y_n. The effect of the joint operation of the encoder and the decoder is to assign an output sequence $\mathbf{Y}$ to each input sequence $\mathbf{X}$. This assignment is described probabilistically by $q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x})$, the conditional density of random vector $\mathbf{Y}$ given random vector $\mathbf{X}$.

For each $q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot)$, there is a joint density between $\mathbf{X}$ and $\mathbf{Y}$ as determined by Bayes' Theorem,

$$f_{\mathbf{XY}}(\mathbf{x},\mathbf{y}) = f_{\mathbf{X}}(\mathbf{x}) q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x}), \tag{4.4}$$

where $f_{\mathbf{X}}(\mathbf{x})$ is the density of $\mathbf{X}$. The single-letter distortion measure $d(\mathbf{X},\mathbf{Y})$ is a random variable over this joint ensemble. The expected value of this distortion measure is called the *average distortion* and is a function of $q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot)$,

$$\begin{aligned}\overline{D}(q) \triangleq E[d(\mathbf{X},\mathbf{Y})] &= \iint f_{\mathbf{XY}}(\mathbf{x},\mathbf{y}) d(\mathbf{x},\mathbf{y})\, d\mathbf{x}\, d\mathbf{y} \\ &= \iint q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x}) f_{\mathbf{X}}(\mathbf{x}) d(\mathbf{x},\mathbf{y})\, d\mathbf{x}\, d\mathbf{y}.\end{aligned} \tag{4.5}$$

A specific $q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot)$ is *D-admissible* if and only if $\overline{D}(q) \leq D$. The set of all D-admissible conditional probability assignments is

$$Q_D \triangleq \left\{ q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x}) : \overline{D}(q) \leq D \right\}. \tag{4.6}$$

The rate-distortion function [6], $R(D)$, for a given continuous random process, X_n, is then defined as

$$R(D) \triangleq \lim_{N\to\infty} R_N(D), \tag{4.7}$$

where

$$R_N(D) \triangleq \frac{1}{N} \inf_{q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot) \in Q_D} I(\mathbf{X};\mathbf{Y}), \tag{4.8}$$

and $I(\mathbf{X};\mathbf{Y})$ is the mutual information between N-dimensional random vectors $\mathbf{X}$ and $\mathbf{Y}$. It can be shown that the limit in Equation 4.7 always exists when X_n is a stationary random process [6]. We will see in Section 4.5 that this limit also exists for certain nonstationary sources. Consequently, the rate-distortion function is well-defined for all stationary random processes as well as for certain nonstationary processes.

One should note that the rate-distortion function as defined above has very little real meaning until appropriate source-coding theorems have been proven. It is only then that the rate-distortion function for a source can be considered to be the minimal rate at which the encoder must send information to the decoder to allow reconstruction at a given amount of average distortion [16]. In the Section 4.4, we will review source-coding theorems that indeed provide this intuitive source-coding property for stationary, continuous random processes. Then in Section 4.5, we will present the limited source-coding results that exist for certain nonstationary random processes. First, however, we define the distortion-rate function which is the inverse of the rate-distortion function and the topic of the next section.

4.3 The Distortion-Rate Function

The distortion-rate function defined in this section is the inverse of the rate-distortion function presented in the previous section. That is, the distortion-rate function is defined as the minimum distortion achievable by a mapping subject to a constrained rate. More precisely, suppose we have stationary, continuous random process X_n whose values are grouped into N-dimensional random vector $\mathbf{X}$ as before. Again, we have a probabilistic assignment, $q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x})$, and an average distortion for single-letter distortion measure $d(\mathbf{X},\mathbf{Y})$,

$$\begin{aligned}\overline{D}(q) = E[d(\mathbf{X},\mathbf{Y})] &= \iint f_{\mathbf{XY}}(\mathbf{x},\mathbf{y})d(\mathbf{x},\mathbf{y})\,d\mathbf{x}\,d\mathbf{y} \\ &= \iint q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x})f_{\mathbf{X}}(\mathbf{x})d(\mathbf{x},\mathbf{y})\,d\mathbf{x}\,d\mathbf{y}.\end{aligned} \tag{4.9}$$

A specific $q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot)$ is *R-admissible* if and only if the mutual information per source symbol is less than or equal to R; i.e., the set of all R-admissible conditional probability assignments is

$$Q_R \triangleq \left\{q_{\mathbf{Y}\,|\,\mathbf{X}}(\mathbf{y}\,|\,\mathbf{X}=\mathbf{x}) : \frac{1}{N}I(\mathbf{X};\mathbf{Y}) \le R\right\}. \tag{4.10}$$

The distortion-rate function [6], $D(R)$, for X_n is then

$$D(R) \triangleq \lim_{N\to\infty} D_N(R) \tag{4.11}$$

where

$$D_N(R) \triangleq \inf_{q_{\mathbf{Y}\,|\,\mathbf{X}}(\cdot)\in Q_R} \overline{D}(q). \tag{4.12}$$

The distortion-rate function as defined in this section and the rate-distortion function as defined in the previous section are inverses of each other [3, 17]. That is, $R_0 = R(D_0)$ for some D_0 and R_0 if and only if $D_0 = D(R_0)$. The fact that the two functions are equivalent yet are defined differently suggests two possible approaches towards the coding of a continuous random process. We will explore this issue further in Section 4.4.

4.4 Source Coding of Stationary, Continuous Random Processes

In this section, we review the source-coding theorems for stationary, continuous random processes. We first assume that the process is also ergodic and return to the more general case later. Briefly, a stationary, *ergodic* process is one whose time averages approach its ensemble averages [2, 4]. For example, the time average of any instance of a mean-ergodic stationary process converges to the ensemble mean of the process. That is, the ensemble mean, $\eta = \eta(t) = E\left[X_t\right]$, of the process X_n is invariant in time due to stationarity, and the time average of the process converges to it in the mean-square sense [2, 4]:

$$\frac{1}{N}\sum_{t=1}^{N} X_t \xrightarrow[N\to\infty]{} \eta. \tag{4.13}$$

In general, there exist two different approaches to source coding of a stationary, ergodic source: the constrained-distortion and constrained-rate approaches. The constrained-distortion approach, covered in Section 4.4.1, involves limiting the allowed distortion and then minimizing the rate of the code. As can be expected, the results derived will be related to the rate-distortion function as defined in Section 4.2. The second, constrained-rate approach, involving the distortion-rate function, will be treated afterwards in Section 4.4.2. We delay the discussion of source coding for stationary, nonergodic processes to Section 4.4.3.

4.4.1 Constrained-Distortion Source Coding

In this section, we present theorems for the constrained-distortion approach to source coding. That is, we limit the allowed distortion and design our communication system using the code with the smallest rate that meets this distortion requirement.

More precisely, let random vector $\mathbf{X} = (X_1, X_2, \ldots, X_N)$ be the first N values of stationary, ergodic random process X_n. Let $\mathcal{C}_{\mathbf{X}} = (\mathcal{C}, \zeta)$ be a *block source code* for random vector $\mathbf{X}$, where $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\}$ is a *codebook* of K N-dimensional *codewords*. The function, ζ, codes each $\mathbf{X}$ as the closest codeword in $\mathcal{C}$; that is,

$$\hat{\mathbf{X}} = \zeta(\mathbf{X}) \triangleq \arg\min_{\mathbf{c}\in\mathcal{C}} d(\mathbf{X}, \mathbf{c}). \tag{4.14}$$

Note that $\hat{\mathbf{X}}$ is a discrete random vector. Suppose that the rate-distortion function, $R(D)$, for X_n is defined for single-letter distortion measure $d(\mathbf{X}, \hat{\mathbf{X}})$,

$$d(\mathbf{X}, \hat{\mathbf{X}}) \triangleq \frac{1}{N}\sum_{i=1}^{N} \rho(X_i, \hat{X}_i), \tag{4.15}$$

and that there exists some $\alpha \in \Re$ such that

$$E\left[\rho(X_i, \alpha)\right] < \infty, \qquad \forall i. \tag{4.16}$$

This α is commonly called a *reference letter*. Equation 4.16 is a technical requirement for the proofs of the following theorems [9, 18], and, as such, will not be discussed further here. We will define the *average distortion* incurred by block source code $\mathcal{C}_{\mathbf{X}}$ as

$$\overline{D}(\mathcal{C}_{\mathbf{X}}) \triangleq E\left[d(\mathbf{X}, \hat{\mathbf{X}})\right] = \int f_{\mathbf{X}}(\mathbf{x}) d(\mathbf{x}, \zeta(\mathbf{x}))\, d\mathbf{x}. \tag{4.17}$$

and the *rate*, $\overline{R}(\mathcal{C}_{\mathbf{X}})$, of the code as

$$\overline{R}(\mathcal{C}_{\mathbf{X}}) \triangleq \frac{1}{N} I(\mathbf{X}; \hat{\mathbf{X}}). \tag{4.18}$$

Because the mutual information measures the amount of information in common between two random vectors, the rate, as defined by Equation 4.18, indicates the amount of information that must be conveyed to represent $\mathbf{X}$ as $\hat{\mathbf{X}}$. However, from Equation 3.15, we can express the rate as

$$\overline{R}(\mathcal{C}_{\mathbf{X}}) = \frac{1}{N} I(\mathbf{X}; \hat{\mathbf{X}}) = \frac{1}{N} H(\hat{\mathbf{X}}) - \frac{1}{N} H(\hat{\mathbf{X}} \mid \mathbf{X}). \tag{4.19}$$

But since $\hat{\mathbf{X}}$ is a deterministic function of $\mathbf{X}$, $H(\hat{\mathbf{X}} \mid \mathbf{X}) = 0$, and we have

$$\overline{R}(\mathcal{C}_{\mathbf{X}}) = \frac{1}{N} H(\hat{\mathbf{X}}) \leq \frac{1}{N} \log_2 K, \tag{4.20}$$

where the inequality comes from Equation 3.2. Note that $\overline{R}(\mathcal{C}_{\mathbf{X}})$, as given by Equation 4.20, becomes the entropy rate of discrete random process $\hat{X}_n$ when N is infinite. Supposing that the above assumptions are valid for any positive N, we have the following theorem.

Theorem 4.4.1 (Positive Source-Coding Theorem (Berger [6]))
For any $\epsilon > 0$ and any $D \geq 0$ such that $R(D) < \infty$, there exists a block source code $\mathcal{C}_{\mathbf{X}}$ for an N sufficiently large such that $\overline{D}(\mathcal{C}_{\mathbf{X}}) \leq D$, and $\frac{1}{N} \log_2 K \leq R(D) + \epsilon$. Consequently, by Equation 4.20, we have $\overline{R}(\mathcal{C}_{\mathbf{X}}) \leq R(D) + \epsilon$ for this $\mathcal{C}_{\mathbf{X}}$.

Theorem 4.4.1 is known as the positive source-coding theorem since it states the existence of a block source code that has a rate no more than $R(D)$ plus an arbitrarily small quantity for a given distortion. The converse is also true, as is given in the following theorem.

Theorem 4.4.2 (Negative Source-Coding Theorem (Berger [6]))
There exist no block source codes of average distortion $\overline{D}(\mathcal{C}_{\mathbf{X}}) \leq D$, where $D \geq 0$, with rate $\overline{R}(\mathcal{C}_{\mathbf{X}}) < R(D)$.

Theorem 4.4.2 applies to block source codes of all block sizes N, finite or infinite. A block source code with an infinite block size represents all coding methods that map an infinite-length random process to an infinite-length code. Consequently, Theorem 4.4.2 also applies to any coding method, including those that are not block-based [6].

Theorem 4.4.1 applies to the first N values of random process X_n where we must choose N sufficiently large so that the rate satisfies $\overline{R}(\mathcal{C}_{\mathbf{X}}) \leq R(D) + \epsilon$ for the desired distortion, D, and accuracy, ϵ. However, because of the stationarity of X_n, we can use this theorem to implement a coding for the entire random process. The details of this coding method follow.

We will denote our coding for stationary random process X_n as $\mathcal{C}_{X_n}$. The output of this code will be discrete random process $\hat{X}_n$, which will be stationary due to the stationarity of X_n. We will define the rate of this code as

$$\overline{R}(\mathcal{C}_{X_n}) \triangleq \lim_{N\to\infty} \frac{1}{N} I(X_1, X_2, \ldots, X_N; \hat{X}_1, \hat{X}_2, \ldots, \hat{X}_N). \tag{4.21}$$

We see that this definition is simply the limit as N goes to infinity of Equation 4.18. Consequently, by Equation 4.20, we can express the rate of our random-process coding as

$$\overline{R}(\mathcal{C}_{X_n}) = \lim_{N\to\infty} \frac{1}{N} H(\hat{X}_1, \hat{X}_2, \ldots, \hat{X}_N), \tag{4.22}$$

which is simply the entropy rate of the coded process; i.e.,

$$\overline{R}(\mathcal{C}_{X_n}) = \mathcal{H}(\hat{X}_n). \tag{4.23}$$

The distortion measure of interest for our random-process code is the average single-letter distortion over all time of the random process,

$$\overline{D}(\mathcal{C}_{X_n}) \triangleq \lim_{N\to\infty} E\left[d(\mathbf{X}, \hat{\mathbf{X}})\right]. \tag{4.24}$$

However, since $d(\mathbf{X}, \hat{\mathbf{X}})$ is a single-letter distortion measure, we have

$$\begin{aligned}\overline{D}(\mathcal{C}_{X_n}) = \lim_{N\to\infty} E\left[d(\mathbf{X}, \hat{\mathbf{X}})\right] &= \lim_{N\to\infty} E\left[\frac{1}{N}\sum_{i=1}^{N}\rho(X_i, \hat{X}_i)\right] \\ &= \lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N} E\left[\rho(X_i, \hat{X}_i)\right],\end{aligned} \tag{4.25}$$

where $\rho(X_i, \hat{X}_i)$ is the per-letter distortion. Because of the stationarity, $E\left[\rho(X_i, \hat{X}_i)\right]$ is independent of time i, so the average single-letter distortion is equal to the average per-letter distortion,

$$\overline{D}(\mathcal{C}_{X_n}) = E\left[\rho(X_i, \hat{X}_i)\right] \triangleq \overline{\rho}(X_n, \hat{X}_n). \tag{4.26}$$

The coding of the random process operates as follows. We partition source process X_n into successive blocks of N values and use the same optimal code, $\mathcal{C}^*_{\mathbf{X}}$, whose existence is guaranteed by Theorem 4.4.1, to code each block. Because X_n is stationary, the successive codewords produced by the code form a stationary discrete random-vector process whose entropy rate is at most $\log_2 K$ bits per vector by Equation 3.19. Consequently, the entropy rate of the output process, $\mathcal{H}(\hat{X}_n)$, is at most $\frac{1}{N}\log_2 K$. Additionally, $\mathcal{H}(\hat{X}_n)$ is at least $R(D)$ by Theorem 4.4.2. By combining the above observations with Theorem 4.4.1, we have

$$R(D) \le \overline{R}(\mathcal{C}_{X_n}) = \mathcal{H}(\hat{X}_n) \le \frac{1}{N}\log_2 K \le R(D) + \epsilon. \tag{4.27}$$

Therefore, the rate of our random-process code is between $R(D)$ and $R(D) + \epsilon$.

Since we have assumed that X_n is ergodic, we have that the time-average per-letter distortion converges to the ensemble average per-letter distortion; i.e.,

$$\tilde{\rho}(X_n, \hat{X}_n) \triangleq \lim_{M\to\infty}\frac{1}{M}\sum_{i=1}^{M}\rho(X_i, \hat{X}_i) \to \overline{\rho}(X_n, \hat{X}_n). \tag{4.28}$$

But the time-average per-letter distortion can be expressed as a time-average single-letter word distortion on the random vectors into which X_n is partitioned; i.e.,

$$\tilde{\rho}(X_n, \hat{X}_n) = \tilde{D}(\mathbf{X}_t, \hat{\mathbf{X}}_t) \triangleq \lim_{T \to \infty} \frac{1}{T} \sum_{j=1}^{T} d(\mathbf{X}_j, \hat{\mathbf{X}}_j), \tag{4.29}$$

where $\mathbf{X}_t$ is the random-vector process formed from our partitioning of the original source, X_n. Since X_n is stationary and ergodic, so is $\mathbf{X}_t$. Consequently, we have

$$\tilde{D}(\mathbf{X}_t, \hat{\mathbf{X}}_t) \to E\left[d(\mathbf{X}_j, \hat{\mathbf{X}}_j)\right] = \overline{D}(\mathcal{C}^*_{\mathbf{X}}). \tag{4.30}$$

Since the code, $\mathcal{C}^*_{\mathbf{X}}$, used by the encoder guarantees that $\overline{D}(\mathcal{C}^*_{\mathbf{X}}) \leq D$, we have

$$\overline{D}(\mathcal{C}_{X_n}) = \overline{\rho}(X_n, \hat{X}_n) \leq D \tag{4.31}$$

for our random-process code.

Now, by increasing the block size N, we can make ϵ in Equation 4.27 as small as desired [6]. Consequently, we see that the rate-distortion function, $R(D)$, is not only a lower bound on the rate for any coding method (as stated in Theorem 4.4.2), but is actually achievable by a block source code with infinite blocking. Therefore, block source coding is, in this sense, optimal—no other coding method achieves a lower rate for a given distortion. One should note that this optimality is only asymptotic: it may be possible for other coding methods to perform better than block source coding when an infinite-length observation of the source is not allowed.

In this section, we have determined that constrained-distortion coding using a block source code is asymptotically optimal for a stationary, ergodic source. In the next section, we present the corresponding results for constrained-rate coding.

4.4.2 Constrained-Rate Coding

In this section, we present theorems for the constrained-rate approach to source coding. That is, we limit the allowed rate and design our communication system using the code with the smallest distortion that meets this rate requirement.

Let us assume that random vector $\mathbf{X}$ consists of the first N values of stationary, ergodic random process X_n as in Section 4.4.1. Suppose that we define a block source code $\mathcal{C}_{\mathbf{X}}$ as before, and the average distortion, $\overline{D}(\mathcal{C}_{\mathbf{X}})$, and rate, $\overline{R}(\mathcal{C}_{\mathbf{X}})$, of this code as was done in Equations 4.17 and 4.20, respectively. Suppose that the distortion-rate function, $D(R)$, is defined for X_n and that the reference-letter condition (Equation 4.16) holds. We then have the following theorems.

Theorem 4.4.3 (Positive Source-Coding Theorem (Gray [3]))
For any $\delta > 0$ and any $R > 0$, there exists a block source code $\mathcal{C}_{\mathbf{X}}$ for an N sufficiently large such that

$$\overline{R}(\mathcal{C}_{\mathbf{X}}) \triangleq \frac{1}{N} H(\hat{\mathbf{X}}) \leq \frac{1}{N} \log_2 K \leq R \tag{4.32}$$

and

$$\begin{aligned} \overline{D}(\mathcal{C}_{\mathbf{X}}) \triangleq E\left[d(\mathbf{X}, \hat{\mathbf{X}})\right] &= \int f_{\mathbf{X}}(\mathbf{x}) d(\mathbf{x}, \zeta(\mathbf{x}))\, d\mathbf{x} \\ &\leq D(R) + \delta. \end{aligned} \tag{4.33}$$

Theorem 4.4.4 (Negative Source-Coding Theorem (Gray [3]))
There exist no block source codes of rate $\overline{R}(\mathcal{C}_{\mathbf{X}}) \leq R$ where $R > 0$, with distortion $\overline{D}(\mathcal{C}_{\mathbf{X}}) < D(R)$. As with Theorem 4.4.2, this result is applicable to all coding methods, including those which are not block-based.

To make use of these theorems in the coding of a stationary, ergodic random process, we can design a constrained-rate random-process code similar to the constrained-distortion code of Section 4.4.1. Again, we partition random process X_n into successive N-dimensional random vectors $\mathbf{X}$ and use the optimal code, $\mathcal{C}^*_{\mathbf{X}}$, whose existence is guaranteed by Theorem 4.4.3, to code each vector. We choose N large enough so that the average distortion satisfies $\overline{D}(\mathcal{C}^*_{\mathbf{X}}) \leq D(R) + \delta$ for the desired rate, R, and accuracy, δ. The resulting average single-letter distortion, $\overline{D}(\mathcal{C}_{X_n})$, defined in Equation 4.24, satisfies

$$D(R) \leq \overline{D}(\mathcal{C}_{X_n}) \leq D(R) + \delta, \tag{4.34}$$

and the rate of the code, $\overline{R}(\mathcal{C}_{X_n})$, defined in Equation 4.21, satisfies

$$\overline{R}(\mathcal{C}_{X_n}) \leq R. \tag{4.35}$$

As before, we can make δ arbitrarily small by increasing the block size N. Consequently we see that the distortion-rate function, $D(R)$, is not only a lower bound on the achievable distortion for any coding method with rate not more than R, but is achievable by a block source code of infinite blocking. Therefore, block source coding is asymptotically optimal in the distortion-rate sense in the same way that it is optimal in the rate-distortion sense, the result shown in the previous section.

4.4.3 Source Coding of Stationary, Nonergodic Random Processes

To this point, we have been concerned with only stationary processes that are also ergodic. We now remove the assumption of ergodicity and review the results that are available for stationary, nonergodic sources.

To establish source-coding theorems for stationary, nonergodic sources, it is common to invoke the ergodic-decomposition property (see, for example, [19, 3]) which states that every stationary, nonergodic process can be considered to be composed of separate stationary, ergodic sources. That is, a nonergodic process can be considered to randomly select a source from a collection of stationary, ergodic processes and then produce output from that process for all time [3]. The ergodic-decomposition property of stationary sources makes it possible to adapt the positive source-coding theorems of the previous sections to nonergodic sources to derive the following results. One should note that the negative source-coding theorems (Theorems 4.4.2 and 4.4.4) can be shown to apply to nonergodic sources directly as previously stated [19].

Gray and Davisson [19] have shown that the optimal fixed-length block code that has the smallest distortion for a given rate does not achieve the distortion-rate function; rather the optimal distortion for a given rate was determined to be a weighted average of the distortion-rate functions of the individual processes of the ergodic decomposition. By *fixed-length* block code, we mean a block source code, $\mathcal{C}_{\mathbf{X}}$, which produces fixed-length codewords and whose average rate, $\overline{R}(\mathcal{C}_{\mathbf{X}})$, is defined as

$$\overline{R}(\mathcal{C}_{\mathbf{X}}) \triangleq \frac{1}{N}\log_2 K. \tag{4.36}$$

Shields *et al.* [20] and Leon-Garcia *et al.* [21] removed the assumption of fixed-length coding, allowing the codewords to have variable length and, consequently, defining the rate of the code as Equation 4.18. It was then determined that the distortion-rate function was indeed achievable by a variable-length code [20, 21].

These results show that block source coding is as equally powerful for stationary, nonergodic sources as it is for stationary, ergodic sources. The removal of the assumption of ergodicity does not diminish the significance of the source-coding theorems, although it does complicate their proofs somewhat. On the other hand, removal of the stationarity assumption does affect the strength of those theorems that can be derived in such a case. In fact, only weakened source-coding theorems have been proven to date for nonstationary sources, and then only under certain additional restrictions, as we shall see in the next section.

4.5 Rate-Distortion Theory for Nonstationary, Continuous Random Processes

Up to this point, the discussion in this chapter has been limited to the consideration of stationary, continuous random processes. We now remove the assumption of stationarity and review the rate-distortion results available for nonstationary sources. Because the assumptions of stationarity and ergodicity simplify the derivation of source-coding theorems, the results available for nonstationary sources are very few and usually concern only limited classes of nonstationary sources.

In general, a complete treatment of rate-distortion theory for a particular source involves the establishment of the existence of the rate-distortion function (defined similarly to Equations 4.7 and 4.8) or, equivalently, the distortion-rate function (defined similarly to Equations 4.11 and 4.12), as well as the derivation of both positive and negative source-coding theorems. The source-coding theorems provide the rate-distortion and distortion-rate functions real significance toward the design of communication systems by showing that the optimal rate-distortion or distortion-rate performance is achievable by a particular coding method. We will see below that, to date, it has been possible to derive only limited source-coding theorems for only a few classes of nonstationary sources.

We proceed to review the results that have been reported for nonstationary random processes. The discussion here will be brief; it is suggested that the interested reader refer to the extensive source-coding summary by Kieffer [22], or to the individual reports themselves, for more details. In Section 4.5.1, we present the various source-coding results that have been derived for a variety of nonstationary processes. Then, in Section 4.5.2, we describe in greater detail the theory that has been developed for the Wiener process, a nonstationary process that will be used extensively to obtain experimental results in the remainder of this text.

4.5.1 Source-Coding Results for Various Nonstationary Processes

Gray and Saadat [23] and Gray [9] have reported source-coding results for asymptotically mean-stationary, ergodic sources. Briefly, an asymptotically mean-stationary random process is a nonstationary random process with a continuous probability density that, when averaged over all time shifts, converges to a stationary density. The stationary process described by this stationary density is known then as the stationary mean process of the asymptotically mean-stationary process. Gray and Saadat [23] have shown that the optimal block source code for an asymptotically mean-stationary process achieves the distortion-rate function of the corresponding stationary mean process.

Gray [16] derived a parametric expression, subsequently corrected by Hashimoto and Arimoto [24], for the rate-distortion function of Gaussian autoregressive processes. Gray [16] also proved a limited version of the positive source-coding theorem, showing the existence of a block source code for a sufficiently large block size N with rate arbitrarily close to the rate-distortion function for a given distortion. Gray's theorem is unfortunately limited in that one cannot achieve optimal coding for the process by using the same code for each successive block of N values as was done for the coding of stationary processes in Section 4.4.1. In this way, Gray's positive source-coding theorem is strictly "one-shot." Berger [18], however, was able to prove a stronger positive source-coding theorem that overcomes this limitation by considering the Wiener process, a special case of the Gaussian autoregressive process. As we will use Berger's results extensively in the remainder of this text, we will present them in greater detail in Section 4.5.2.

However, before turning to Berger's results for the Wiener process, it is worth mentioning two other reports pertaining to the coding of nonstationary processes. Ziv [25] has obtained an expression for the optimal distortion achievable for a given rate for the coding of an individual sequence, and Kieffer [26] has extended this formula to more general nonstationary sources. Whereas the results mentioned previously in this section are generally applicable to continuous as well as discrete processes, Ziv and Kieffer's results are limited to discrete processes.

4.5.2 Source-Coding Results for the Wiener Process

In this section, we present the rate-distortion results obtained by Berger [18] for the Wiener process, a special case of the Gaussian autoregressive process. The Wiener process is defined as

$$X_n = \sum_{i=1}^{n} Z_i, \qquad n = 1, 2, \dots , \tag{4.37}$$

where Z_n is an iid random process whose random variables, Z_i, are independent, zero-mean Gaussian random variables with variance $\sigma^2 > 0$ [27]. Note that we may equivalently express the Wiener process as a recursion,

$$X_n = X_{n-1} + Z_n, \qquad n = 1, 2, \dots , \tag{4.38}$$

where we set $X_0 = 0$ as an arbitrary starting point of the recursion.

Berger [18] has obtained a parametric expression for the rate-distortion function of the Wiener process. Suppose we define the rate-distortion function as was done in Section 4.2. Then, according to Berger,

$$R(D) = \frac{1}{2}\log_2\left(\frac{\sigma^2}{D}\right), \qquad 0 \le D \le \frac{\sigma^2}{4}. \tag{4.39}$$

For $D > \frac{\sigma^2}{4}$, the rate-distortion function is given parametrically by

$$D(\theta) = \left(\frac{1}{\pi}\right)\left[2\theta \arcsin\left(\frac{\sigma}{2\sqrt{\theta}}\right) + \left(\frac{\sigma}{2}\right)\sqrt{4\theta - \sigma^2}\right], \tag{4.40}$$

$$R(\theta) = \left(\frac{1}{\pi}\right)\left[C_2\left(2\arcsin\left(\frac{\sigma}{2\sqrt{\theta}}\right)\right) + \arcsin\left(\frac{\sigma}{2\sqrt{\theta}}\right)\log_2\left(\frac{\sigma^2}{\theta}\right)\right], \tag{4.41}$$

where θ is the variable of parameterization, and $C_2(y)$ is Clausen's second integral,

$$C_2(y) = -\int_0^y \log_2\left[2\sin\left(\frac{t}{2}\right)\right] dt. \tag{4.42}$$

Figure 4.1 plots $R(D)$ as given by Equations 4.39 through 4.41 for a Wiener process with several variances. To create Figure 4.1 for $D > \frac{\sigma^2}{4}$, we trace out the $R(D)$ curve by calculating both $D(\theta)$ and $R(\theta)$ for a number of θ values chosen from a linear sequence. Additionally, in the calculation of each R value, we evaluate $C_2(y)$ numerically.

To show that the rate-distortion function is achievable by an optimal block code, Berger [18] starts with the usual positive source-coding theorem, Theorem 4.4.1, and makes the following observations. First, to code a block of N values of the process, it is sufficient to code the process Z_n along with the value of X_n at the start of the block.

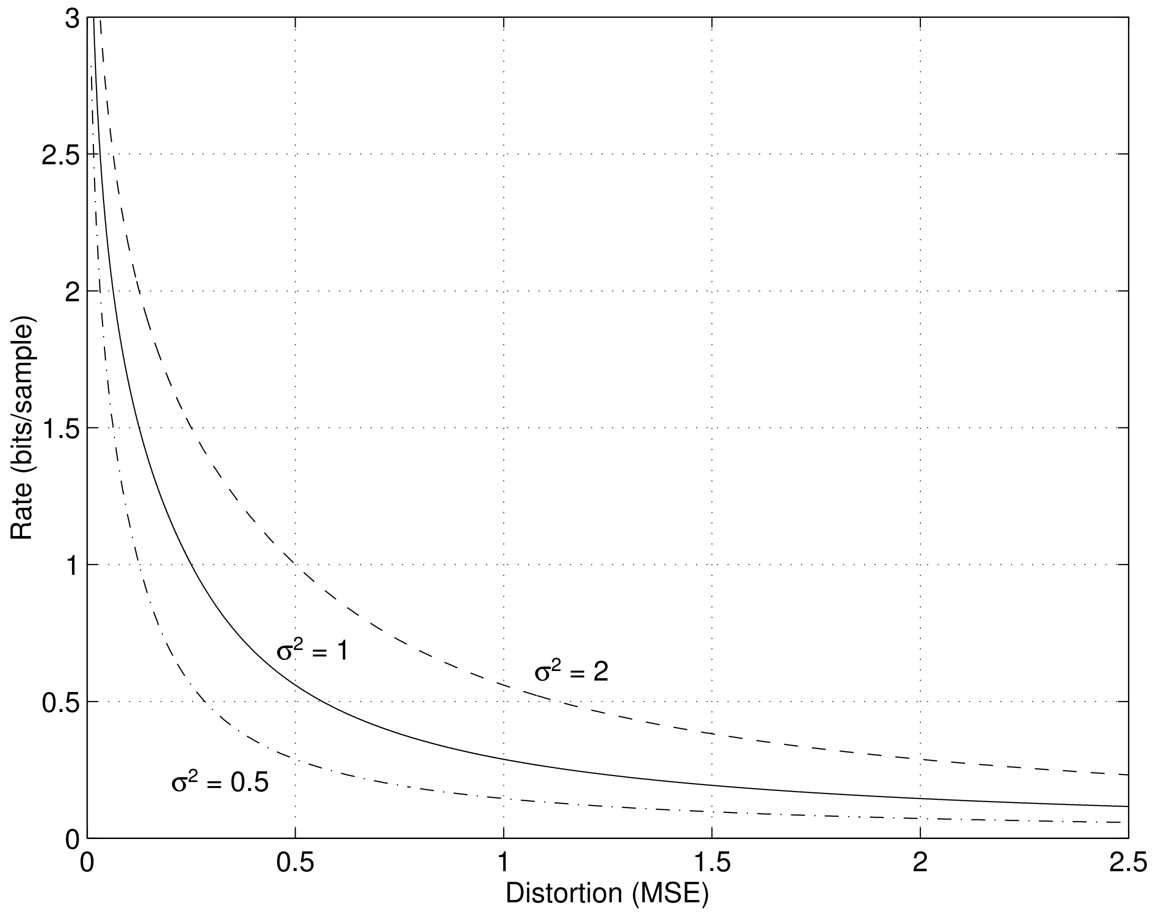


Figure 4.1: The rate-distortion function, $R(D)$, for the Wiener process with variance σ^2 equal to $\frac{1}{2}$, 1, and 2.

However, since the process Z_n is iid, the optimal code, $\mathcal{C}^*_{\mathbf{Z}}$, for the first N values of Z_n will be optimal for the remaining blocks. All that is left is to code the sequence of block starting values so that the total rate does not exceed $R(D) + \epsilon$. Berger [18] has shown that a delta-modulation scheme can be devised to track the block starting values under this rate constraint, thus establishing the positive source-coding theorem valid for entire length of process X_n. This result is more powerful than the positive source-coding theorem developed by Gray [16] for the Gaussian autoregressive process as it does not require a different optimal code for each block of N values of the process. However, the negative source-coding theorem can be established under a proof similar to the one for the corresponding theorem (Theorem 4.4.2) for stationary sources [16], and is thus valid for both the Wiener process and the more general Gaussian autoregressive process.

These observations conclude our initial discussion of the coding of continuous random processes. In this chapter, we established the source-coding capabilities of block source codes for various sources. In the next chapter, we explore vector quantization, a source-coding technique related to block source coding that has been used extensively in practice for the coding of continuous random processes.

CHAPTER 5

VECTOR QUANTIZATION

In the previous chapter, we discussed the coding of continuous random processes. Key to this discussion was the concept of a block source code mapping a vector of values from a random process to a codeword chosen from a predetermined codebook. In the case of stationary processes, it was shown that the block source code was asymptotically optimal, achieving performance on the rate-distortion curve when the vector length approached infinity. However, the theorems of Chapter 4 showed merely the existence of optimal block source codes; no methods were presented for deriving an optimal code for a given source. Additionally, any practical implementation of a block-coding system will be constrained to use finite-length vectors and consequently will not be capable, in general, of performance on the rate-distortion curve. In the design of real communication systems, we need practical block-code design algorithms using finite-length block sizes. We will be interested in evaluating the performance these codes in terms of both rate and distortion. To address these concerns, we appeal now to a coding technique known as vector quantization, which, as we will see later, includes block source coding as a special case.

Vector quantization (VQ) is the generalization of scalar quantization to higher dimensions [2]. Scalar-quantization techniques, which operate on one-dimensional signals, grew out of efforts in the design of analog-to-digital converters, those electrical devices which produce digital representations of analog signals. Many of the ideas behind scalar quantization are equally applicable in higher dimensions; however, VQ, being more general, offers much greater source-coding potential than scalar quantization.

In this chapter, we first present the basic mathematics that define the concept of a vector quantizer in terms of a function of a random vector. Then, in Section 5.2, we discuss optimal VQ, defining what it means for a vector quantizer to be optimal and presenting conditions necessary for such optimality. In Section 5.3, we present a model of a communication system using VQ, define the concept of local optimality of such a system, and then describe several practical VQ-design algorithms that, although not guaranteed to produce optimal quantizers, do produce vector quantizers with some form of local optimality. Finally, in Section 5.4, we extend these ideas to communication systems that code random processes by blocking the process into vectors and using a predesigned vector quantizer to code each vector. By predesigned, we mean that the vector quantizer is completely designed beforehand and is not modified during the coding of the random process. We will call this type of vector quantizer *nonadaptive VQ*. The more general case of *adaptive VQ* (AVQ), our primary topic, will be investigated later starting in Chapter 6.

5.1 Preliminary Definitions

In this section, we present the mathematical preliminaries behind the theory of VQ. Most of the content of this section, as well as that of the rest of this chapter, is based on well known theory that is discussed extensively elsewhere; the reader is referred to, in particular, the thorough book [2] by Gersho and Gray.

We define a *vector quantizer* [2], Q, as a mapping from N-dimensional Euclidean space, $\Re^N$, to a finite subset, $\mathcal{C}$, of $\Re^N$,

$$Q : \Re^N \to \mathcal{C}, \qquad \mathcal{C} \subset \Re^N. \tag{5.1}$$

Subset $\mathcal{C}$, called the *codebook*, is composed of K codewords, $\mathbf{c}_i$, in $\Re^N$,

$$\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\}, \qquad \mathbf{c}_i \in \Re^N. \tag{5.2}$$

Given an N-dimensional vector $\mathbf{x} \in \Re^N$, the output of the vector quantizer is another N-dimensional vector, $\hat{\mathbf{x}}$,

$$\hat{\mathbf{x}} = Q(\mathbf{x}), \qquad \hat{\mathbf{x}} \in \mathcal{C}. \tag{5.3}$$

Thus, the vector quantizer divides the space $\Re^N$ into separate regions with each region being assigned a codeword from $\mathcal{C}$ to represent it. Mathematically, the vector quantizer divides the space $\Re^N$ into K *partition regions*, $\mathcal{R}_i$, defined as

$$\mathcal{R}_i \triangleq \left\{\mathbf{x} \in \Re^N : Q(\mathbf{x}) = \mathbf{c}_i\right\}, \qquad 1 \leq i \leq K. \tag{5.4}$$

From this definition, it follows that

$$\bigcup_i \mathcal{R}_i = \Re^N \tag{5.5}$$

and

$$\mathcal{R}_i \cap \mathcal{R}_j = \varnothing, \qquad i \neq j, \tag{5.6}$$

so the set of $\mathcal{R}_i$'s form a partition of the space $\Re^N$ [2]. Additionally, if the vector quantizer assigns vector $\mathbf{x}$ to region $\mathcal{R}_i$, $\mathbf{x}$ is reproduced as codeword $\mathbf{c}_i$. Consequently, $\mathbf{c}_i$ is sometimes called the *reproduction vector* for region $\mathcal{R}_i$.

Associated with the vector quantizer is a vector distortion measure, $d(\mathbf{x}, \hat{\mathbf{x}})$, which measures the difference between two vectors. Distortions measures were thoroughly discussed previously in Section 4.1; common distortion measures used in VQ systems are the squared error and the mean squared error.

The definitions given thus far define vector quantizers in the most general sense. Restricting these definitions somewhat produces classes of vector quantizers that will be of interest later. We define a *regular vector quantizer* [2] as a vector quantizer whose partition regions, $\mathcal{R}_i$, are each convex sets, and whose corresponding reproduction vectors, $\mathbf{c}_i$, are found within these sets. That is, if $\mathbf{a} \in \mathcal{R}_i$ and $\mathbf{b} \in \mathcal{R}_i$, then

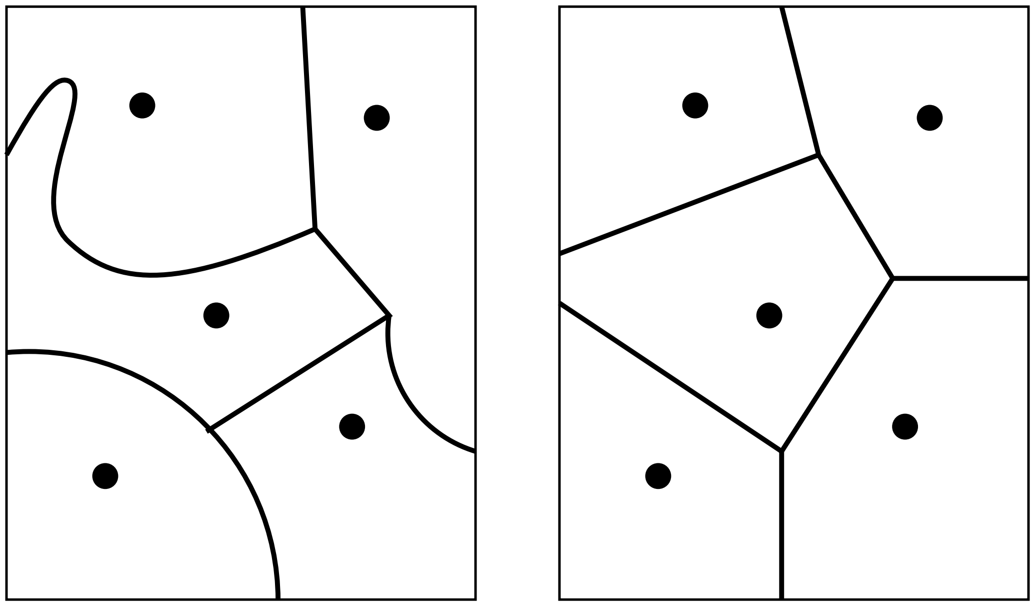
$$(1-\lambda)\mathbf{a} + \lambda\mathbf{b} \in \mathcal{R}_i, \qquad 0 < \lambda < 1, \tag{5.7}$$

and

$$\mathbf{c}_i \in \mathcal{R}_i, \tag{5.8}$$

for a regular vector quantizer for all i. Additionally, we define a *polytopal vector quantizer* [2] as a regular vector quantizer whose partition regions are bounded by segments of hyperplanes surfaces in $\Re^N$. Figure 5.1 illustrates examples of a nonregular and a polytopal vector quantizer in two dimensions.

One might be lead to believe that, since polytopal vector quantizers are not as general as nonregular vector quantizers, polytopal vector quantizers suffer in performance when used to quantize a random vector. However, this is not always the case. As we shall see in the next section, the class of polytopal vector quantizers perform at least as well as the class of nonregular vector quantizers when the squared error is used as the distortion measure. Indeed, necessary conditions for the optimality of



(a) Nonregular (b) Polytopal

Figure 5.1: Examples of nonregular and polytopal vector quantizers in two dimensions

a vector quantizer under the squared-error distortion measure result in a polytopal vector quantizer. We proceed now to the next section to further investigate these optimality conditions.

5.2 Optimal Vector Quantization of a Random Vector

In this section, we discuss the optimal VQ of a random vector. We will assume throughout this section, as well as through the rest of this chapter, that the VQ distortion measure is the squared error, unless otherwise specified. We start by assuming that we have an N-dimensional continuous random vector, $\mathbf{X}$, that is vector quantized to produce another random vector, $\hat{\mathbf{X}}$,

$$\hat{\mathbf{X}} = Q(\mathbf{X}). \tag{5.9}$$

Note that, since the codebook, $\mathcal{C}$, associated with the vector quantizer is a finite set, $\hat{\mathbf{X}}$ is a discrete random vector.

We define the *rate*, $\overline{R}(Q)$, of the vector quantizer as

$$\overline{R}(Q) \triangleq \frac{1}{N} I(\mathbf{X};\hat{\mathbf{X}}). \tag{5.10}$$

Note that the rate, as defined by Equation 5.10, depends on the pdf of the input random vector, $\mathbf{X}$. However, we can derive a pdf-independent bound on the rate. Since the vector quantizer is a deterministic function, we know that $H(\hat{\mathbf{X}} \,|\, \mathbf{X}) = 0$, so we have

$$\begin{aligned} \overline{R}(Q) &= \frac{1}{N} I(\mathbf{X};\hat{\mathbf{X}}) \\ &= \frac{1}{N} H(\hat{\mathbf{X}}) - \frac{1}{N} H(\hat{\mathbf{X}} \,|\, \mathbf{X}) \\ &= \frac{1}{N} H(\hat{\mathbf{X}}). \end{aligned} \tag{5.11}$$

Using the bound from Equation 3.2, we then have

$$\overline{R}(Q) = \frac{1}{N} H(\hat{\mathbf{X}}) \leq \frac{1}{N} \log_2 K, \tag{5.12}$$

where K is the size of the codebook.

We can quantify the performance of the vector quantizer given input random vector $\mathbf{X}$ by examining the *average distortion* [2], $\overline{D}(Q)$, of the quantizer, defined as

$$\overline{D}(Q) \triangleq E\left[d(\mathbf{X},\hat{\mathbf{X}})\right] = \int f_{\mathbf{X}}(\mathbf{x}) d(\mathbf{x}, Q(\mathbf{x}))\, d\mathbf{x}, \tag{5.13}$$

where $f_{\mathbf{X}}(\mathbf{x})$ is the pdf of random vector $\mathbf{X}$. An *optimal* vector quantizer for a given input random vector, $\mathbf{X}$, and a fixed codebook size, K, is one that produces the smallest average distortion [2]. There are two well known conditions that are necessarily satisfied by an optimal vector quantizer: the *nearest-neighbor* and the *centroid* conditions [2].

The nearest-neighbor condition states that the best encoding of a given vector is produced by the codeword that is closest to it, where "closest" is in terms of the distortion measure [2]. Mathematically, the nearest-neighbor condition implies that an optimal vector quantizer must satisfy

$$\hat{\mathbf{x}} = Q(\mathbf{x}) = \mathbf{c}_i \quad \text{only if} \quad d(\mathbf{x}, \mathbf{c}_i) \leq d(\mathbf{x}, \mathbf{c}_j), \quad \forall j. \tag{5.14}$$

Thus, given a codebook, the nearest-neighbor condition determines the partition regions of an optimal quantizer, which satisfy

$$\mathcal{R}_i \subset \left\{\mathbf{x} \in \Re^N : d(\mathbf{x}, \mathbf{c}_i) \leq d(\mathbf{x}, \mathbf{c}_j), \, \forall j \right\}, \tag{5.15}$$

where the subset is used in this expression so that a point that is equidistant to two or more codewords may be assigned arbitrarily to any of the corresponding regions without affecting the average distortion. Nearest-neighbor vector quantizers are commonly used in practice. Not only do these vector quantizers satisfy one of the necessary optimality conditions, but they also result in simple encoding algorithms that do not require storing the geometries of all the partition regions of the quantizer [2].

The centroid condition states that, given a set of partition regions, the codeword associated with each region should be assigned so as to minimize the distortion between that codeword and all the vectors assigned to the region [2]. Mathematically, this generalized centroid condition implies

$$\mathbf{c}_i = \arg\min_{\mathbf{y}} E\left[d(\mathbf{X}, \mathbf{y}) \mid \mathbf{X} \in \mathcal{R}_i\right]. \tag{5.16}$$

Since we are assuming that the distortion measure is the squared error, Equation 5.16 can be shown to be

$$\mathbf{c}_i = E\left[\mathbf{X} \mid \mathbf{X} \in \mathcal{R}_i\right], \tag{5.17}$$

which corresponds to the familiar definition of the centroid used in mechanics [2].

It can be shown that the nearest-neighbor condition results in an optimal vector quantizer having convex, polytopal partition regions [2]. Additionally, the centroid condition guarantees that the reproduction vectors lie within their respective partition regions [2]. These observations result in the fact that an optimal vector quantizer is not only regular, but also polytopal [2]. Consequently, the class of polytopal vector quantizers performs at least as well as the class of more general, nonregular vector quantizers, a result that was alluded to in the previous section.

Additionally, one should observe that a block source code, as defined in Section 4.4.1, is a vector quantizer satisfying the nearest-neighbor condition. Consequently, a block source code is a special case of a vector quantizer, namely a nearest-neighbor vector quantizer. Thus, the positive and negative source-coding theorems of Sections 4.4.1 and 4.4.2 apply to an optimal vector quantizer used to code a stationary random process; this topic will be investigated further in Section 5.4.

Now that we have discussed conditions for optimality of a vector quantizer, we turn to the performance, in terms of rate and distortion, that is achievable by an optimal vector quantizer meeting these conditions. We have defined the average distortion of a vector quantizer in Equation 5.13. It would be of interest to express the minimum average distortion, $\overline{D}(Q^*)$, achievable for an optimal vector quantizer, Q^*, in terms of the input probability density, $f_{\mathbf{X}}(\mathbf{x})$, assuming a fixed number of codewords, K, and a fixed vector dimension, N. No expression of this kind is known in this general case;

however, if we make the assumption of a high-resolution quantizer, i.e., a quantizer with K very large, then approximate bounding expressions on the minimum average distortion can be derived. For example, it has been shown that a lower bound is

$$\overline{D}(Q^*) \geq \frac{N}{N+2} \left(\frac{2\pi^{\frac{N}{2}}}{N\Gamma(\frac{N}{2})} \right)^{-\frac{2}{N}} \left[\int (f_{\mathbf{X}}(\mathbf{x}))^{\frac{N}{N+2}} \, d\mathbf{x} \right]^{\frac{N+2}{N}}, \tag{5.18}$$

where $\Gamma(\cdot)$ is the gamma function [2, 28, 29].

The lack of a suitable expression of the minimum distortion in the general case coupled with the usually questionable applicability of the high-resolution assumption in practice makes it difficult to know whether a given vector quantizer is optimal. This difficulty will arise again when we consider the design of VQ codebooks in Section 5.3.

We have defined the rate of the vector quantizer in Equation 5.10. As the optimality conditions place no constraints on the rate, an optimal vector quantizer will have, in general, a rate less than or equal to $\frac{1}{N}\log_2 K$, this upper bound being dictated by Equation 5.12. We shall see later in Section 5.3 that practical design methods exist for constructing vector quantizers that have a rate guaranteed to be less than this upper bound. These vector quantizers will usually be suboptimal in the sense of minimum distortion as discussed in this section since a vector quantizer that is optimized without constraints on its rate will usually achieve a smaller average distortion [2].

Finally, one should note that the nearest-neighbor and centroid conditions are only necessary conditions, not sufficient conditions. An optimal vector quantizer will satisfy these conditions; however, in designing a vector quantizer, the fact that these conditions are satisfied does not guarantee that an optimal quantizer has been found. Regardless, these necessary conditions do provide a basis for VQ-design methods that are commonly used in practice. These design methods are the topic of the next section.

5.3 Vector-Quantizer System Design

Thus far, we have presented the mathematical basics to VQ. In this section, we will see how we can use VQ in real communication systems as well as how we can design the vector coders for these quantizers. We will first present a communication-system model using VQ and then, in Sections 5.3.1 through 5.3.2, we will describe several popular methods of VQ design.

Figure 5.2 shows the incorporation of VQ into the communication-system model presented previously in Figure 2.2 of Section 2.3. For the moment, we will assume that the source, $\mathbf{X}$, is a random vector; we will consider the case of a random-process source later in Section 5.4.

As seen in Figure 5.2, we divide the VQ system implementing vector quantizer Q into two major components, the *encoder* and the *decoder*. Each of these major components is again divided into two simpler parts. The encoder consists of a *vector coder* and a *index coder*. The vector coder is a mapping, $\alpha(\cdot)$, that outputs a random variable I, called the *index*, given input random vector $\mathbf{X}$; i.e.,

$$I = \alpha(\mathbf{X}). \tag{5.19}$$

We define $\alpha(\cdot)$ so that

$$I = \alpha(\mathbf{X}) \triangleq i \quad \text{when} \quad Q(\mathbf{X}) = \mathbf{c}_i. \tag{5.20}$$

The vector coder is usually chosen to be a nearest-neighbor mapping based on the distortion between X and $\mathbf{c}_i$ so that the VQ system implements a nearest-neighbor vector quantizer. In this case, simply specifying the codebook $\mathcal{C}$ suffices to define $\alpha(\cdot)$. The second component of the encoder, the index coder, is a one-to-one function, $\gamma(\cdot)$,

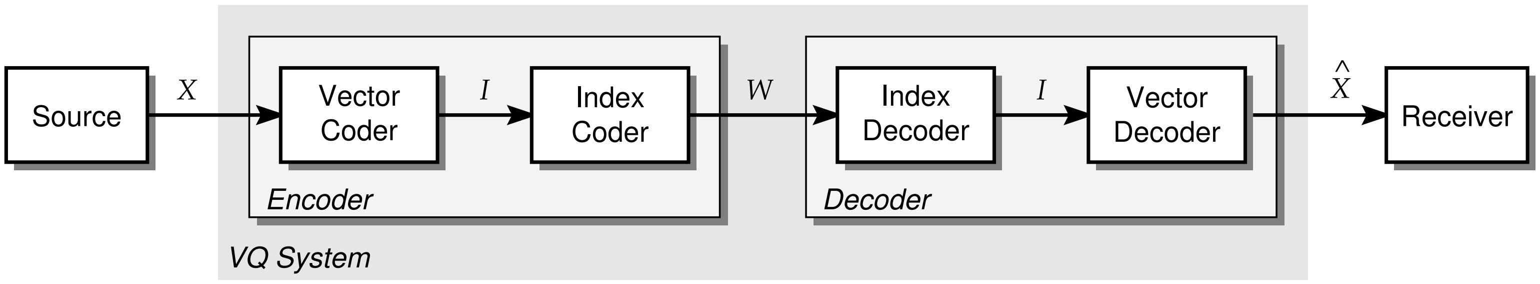


Figure 5.2: Communication system incorporating VQ (adapted from [30]).

that maps each index to a set of K codewords, w_i; i.e.,

$$\gamma : \{1, 2, \ldots, K\} \to \{w_1, w_2, \ldots, w_K\}, \tag{5.21}$$

where each w_i is a string of binary digits.

On the decoder side of the system, the *index decoder* is simply the inverse function, $\gamma^{-1}(\cdot)$, of the index coder. The *vector decoder* outputs random vector $\hat{\mathbf{X}}$ from input random variable I. The vector decoder is consequently a one-to-one function, $\beta(\cdot)$, defined as

$$\beta(I) \triangleq \mathbf{c}_i \quad \text{when} \quad I = i, \tag{5.22}$$

where $\mathbf{c}_i$ are the codewords in the VQ codebook, $\mathcal{C}$. We see that the concatenation of the components of the encoder and the decoder implements vector quantizer Q,

$$\hat{\mathbf{X}} = \mathbf{c}_i = \beta(\gamma^{-1}(\gamma(\alpha(\mathbf{X})))) = Q(\mathbf{X}). \tag{5.23}$$

The design of a VQ system consists of two phases. In the first phase, a vector coder, $\alpha(\cdot)$, is designed, which determines the codebook, $\mathcal{C}$, and the vector decoder, $\beta(\cdot)$. Afterwards, the index coder and index decoder are designed. The performance of the VQ system is then measured in terms of the average distortion,

$$\overline{D}(Q) \triangleq E\left[d(\mathbf{X}, \hat{\mathbf{X}})\right], \tag{5.24}$$

and the average code length,

$$\overline{L}(Q) \triangleq \frac{1}{N} E\left[l(\gamma(\alpha(\mathbf{X})))\right], \tag{5.25}$$

where $l(w_i)$ is the length in bits of the codeword w_i and N is the vector dimension. In general, both the average distortion and the average code length will depend on the pdf of the random-vector source, $\mathbf{X}$.

Let us focus for the moment on the second phase of the VQ-system-design procedure, assuming we are given a codebook, $\mathcal{C}$, for random vector $\mathbf{X}$. There are two straightforward ways of designing the index coder, $\gamma(\cdot)$. The first, and most simple, is to assign to each index the fixed-length code corresponding to its binary representation. That is, each w_i is simply the binary representation of index i. Consequently, each codeword has the same length, so the average code length is

$$\overline{L}(Q) = \frac{\lceil \log_2 K \rceil}{N}, \tag{5.26}$$

where K is the size of the codebook. This type of index coder is called a *fixed-length* coder. Note that the average code length for the VQ system using a fixed-length coder does not depend on the pdf of the source. The second type of index coder is a *variable-length* coder which implements any of the entropy-coding methods discussed in Section 3.3. Use of a variable-length coder will generally result in the VQ system having a smaller average code length than that of a system using a fixed-length code. In fact, from Theorem 3.3.1, the average code length for a variable-length system satisfies

$$\frac{1}{N}H(\hat{\mathbf{X}}) \le \overline{L}(Q) \le \frac{1}{N}H(\hat{\mathbf{X}}) + \frac{1}{N} \tag{5.27}$$

when the variable-length coder implements an optimal prefix code. However, from the definition of the rate of the VQ system in Equation 5.11, we can rewrite Equation 5.27 as

$$\overline{R}(Q) \le \overline{L}(Q) \le \overline{R}(Q) + \frac{1}{N}. \tag{5.28}$$

Consequently, the average code length of the variable-rate VQ system is close to the rate of the VQ system, particularly so when the vector dimension N is large. For this reason, it is common to think of the average code length as being an *operational rate* of the variable-length VQ system. The operational rate of the VQ system is guaranteed to be close to the true rate of the system when the vector dimension is large.

The design of the index coder of a VQ system relies on a predefined vector coder, the design of this vector coder being the first phase in the overall design of the VQ system. In general, the design of an optimal vector coder is a very complex problem since the known optimality conditions are only necessary, not sufficient, conditions. In fact, the VQ-design problem can be shown to be at least NP-complete [31], meaning that there exists no algorithm that is sure to produce an optimal vector coder in polynomial time. Consequently, the algorithms commonly used for VQ design are guaranteed to produce vector coders that are only locally optimal—"small" changes in a locally optimal vector coder will not result in a reduction of the average distortion. However, these locally optimal vector coders are not, in general, globally optimal. Furthermore, since the average distortion of a globally optimal quantizer is not known in general (see Section 5.2), we have no way of evaluating how close in distortion a local optimum is to the global optimum. We can only hope that the local optimum is a "good" optimum that corresponds to an average distortion that is close minimal.

Below, we survey several algorithms that are locally optimal. Although there exist versions of some of the algorithms that are designed for known source pdf's, we present the version of each algorithm that assumes that the pdf of the source is unknown, as it is this case that will be of interest later. These algorithms use a set of sample vectors generated from the source as a "training set." Each of the algorithms starts with an initial vector coder that is subsequently modified during several iterations through the training set. For each algorithm, a cost function is defined. This cost function is simply the average distortion for most algorithms, although, in the case of the entropy-constrained-VQ (ECVQ) algorithm, the cost function considers the rate as well. For each iteration of each algorithm, a new vector coder with a lower cost over the training set is produced. Each algorithm converges to a vector coder producing a locally minimal cost given the training set. The initial codebook used by each design algorithm is usually chosen so that the initial codewords are reasonably scattered over the space expected to be covered by the source pdf [2]. Common approaches for generating the initial codebook are using random values for the codebook components or randomly choosing the codewords from the training set.

The algorithms we survey below are the generalized Lloyd algorithm, Kohonen learning, and ECVQ. Although there exist other techniques (see [2]), we focus on these three algorithms because adaptive-VQ (AVQ) algorithms have been based on these three nonadaptive-VQ-design methods. We will discuss AVQ versions of these algorithms later in Chapter 7.

5.3.1 The Generalized Lloyd Algorithm

The most popular VQ-design algorithm is the generalized Lloyd algorithm which is a generalization to higher dimensions of Lloyd's Method I algorithm [32] for scalar quantizers. The generalized Lloyd algorithm is also commonly known as the LBG algorithm due to the paper [33] by Linde *et al.* which popularized the algorithm in VQ-research circles. The generalized Lloyd algorithm consists of alternatively applying the nearest-neighbor and centroid conditions to generate a sequence of codebooks with decreasing average distortion. Because the generalized Lloyd algorithm yields a nearest-neighbor vector quantizer, the final codebook produced by the algorithm completely defines the vector coder $\alpha(\cdot)$ and vector decoder $\beta(\cdot)$.

The algorithm operates as follows. We begin with an initial codebook of size K. We partition the training set into K sets of vectors using the nearest-neighbor rule to associate each training vector with one of the K codewords of the codebook. This partitioning defines the vector-coder mapping, $\alpha(\cdot)$. We then create a new codebook with the new codewords being the centroids of each of the K partition regions of training vectors. This new codebook defines the vector-decoder mapping $\beta(\cdot)$. The algorithm repeats these steps and calculates the average distortion for the new codebook over the entire training set at the end of each iteration. When the average distortion fails to decrease by a certain threshold amount, it is assumed that a local minimum in distortion has been reached and the algorithm terminates. Figure 5.3 gives a detailed description of the flow of the algorithm.

The generalized Lloyd algorithm is a descent algorithm, that is, an algorithm that "descends" down the surface of a complex cost function in a high-dimensional space [2]. In the case of the Lloyd algorithm, the cost function is the average distortion and the descent is accomplished by making small perturbations in the codebook. Since the average distortion is a complex function possessing, in general, many local minima, the generalized Lloyd algorithm will descend into a local minimum near the starting codebook. Consequently, the choice of the initial codebook may have a substantial effect on the performance of the final codebook produced by the algorithm. Attempts to rid the generalized Lloyd algorithm of its dependence on the initial codebook have resulted in several modified algorithms. Examples include stochastic-relaxation

Given: initial codebook, $\mathcal{C}_0$, for initial vector quantizer, Q_0
training set, $\mathcal{T}$
average distortion of initial quantizer, $D(Q_0)$
stopping threshold, ϵ
iteration counter, $m = 1$

Step 1: For each $\mathbf{x} \in \mathcal{T}$, define the vector coder as

$$\alpha(\mathbf{x}) = \arg\min_i \left[d(\mathbf{x}, \mathbf{c}_i)\right]$$

where $\mathbf{c}_i \in \mathcal{C}_m$ during iteration m. This produces K partition sets:

$$\mathcal{R}_i = \{\mathbf{x} : \mathbf{x} \in \mathcal{T},\, \alpha(\mathbf{x}) = i\}\,.$$

Step 2: Form the new codebook, $\mathcal{C}_{m+1}$, and consequently the vector decoder, $\beta(\cdot)$, from the centroids of each partition region:

$$\mathcal{C}_{m+1} = \left\{\mathbf{c}_i : \mathbf{c}_i = \beta(i) = \frac{1}{|\mathcal{R}_i|} \sum_{\mathbf{x} \in \mathcal{R}_i} \mathbf{x}\right\}.$$

Step 3: Estimate the average distortion for the new codebook, $D(Q_{m+1})$:

$$D(Q_{m+1}) = \frac{1}{|\mathcal{T}|} \sum_{\mathbf{x} \in \mathcal{T}} d(\mathbf{x}, \beta(\alpha(\mathbf{x}))).$$

Step 4: If

$$\frac{D(Q_m) - D(Q_{m+1})}{D(Q_m)} < \epsilon$$

then stop. Otherwise, set $m = m + 1$ and go to Step 1.

Figure 5.3: The generalized Lloyd algorithm for VQ design [33]. The distortion measure is assumed to be the squared error.

techniques which introduce randomness into the generalized Lloyd algorithm to enable it to escape local minima. These stochastic-relaxation VQ-design methods are surveyed by Gersho and Gray [2].

The generalized Lloyd algorithm evolved from the iterative application of the two necessary conditions for VQ optimality introduced in Section 5.2. In the next section, we present a family of algorithms that had origins in the theory of artificial neural networks. We shall see that these algorithms are also related to the two necessary optimality conditions.

5.3.2 Kohonen Learning

An important family of VQ-design algorithms is based on an artificial-neural-network algorithm called Kohonen learning [34]. Although originally developed to describe the formation of topological feature maps in the brain, the Kohonen-learning algorithm and its variants have been used extensively for pattern classification and clustering [35]. Figure 5.4 describes in detail the flow of the Kohonen-learning algorithm. Like the generalized Lloyd algorithm, Kohonen learning produces nearest-neighbor vector quantizers.

Featured in the Kohonen-learning algorithm are the *learning rate* and the *neighborhood function*. The learning rate, $\phi_i(n)$, is a function that decays monotonically from 1 to 0 as learning progresses. The learning rate ensures that the codewords are initially very mobile but finally converge to static locations. The neighborhood function, $\eta_i(j, n)$, is a function of the index j of the nearest-neighbor codeword of the current training vector. Usually the neighborhood function is defined to be nonzero only for codewords close to the nearest-neighbor codeword. The size of this neighborhood gradually reduces as training progresses.

Given: initial codebook, $\mathcal{C}_0$, for initial vector quantizer, Q_0
training set, $\mathcal{T}$
average distortion of initial quantizer, $D(Q_0)$
stopping threshold, ϵ
learning rate, $\phi_i(n)$, for each codeword
neighborhood function, $\eta_i(j,n)$, for each codeword
codeword counter, $n = 0$
iteration counter, $m = 1$

Step 1: For each $\mathbf{x} \in \mathcal{T}$, do

Step 1a: Determine the index of the closest codeword:

$$j = \arg\min_i \left[d(\mathbf{x}, \mathbf{c}_i) \right],$$

Step 1b: Update each codeword according to the learning rule:

$$\mathbf{c}_i = \mathbf{c}_i + \phi_i(n)\eta_i(j,n) \left[\mathbf{x} - \mathbf{c}_i \right].$$

Step 1c: Reduce the learning rate and neighborhood function and set $n = n + 1$.

Step 2: Form the new codebook $\mathcal{C}_{m+1}$ from the updated codewords $\mathbf{c}_i$ which determines the vector coder, $\alpha(\cdot)$, and the vector decoder, $\beta(\cdot)$.

Step 3: Estimate the average distortion of the new codebook:

$$D(Q_{m+1}) = \frac{1}{|\mathcal{T}|} \sum_{\mathbf{x} \in \mathcal{T}} d(\mathbf{x}, \beta(\alpha(\mathbf{x}))).$$

Step 4: If

$$\frac{D(Q_m) - D(Q_{m+1})}{D(Q_m)} < \epsilon$$

then stop. Otherwise, set $m = m + 1$ and go to Step 1.

Figure 5.4: The Kohonen-learning algorithm for VQ design [34]

There are several variants of the Kohonen-learning algorithm in widespread use. Perhaps the most popular is *competitive learning* [36, 34]. In the competitive-learning algorithm, the neighborhood function is defined to be 1 for the nearest-neighbor codeword and 0 for all others. It has been shown that the competitive-learning algorithm is derivable from the two necessary conditions upon which the generalized Lloyd algorithm is based [37]. Consequently, it is not surprising that it has been observed that competitive learning and its variants have performance comparable to that of the generalized Lloyd algorithm [37, 38, 39]. Competitive learning is less computationally expensive than Kohonen learning as it lacks the extensive calculations that must be performed for neighboring codewords. However, the Kohonen-learning algorithm is able to avoid "bad" local minima due to its neighborhood function. Consequently, there have been several attempts to modify competitive learning to reduce the likelihood of convergence to bad local minima. These efforts have resulted in a family of competitive-learning algorithms featuring so-called *conscience mechanisms* [40, 41] that reduce the chances that the algorithm excessively modifies one codeword to the neglect of the others. Frequency-sensitive competitive learning [35] and competitive selective learning [37] are examples of such an approach.

Both the generalized Lloyd algorithm and Kohonen learning use the two necessary conditions introduced in Section 5.2 as a basis for VQ-codebook generation. As these conditions concern the minimization of the average distortion regardless of rate, the rate of the quantizer generated by these algorithms may be as high as the maximum value $\frac{1}{N}\log_2 K$. In the next section, we present an algorithm that is designed to constrain the rate to be below this maximum.

5.3.3 Entropy-Constrained Vector Quantization

Entropy-constrained VQ (ECVQ) was developed by Chou *et al.* [30] as a modification to the generalized Lloyd algorithm. The ECVQ algorithm produces a vector coder that minimizes the average distortion subject to a constraint on the entropy of the quantizer output, $H(\hat{\mathbf{X}})$. Whereas the generalized Lloyd algorithm left the design of the index coder, $\gamma(\cdot)$, to be completed separately after the codebook was generated, the ECVQ algorithm works under the assumption that the index coder will be variable length. The estimated lengths of the codewords of this variable-length coder are used in a cost function that weighs the average distortion against the rate. The minimization of this cost function results in the ECVQ algorithm being similar to the generalized Lloyd algorithm. However, instead of the two-step iteration of the generalized Lloyd algorithm, each iteration of the ECVQ algorithm consists of three steps: the application of a modified nearest-neighbor rule, the updating of the estimated lengths of the variable-length codewords, and finally the updating of the codebook with the centroid rule.

More precisely, the ECVQ algorithm is based on a cost function of the form

$$J(Q) = \overline{D}(Q) + \lambda \overline{L}(Q), \tag{5.29}$$

where λ is a predefined parameter and $\overline{D}(Q)$ and $\overline{L}(Q)$ are defined in Equations 5.24 and 5.25, respectively. The ECVQ algorithm consists of minimizing $J(Q)$ by optimizing one of the functions, $\alpha(\cdot)$, $\beta(\cdot)$, or $\gamma(\cdot)$, while considering the other two fixed. A detailed description of the algorithm is given in Figure 5.5.

Given: initial codebook, $\mathcal{C}_0$, for initial vector quantizer, Q_0
initial codeword lengths, $l(\gamma(i))$
training set, $\mathcal{T}$
average distortion of initial quantizer, $D(Q_0)$
stopping threshold, ϵ
rate-distortion parameter λ
iteration counter, $m = 1$

Step 1: For each $\mathbf{x} \in \mathcal{T}$, define the vector coder as

$$\alpha(\mathbf{x}) = \arg\min_i \left[d(\mathbf{x}, \mathbf{c}_i) + \lambda l(\gamma(i))\right]$$

where $\mathbf{c}_i \in \mathcal{C}_m$ during iteration m. This produces K partition sets:

$$\mathcal{R}_i = \{\mathbf{x} : \mathbf{x} \in \mathcal{T},\, \alpha(\mathbf{x}) = i\}.$$

Step 2: Form $\mathcal{C}_{m+1}$ and the new vector decoder, $\beta(\cdot)$:

$$\mathcal{C}_{m+1} = \left\{\mathbf{c}_i : \mathbf{c}_i = \beta(i) = \frac{1}{|\mathcal{R}_i|}\sum_{\mathbf{x}\in\mathcal{R}_i}\mathbf{x}\right\}.$$

Step 3: Estimate the new codeword lengths:

$$l(\gamma(i)) = -\log_2 \frac{|\mathcal{R}_i|}{|\mathcal{T}|}$$

Step 4: Estimate the average distortion and average code length for the new codebook:

$$D(Q_{m+1}) = \frac{1}{|\mathcal{T}|}\sum_{\mathbf{x}\in\mathcal{T}} d(\mathbf{x}, \beta(\alpha(\mathbf{x}))), \qquad L(Q_{m+1}) = \frac{1}{N|\mathcal{T}|}\sum_{\mathbf{x}\in\mathcal{T}} l(\gamma(\alpha(\mathbf{x}))).$$

Step 5: Calculate new cost function:

$$J(Q_{m+1}) = D(Q_{m+1}) + \lambda L(Q_{m+1}).$$

Step 6: If

$$\frac{J(Q_m) - J(Q_{m+1})}{J(Q_m)} < \epsilon$$

then stop. Otherwise, set $m = m + 1$ and go to Step 1.

Figure 5.5: The ECVQ algorithm for VQ design [30]. The distortion is assumed to be the squared error.

The ECVQ algorithm uses an estimate of the lengths of the codewords of the variable-length index coder in the form of

$$l(\gamma(i)) = -\log_2 p(i), \tag{5.30}$$

where $p(i)$ is an estimate of the probability that random vector $\mathbf{X}$ belongs to the i^{th} partition. The algorithm factors these length estimates into a vector coder embodying a modified nearest-neighbor rule which implements a cost function based on both rate and distortion. The modified nearest-neighbor rule assigns source vectors to codewords so to minimize the combined rate-distortion cost of the assignment. The resulting entropy of the output of the quantizer, $H(\hat{\mathbf{X}})$, and consequently the rate of the quantizer, are affected. Additionally, one should note that, contrary to the other VQ-design algorithms seen thus far, the quantizer resulting from the ECVQ algorithm is not a nearest-neighbor vector quantizer, since the vector coder does not implement a distortion-based nearest-neighbor mapping.

The parameter λ of the algorithm controls the relative weight that the rate of the vector quantizer plays in the cost function and thus dictates how far below the maximum value of $\frac{1}{N}\log_2 K$ that the rate is constrained to be. Small values of λ emphasize minimizing the distortion over the rate and result in codebooks with relatively high entropy. In fact, if $\lambda = 0$, the ECVQ algorithm reduces to the generalized Lloyd algorithm which minimizes distortion with unconstrained rate. Larger values of λ increase the weighting of the rate in the cost which tends to reduce the rate of the quantizer at the expense of larger average distortion. At $\lambda = \infty$, the resulting quantizer has $\overline{R}(Q) = 0$.

In order to constrain the rate of the VQ system to be less than some value R^*, $0 < R^* < \frac{1}{N}\log_2 K$, we must find a value of λ yielding a vector coder with $\overline{R}(Q) \leq R^*$, which is generally done by trial and error. Once the vector coder is designed meeting the desired rate constraint, $\overline{R}(Q) \leq R^*$, the index coder, $\gamma(\cdot)$, can be designed using any of the entropy-coding techniques discussed in Section 3.3. The resulting average code length, or operational rate, of the final VQ system will satisfy Equation 5.28.

During the ECVQ training procedure, some codewords usually become unpopulated; that is, during Step 1 of Figure 5.5, partition sets $\mathcal{R}_i$ become empty for some i [30]. The ECVQ algorithm deletes the codewords corresponding to these empty partition sets from the codebook resulting in a final codebook that is, in general, smaller than the initial codebook. Methods commonly used to repopulate empty partitions in the generalized Lloyd algorithm are generally not applicable to ECVQ as, although repopulating empty partitions may successfully reduce distortion, this action may also increase the entropy of the quantizer [30].

Because it does not produce a nearest-neighbor vector quantizer, ECVQ is theoretically suboptimal. Usually the resulting quantizer has a higher average distortion than would a quantizer designed by the generalized Lloyd algorithm for the same initial codebook size [30]. However, it has been observed that ECVQ achieves better distortion performance when compared at equal rates [30]. For example, it has been observed that a codebook created by the generalized Lloyd algorithm followed by a separately designed variable-length entropy coder has higher distortion than an ECVQ quantizer operating at the same rate [30].

Thus far, we have presented three popular algorithms for VQ design: ECVQ, Kohonen learning, and the generalized Lloyd algorithm. We have presented these algorithms in a context of designing a vector quantizer for a random-vector source using training samples from this source. In the next section, we will consider using the resulting VQ systems for coding random processes rather than random vectors.

5.4 Nonadaptive Vector Quantization of a Random Process

Up to this point, the discussion of this chapter has been concerned with VQ systems that operate on a single random vector. In this section, we extend the previously presented ideas to the case in which the source is a random process. In this section, we assume that the VQ-design process takes place *offline*, that is, before coding of the random process begins. As the VQ-design algorithms described in the previous section are very computationally intensive, the time required for offline training may be very long. However, the expense of training is incurred only once, because, after the quantizer is designed, it is not changed. A VQ system operating in this manner is said to be *nonadaptive.* Such nonadaptive VQ systems can be optimal in a rate-distortion sense when used to code certain random processes, as we will see below.

More precisely, suppose X_n is a stationary, ergodic random process that we partition into successive blocks of N values each. Let random vector $\mathbf{X}$ be the first block of values. As we have observed previously, an optimal vector quantizer, Q^*, for random vector $\mathbf{X}$ will be a nearest-neighbor vector quantizer. Consequently, Q^* is a block source code as described in Section 4.4. Therefore, by the positive and negative source-coding theorems, Theorems 4.4.1 and 4.4.2, for a given $\epsilon > 0$ and a given average distortion, $D \geq 0$, the optimal vector quantizer will have average distortion, defined by Equation 5.13, satisfying

$$\overline{D}(Q^*) \leq D, \tag{5.31}$$

and rate, defined by Equation 5.10, satisfying

$$R(D) \leq \overline{R}(Q^*) \leq R(D) + \epsilon, \tag{5.32}$$

where $R(D)$ is the rate-distortion function for the random process, *if* the vector dimension N is chosen large enough. An equivalent set of expressions involving the distortion-rate function is also possible via Theorems 4.4.3 and 4.4.4. For a stationary source, these expressions hold for every successive block of N values from the process, where each block is coded with the same vector quantizer, Q^*. Following a discussion similar to that of Section 4.4.1, the time-average per-letter distortion of the coding of the random process converges to the average distortion of the vector quantizer, i.e.,

$$\tilde{\rho}(X_n, \hat{X}_n) \triangleq \lim_{M \to \infty} \frac{1}{M} \sum_{i=1}^{M} \rho(X_i, \hat{X}_i) \to \overline{D}(Q^*) \leq D, \tag{5.33}$$

and the entropy rate of the output process satisfies

$$R(D) \leq \mathcal{H}(\hat{X}_n) \leq R(D) + \epsilon. \tag{5.34}$$

Thus, coding performance arbitrarily close to the rate-distortion function of a stationary, ergodic random process is achievable by an optimal vector quantizer of sufficiently high dimension. This theoretical optimal efficiency has motivated the use of VQ in many practical coding applications. However, the actual practical performance of VQ systems may be quite suboptimal due to several factors that complicate the application of theoretical results to real systems. These complications result in the best performance obtained to date by current VQ systems often being quite far from that which theory predicts is achievable. The first of these problems lies in the fact that the VQ-design algorithms available for practical systems, those algorithms described in Section 5.3, result in locally, rather than globally, optimal quantizers. The second is that it is difficult to create a training set of a practical size that meaningfully captures the statistical behavior of the process for all time. The third complication is that practical systems face a limited range of available vector dimensions that bound the performance of the vector quantizer away from the rate-distortion function. The final problem in the application of theoretical results to practical VQ systems is that the source process can rarely be assumed to be stationary in reality. These problems have prompted a search for more general VQ architectures capable of better practical performance than nonadaptive VQ. A possible solution to these problems lies in the class of algorithms known as adaptive VQ (AVQ). Contrary to the nonadaptive-VQ-system model discussed thus far, AVQ algorithms allow the codebook to be modified while the coding of the source process progresses. This class of algorithms will be the subject of the remainder of this text, beginning with the next chapter.

CHAPTER 6

ADAPTIVE VECTOR QUANTIZATION

In Chapter 5, we presented the theory of nonadaptive VQ and showed that a nonadaptive vector quantizer can achieve rate-distortion performance arbitrarily close to the rate-distortion curve of a stationary, ergodic random process, provided that the vector dimension of the quantizer is sufficiently large. In this chapter, we turn to a more general method of coding, adaptive vector quantization (AVQ). In AVQ, the quantizer is no longer fixed but varies in time as coding progresses. The adaptive nature of AVQ algorithms offers the potential for significant performance improvement over nonadaptive VQ, most importantly in the case of nonstationary sources.

Central to the discussion of rate-distortion theory in Chapter 4 was the concept of an optimal block source code. We saw in Chapter 5 that VQ could be viewed as a practical implementation of block source coding and the VQ-design algorithms of Section 5.3 as attempts to find the optimal block source code for a given stationary source. AVQ algorithms, on the other hand, are generally intended for nonstationary sources, and, as we saw in Section 4.5, the rate-distortion results for nonstationary processes are quite few. Consequently, even though we are able to develop a mathematical definition of AVQ (Section 6.1) that parallels that of nonadaptive VQ (Section 5.1), we are unable to establish corresponding optimal coding properties for the AVQ of nonstationary sources. There have been, however, investigations that establish rate-distortion optimality for AVQ of certain stationary sources; in particular, Zhang and Wei [42] describe an AVQ algorithm and show its rate-distortion optimality for all iid sources with a finite alphabet. However, since our main interest is the development of practical algorithms for the coding of nonstationary rather than stationary sources, we do not consider the results of Zhang and Wei further here, nor do we attempt to derive similar results for the AVQ algorithms under consideration later.

In this chapter, we begin our discussion of AVQ by presenting, in Section 6.1, a mathematical definition of AVQ similar to the one developed for nonadaptive VQ earlier in Section 5.1. Then, in Section 6.2, we limit the generality of our AVQ mathematical definition, producing a model for real AVQ communication systems. We then detail the various components of our AVQ communication-system model and finally define operational measures to gauge the performance of AVQ algorithms implemented under this model. The AVQ communication-system model developed in this chapter will allow, in Chapter 7, the classification of AVQ algorithms from previous literature, as well as an evaluation of their performance in Chapter 9.

6.1 A Mathematical Definition of Adaptive Vector Quantization

In this section, we define mathematically the concept of an adaptive vector quantizer. The mathematical definition presented here will be our most general description of AVQ; this generality will be restricted somewhat in Section 6.2 when we derive our AVQ communication-system model to more realistically describe the operation of practical AVQ algorithms.

Assume that we have initially a random process, X_n. We form random-vector process, $\mathbf{X}_t$, by repetitively grouping N values from X_n to form a sequence of N-dimensional vectors. Then, we define *adaptive vector quantizer*, Q_t, as follows. Let $\mathcal{C}^*$ denote a large *universal codebook*, $\mathcal{C}^* \subseteq \Re^N$, that is fixed for all time t. We define a sequence of *local codebooks*, $\mathcal{C}_t$, such that

$$\mathcal{C}_t \subset \mathcal{C}^* \tag{6.1}$$

at each time t. We restrict each set $\mathcal{C}_t$ to be finite. Adaptive vector quantizer Q_t is a time-variant mapping from N-dimensional Euclidean space to the local codebook for time t; i.e.,

$$Q_t : \Re^N \to \mathcal{C}_t. \tag{6.2}$$

The output of the adaptive vector quantizer is another random-vector process,

$$\hat{\mathbf{X}}_t = Q_t(\mathbf{X}_t). \tag{6.3}$$

The quantized random-vector process, $\hat{\mathbf{X}}_t$, is then unblocked to produce the final, quantized random process, $\hat{X}_n$.

The above mathematical definition of AVQ is sufficiently general to encompass the majority of algorithms reported in prior literature purporting to be "adaptive vector quantization," although there are some exceptions. In our definition, we consider only those algorithms which vary the local-codebook composition as AVQ algorithms. However, other VQ algorithms exist that can be considered, in one way or another, to be adaptive. For example, one algorithm in particular is variable-dimension VQ [2], which features a quantizer that adjusts the vector dimension, N, of the blocking of the input random process and then uses a separate codebook for each vector dimension.

Although such algorithms can be useful in practice, the design procedures associated with techniques such as variable-dimension VQ generally treat these algorithms as collections of nonadaptive source coders rather than one adaptive algorithm. For this reason, we will not consider these types of algorithms further here.

We make some further comments on our AVQ mathematical definition in the following sections. Then in Section 6.2, we discuss how the mathematical definition forms a basis for the design of practical communication systems using AVQ.

6.1.1 The Universal Codebook

The universal codebook, $\mathcal{C}^*$, is one of the main features of the previously presented mathematical definition of AVQ. The intuition behind the use of this universal codebook is that we assume that this large codebook is capable of efficiently coding the random-vector process $\mathbf{X}_t$. As such, the AVQ universal codebook is reminiscent of universal source coding [43, 22]. The theory of universal source coding is rather complex, but can be summarized as follows. Given a class of sources, such as the class of stationary, ergodic sources, a source coder is said to be universal if it can optimally code each of the sources of the class without knowing beforehand which source is being coded [43, 22]. Some approaches to universal source coding use universal codebooks; for example, Neuhoff *et al.* [43] describe a large universal codebook that contains all the optimal block source codes for each of the individual sources of the class. Lossy universal source-coding algorithms have been mainly of theoretical interest [42] and often rely on infinite block lengths and random-coding arguments to prove optimality (e.g., the algorithm of [42]). However, the theory of universal source coding is an important inspiration behind the mathematical definition of AVQ previously presented.

Indeed, if an AVQ algorithm can be proven to achieve rate-distortion optimality for a particular class of sources, it can be considered to be a universal algorithm for that class. Such universal results have been shown for AVQ algorithms (e.g., [42]).

The idea of a universal codebook, also known as a "super-codebook" occasionally, has appeared previously in AVQ literature (e.g., [44, 45, 46]). In these instances, the universal codebook was restricted to be a finite set of vectors. We, however, place no such restriction on the universal codebook, allowing it to be finite, countably infinite, or uncountably infinite. By including such infinite-size codebooks in our mathematical definition, we allow for the most general interpretation of the universal codebook, that is, simply, a set from which we draw codewords to form a local codebook. Such generality allows our mathematical definition to include AVQ algorithms from previous literature that do not explicitly specify a universal codebook. Oftentimes, the universal codebook for these algorithms will be the N-dimensional source alphabet, $\mathcal{X}^N$, or even the entire Euclidean space $\Re^N$. Nonetheless, these will be considered to be valid universal codebooks under our definition of AVQ.

The universal codebook is the source from which the adaptive vector quantizer draws the sequence of local codebooks. We elaborate on the properties of these local codebooks in the next section.

6.1.2 The Local Codebook

Contrary to the universal codebook, which may possibly be infinite, we restrict each local codebook, $\mathcal{C}_t$, to be a finite set. The intuition behind the use of local codebooks is as follows. If we used the universal codebook as the codebook for a fixed, nonadaptive vector quantizer, we would achieve good distortion performance (since the universal codebook ideally is capable of universal source coding), however, the rate would be prohibitively high for practical implementation (because the universal codebook is large, if not infinite). The local codebook is a mechanism for restricting the rate of the coding of the source by allowing only a portion of the universal codebook to be used at a given time. In order to have a finite quantizer rate at each time t, and consequently a finite time-average rate, we restrict each local codebook to be a finite set. We let random process K_t be the size of the local codebook; i.e.,

$$K_t = |\mathcal{C}_t|. \tag{6.4}$$

It is of convenience, at this time, to define an additional random process for describing the content of each local codebook. First, we define a mapping, s_t, that describes the composition of local codebook, $\mathcal{C}_t$, at time t as

$$s_t : \mathcal{C}^* \to \{0, 1\} \tag{6.5}$$

such that

$$\mathcal{C}_t = \{\mathbf{c} : \mathbf{c} \in \mathcal{C}^*, s_t(\mathbf{c}) = 1\}. \tag{6.6}$$

If $\mathcal{C}^*$ is finite or countably infinite, it is possible to order each codeword $\mathbf{c}_i \in \mathcal{C}^*$ with a unique integer index, $i = 1, 2, 3, \ldots$ We then define random variable S_t as

$$S_t = s_t(\mathbf{c}_1) \circ s_t(\mathbf{c}_2) \circ s_t(\mathbf{c}_3) \circ \cdots, \tag{6.7}$$

where the operator $\circ$ denotes the concatenation of binary digits. We can consider this sequence of concatenated binary digits to be a binary fraction if we locate the radix point to the left of $s_t(\mathbf{c}_1)$. Consequently, S_t can be considered to be a random variable taking on values between 0 and 1 for the possible local-codebook compositions at time

t. S_t will be a discrete random variable for $\mathcal{C}^*$ finite, and a continuous random variable for $\mathcal{C}^*$ countably infinite. Thus, random process S_t is called the *codebook-selection process* as it completely describes the composition of each of the local codebooks at time t. In Appendix A, we show, using some ideas from measure theory, how this same codebook-selection process may be defined for the case of an uncountably infinite universal codebook. Consequently, we can assume the existence of S_t for all possible universal codebooks.

We now define two more quantities related to the local codebook. The *learning set* at time t is defined as

$$\mathcal{L}_t = \mathcal{C}_t - \mathcal{C}_{t-1}; \tag{6.8}$$

that is, the learning set is the set of codewords that are added to the local codebook when going from time $t-1$ to time t. Similarly, we define the *forgetting set* as

$$\mathcal{F}_t = \mathcal{C}_{t-1} - \mathcal{C}_t, \tag{6.9}$$

that is, the set of codewords removed from the local codebook.

The learning and forgetting sets are crucial to the performance of an adaptive vector quantizer as they control which codewords the quantizer is allowed to use at a given time. We complete our mathematical definition of AVQ in the next section by giving the definitions of rate and distortion necessary to a discussion of such performance.

6.1.3 Performance Measures of an Adaptive Vector Quantizer

In this section, we define two quantities that measure the performance of an adaptive vector quantizer. We define the *average distortion*, $\overline{D}(Q_t)$, of adaptive vector quantizer Q_t as

$$\overline{D}(Q_t) \triangleq \lim_{T\to\infty} E\left[\frac{1}{T}\sum_{t=1}^{T} d(\mathbf{X}_t, \hat{\mathbf{X}}_t)\right], \tag{6.10}$$

when the expression exists. We will assume that $d(\mathbf{X}_t, \hat{\mathbf{X}}_t)$ is a single-letter distortion measure unless otherwise stated so that the average distortion is units of distortion per vector component.

Similarly, we define the *rate*, $\overline{R}(Q_t)$, of adaptive vector quantizer Q_t as

$$\overline{R}(Q_t) \triangleq \lim_{T\to\infty} \frac{1}{NT} I(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \ldots; \hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, \hat{\mathbf{X}}_3, \ldots), \tag{6.11}$$

when the limit exists.

The expressions of Equations 6.10 and 6.11 have limited utility because their existence is not guaranteed. It is not clear that these expressions will exist even if input random-vector process $\mathbf{X}_t$ is stationary and ergodic, because the stationarity and ergodicity of $\hat{\mathbf{X}}_t$ will depend on the sequence of local codebooks, $\mathcal{C}_t$. Consequently, it is difficult to make general statements about the rate-distortion performance of adaptive vector quantizers. However, if we restrict the generality of our mathematical definition of AVQ, we can define operational rate and distortion measures that form approximations to the true rate and distortion described here while allowing us to empirically evaluate the rate and distortion of AVQ algorithms. In the next section, we present a model for AVQ communication systems that, although less general than our mathematical definition of AVQ, nonetheless accurately represents the practical AVQ algorithms that have been reported in previous literature. Then, in Section 6.2.4, we define operational rate and distortion measures for this AVQ communication-system model that will permit the empirical evaluation of the rate-distortion performance of AVQ algorithms.

6.2 A Model of Communication Systems Using Adaptive Vector Quantization

In this section, we present a model of a communication system using AVQ. This communication system will be similar to the one developed for nonadaptive VQ in Section 5.3 with the addition of components to allow the quantizer to vary in time. The mathematical definition of AVQ presented in Section 6.1 forms the basis of the communication-system model which, in turn, represents a practical, albeit less general, implementation of this mathematical definition.

Figure 6.1 depicts our AVQ communication-system model. We see from this figure that the AVQ communication-system model consists of the nonadaptive-VQ model that was presented in Figure 5.2 with additional components that make the system adaptive. We discuss each of the components of this communication-system model below.

The input to the AVQ communication system is continuous random process X_n. The first component of the system is the *vector blocker* which produces the random-vector process $\mathbf{X}_t$ by blocking together N consecutive values of X_n into a sequence of N-dimensional vectors. The random-vector process $\mathbf{X}_t$ is then fed into the *encoder*. The encoder of the AVQ system consists of the VQ encoder, the codebook selector, the codebook coder, and a codebook decoder.

The *VQ encoder* of the AVQ encoder is a time-varying version of the nonadaptive VQ encoder presented in Section 5.3. The VQ encoder of the AVQ system consists of two components, the vector coder and the index coder. The *vector coder* is a time-varying mapping, $\alpha_t(\cdot)$, that outputs a discrete random process I_t, called the *index*

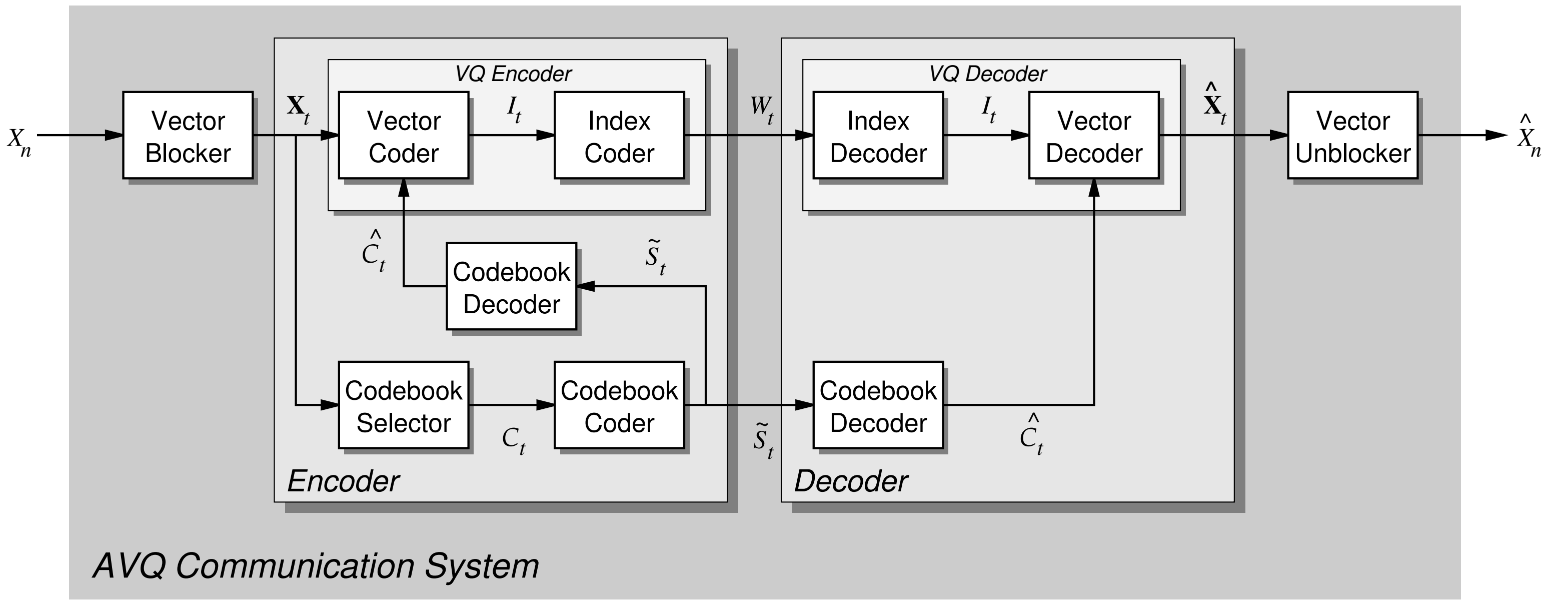


Figure 6.1: Model of a communication system using AVQ

process, given continuous random-vector process $\mathbf{X}_t$ as input; i.e.,

$$I_t = \alpha_t(\mathbf{X}_t). \tag{6.12}$$

We define mapping $\alpha_t(\cdot)$ so that

$$I_t = \alpha_t(\mathbf{X}_t) \triangleq i \quad \text{when} \quad Q_t(\mathbf{X}_t) = \mathbf{c}_i, \tag{6.13}$$

where $\mathbf{c}_i \in \hat{\mathcal{C}}_t$ and $1 \leq i \leq K_t$. That is, the vector coder maps input vectors to codewords of the current approximation, $\hat{\mathcal{C}}_t$, to the local codebook. For most AVQ algorithms, the vector coder is simply a nearest-neighbor mapping with respect to some distortion measure, although a few algorithms, such as our new AVQ algorithm presented in Chapter 8, use a more sophisticated mapping based on both rate and distortion. We note that the vector coder actually uses an approximation, $\hat{\mathcal{C}}_t$, to the local codebook instead of the local codebook itself so that the decoder and the encoder function identically. This point will be elaborated below.

The *index coder* of the VQ encoder is a time-varying, one-to-one function, $\gamma_t(\cdot)$, that maps each index to a set of K_t codewords, w_i; i.e.,

$$\gamma_t : \{1, 2, \ldots, K_t\} \to \{w_1, w_2, \ldots, w_{K_t}\}, \tag{6.14}$$

where each w_i is a string of binary digits, and $K_t = |\hat{\mathcal{C}}_t| = |\mathcal{C}_t|$. The output of the index coder, discrete random process W_t, is sent to the AVQ system decoder. Although more general index coders are possible, i.e, those that map more than one index symbol to a codeword, the index coder as defined here, which maps one index symbol to one codeword, is nearly always the method of choice for practical AVQ algorithms. We investigate this topic further in Section 6.2.1.

The AVQ-system encoder contains several components that give the AVQ system its adaptive nature, namely, the codebook selector, the codebook coder, and the codebook decoder. The *codebook selector* monitors the input random-vector process and creates a sequence of local codebooks. The *codebook coder* transmits a description of this sequence of local codebooks to the decoder. The codebook-sequence information is conveyed to the decoder in the form of random process $\tilde{S}_t$, which is an approximation to codebook-selection process S_t. In general,

$$\tilde{S}_t \neq S_t. \tag{6.15}$$

Since both the codebook selector and the codebook coder are crucial to the performance of the AVQ communication system, we discuss them in greater detail below in Sections 6.2.2 and 6.2.3.

The AVQ-system decoder consists of the VQ decoder and a codebook decoder. The *codebook decoder* reconstructs a sequence of approximate local codebooks, $\hat{\mathcal{C}}_t$, from the random process $\tilde{S}_t$, which is the codebook-selection information transmitted to the decoder. The AVQ-system encoder also has a copy of the codebook decoder. Since the codebook-selection information conveyed to the decoder, $\tilde{S}_t$, is not necessarily the same as the codebook-selection process, the sequence of local codebooks produced by the codebook decoder will, in general, differ from the true sequence of local codebooks, $\mathcal{C}_t$. However, both the encoder and the decoder use the same approximate local codebook during coding. We discuss the implications of the use of an approximate local codebook sequence along with the design and operation of the codebook coder below in Section 6.2.3.

The *VQ decoder* is similar to the decoder of the nonadaptive-VQ system of Section 5.3, except that it is time-varying. The AVQ-system VQ decoder consists of the index decoder and the vector decoder. The *index decoder* is simply the inverse function, $\gamma_t^{-1}(\cdot)$, of the index coder of the AVQ-system VQ encoder. The *vector decoder* receives the index process output by the index decoder and outputs the corresponding sequence of codewords from the local codebook. The vector decoder is consequently a time-varying one-to-one function, β_t, from the set of indices to the local codebook of the decoder; i.e.,

$$\beta_t : \{1, 2, \ldots, K_t\} \to \hat{\mathcal{C}}_t. \tag{6.16}$$

The operation of the AVQ system is the concatenation of all the operations,

$$\hat{\mathbf{X}}_t = \beta_t(\gamma_t^{-1}(\gamma_t(\alpha_t(\mathbf{X}_t)))) = Q_t(\mathbf{X}_t), \tag{6.17}$$

which implements the adaptive vector quantizer, Q_t. Finally, the *vector unblocker* of the AVQ system produces the random process $\hat{X}_n$ from the random-vector process $\hat{\mathbf{X}}_t$ by unblocking the N-dimensional vectors.

The vector coder and vector decoder of the AVQ system are the same as the corresponding components of the nonadaptive-VQ system described thoroughly before (Section 5.3), with the exception that the codebook used for the coding is time-varying. However, the other components of the AVQ system, namely the index coder, the codebook selector, and the codebook decoder warrant further discussion. We elaborate on the operation and typical implementation of each of these components below in Sections 6.2.1 through 6.2.3.

6.2.1 The Index Coder

The task of the index coder in the AVQ-system encoder is to generate an efficient, lossless coding of the index process, I_t. Assuming for the moment that $\mathbf{X}_t$ and $\hat{\mathcal{C}}_t$ are such that index process I_t is stationary, then, ideally, the job of the index coder is to produce a coding whose average length approaches the entropy rate of the index process. As we saw in Section 3.3, such a coding is theoretically possible if the index coder uses a prefix code with infinite blocking of symbols from the index process. In practice, such infinite blocking is not possible, so one usually settles for a prefix code that approaches the first-order entropy of the index process.

Practical AVQ algorithms usually feature an index coder that outputs one codeword for each symbol of the index process. Consequently, in Section 6.2, we defined the index coder as function $\gamma_t(\cdot)$,

$$W_t = \gamma_t(I_t), \tag{6.18}$$

featuring single-symbol blocking of the index process. Assuming single-symbol blocking, we define the *average code length* of the index coder as

$$\tilde{L}(W_t) \triangleq \lim_{T\to\infty} \frac{1}{NT} \sum_{t=1}^{T} l(\gamma_t(\alpha_t(\mathbf{X}_t))), \tag{6.19}$$

when the limit exists. Note that $l(w_i)$ is the length in bits of codeword w_i and that the average code length has units of bits per vector component.

It is common in AVQ algorithms that the index coder is further simplified so that function $\gamma_t(\cdot)$ is not time-varying; i.e., the index coder uses a static mapping, $\gamma(\cdot)$. In the case of a *static-mapped* index coder, one assumes that the size of the local codebook is constant; i.e.,

$$K_t = K, \tag{6.20}$$

for all time t. In general, there are two types of static mappings. The first, and most simple, is the *fixed-length* index coder. For a fixed-length index coder, one assigns to each index the fixed-length code corresponding to its binary representation. Consequently the average code length for a fixed-length index coder is

$$\tilde{L}(W_t) = \frac{\lceil \log_2 K \rceil}{N}. \tag{6.21}$$

The other type of static-mapped index coder is a *variable-length* index coder. In a variable-length index coder, some form of entropy coding, such as Huffman coding [10], is used so that the average code length of the index coder approaches the first-order entropy of the index process. Usually it is necessary that, after the index coder is designed for a particular source, a code table be transmitted to the decoder.

Entropy coding is also used in *dynamic* index coders, those index coders which are not static-mapped. With dynamic index coders, it is common to make use of multi-symbol blocking. Arithmetic coding [12] and Lempel-Ziv coding [13, 14] are two examples of entropy-coding techniques useful in the design of such index coders. Dynamic index coders are sometimes preferred in the index-coder design as they typically do not rely on the transmission of a predesigned code table to the decoder. However, one should note that, even though variable-length (both static and dynamic) index coders are, in general, capable of more efficient coding performance than fixed-length coders, fixed-length coders are sometimes used in practical implementations because they do not require the complicated buffer schemes (e.g., [47]) often needed to match variable-length coders to fixed-rate channels.

In this section, we have surveyed the different possible types of index coders. The design of these index coders, particularly those involving entropy coding, can be quite complicated and is beyond the scope of this discussion. Indeed, most of the presentations of AVQ algorithms in previous literature either ignore the problem of the design of the index coder, assume a simple, fixed-length index coder, or assume the existence of a perfect dynamic index coder yielding an average code length equal to the first-order entropy of the index process. Since the above-mentioned dynamic entropy-coding techniques do theoretically achieve the first-order entropy, we will also take the latter approach to the index-coder problem by assuming, from this point on, the availability of such a perfect index coder. The interested reader is referred to the book [11] by Williams for more details on the implementation of suitable dynamic index coders.

6.2.2 The Codebook Selector

The task of the codebook selector is to determine, at each time t, the learning and forgetting sets, $\mathcal{L}_t$ and $\mathcal{F}_t$, as defined in Equations 6.8 and 6.9. The codebook selector can be decomposed, as shown in Figure 6.2, into two components, the *learning process* and the *forgetting process.* The learning process creates the learning set and the forgetting process creates the forgetting set. The local codebook is then formed as

$$\mathcal{C}_t = \mathcal{C}_{t-1} \cup \mathcal{L}_t - \mathcal{F}_t. \tag{6.22}$$

The learning process of the codebook selector usually implements a modified version of one the nonadaptive-VQ training algorithms described in Sections 5.3.1 through 5.3.3. As the operation of the learning process is crucial to the performance of the AVQ system, the method of implementation of this process provides the best means of differentiating between the various reported AVQ algorithms. Consequently, the learning process will provide a basis for the development of a taxonomy of prior AVQ algorithms later in Chapter 7.

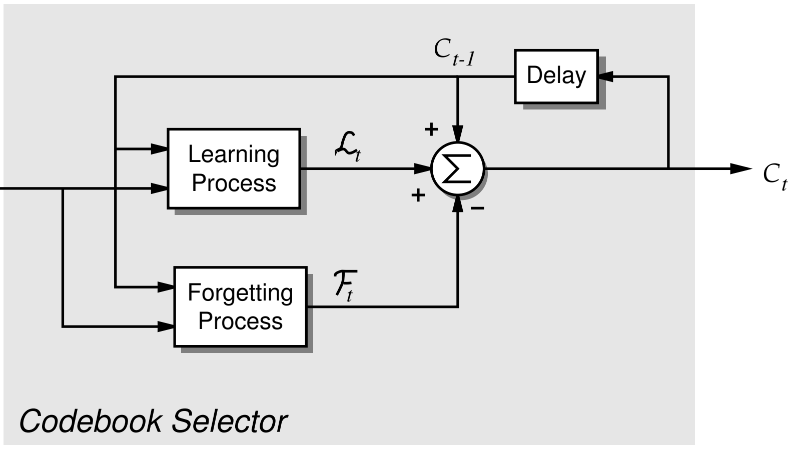


Figure 6.2: The codebook selector of the AVQ communication system. The Σ operator implements set-union and set-difference operations.

We identify, however, at this time, two very broad categories of learning processes: *backward adaption* and *forward adaption* [2]. In backward adaption, the adaptive vector quantizer can determine the learning set from only past quantities. In backward adaption based on the *true past*, the quantizer can use past values of the codebook-selection process and past values of the source process. In backward adaption based on the *coded past*, the quantizer can use past values of the codebook-selection process and past values of the output of the quantizer. In forward adaption, the quantizer can determine the learning set from quantities from both the past and the future. More specifically, for backward adaption based on the true past,

$$S_t = g(S_{t-1}, S_{t-2}, \ldots, S_1, \mathbf{X}_{t-1}, \mathbf{X}_{t-2}, \ldots, \mathbf{X}_1), \tag{6.23}$$

where $g(\cdot)$ is some function. Similarly for backward adaption based on the coded past,

$$S_t = g(S_{t-1}, S_{t-2}, \ldots, S_1, \hat{\mathbf{X}}_{t-1}, \hat{\mathbf{X}}_{t-2}, \ldots, \hat{\mathbf{X}}_1). \tag{6.24}$$

For forward adaption,

$$S_t = g(S_{t-1}, S_{t-2}, \ldots, S_1, \mathbf{X}_{t+\tau}, \mathbf{X}_{t+\tau-1}, \ldots, \mathbf{X}_{t+1}, \mathbf{X}_t, \mathbf{X}_{t-1}, \mathbf{X}_{t-2}, \ldots, \mathbf{X}_1), \tag{6.25}$$

for some value τ.

Contrary to the learning process, for which nearly every AVQ algorithm has a different implementation, the forgetting process is usually chosen to be one of two types: *least-recently used* (LRU) (see, for example, [48]) or *least-frequently used* (LFU) (see, for example, [49]). With LRU forgetting, the codebook selector removes codewords from the local codebook that have not been used recently in the coding of the source. It is assumed that, if these codewords have not been used recently, then they are no longer relevant to the current statistics of the source. With LFU forgetting, the quantizer removes codewords that have not been used often in the coding of the source. LFU and LRU are, in general, very similar since one would expect that the frequency of use of a codeword and the time between successive uses of a codeword are closely related quantities. We will see examples of AVQ algorithms for both types of forgetting processes in Chapter 7.

The common assumption made for most AVQ algorithms is that, even if the source $\mathbf{X}_t$ is nonstationary, the statistics of the source do not vary greatly between consecutive vectors. In fact, it is generally assumed that there exists some *local stationarity*; that is, the statistics of vectors $\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots \mathbf{X}_{t+\tau}$ are approximately stationary over a *local context* τ, which is suitably small in comparison to the total length of the nonstationary process under consideration. If we assume that local stationarity holds, we can expect that the local codebooks of the adaptive vector quantizer will not differ greatly between time t and time $t-1$. Consequently, both the learning and forgetting sets will generally contain only a small number of vectors each. Indeed, some AVQ algorithms allow only one codeword to be added to the local codebook at any time ($|\mathcal{L}_t| = |\mathcal{F}_t| \leq 1$). Other AVQ algorithms allow the local codebook to change only after an interval of τ vectors have been processed. We will see examples of both of these types of algorithms in Chapter 7.

Unless otherwise stated, we will assume that the codebook selector does not allow the size of the local codebooks to increase or decrease over time. That is, we assume that

$$K_t = K. \tag{6.26}$$

This assumption allows a static-mapped index coder to be used, and thus facilitates the maintaining of an approximately constant number of bits per symbol output from the index coder, a property often desirable in real AVQ implementations. We note, however, that our general model can accommodate the variation over time of the size of the local codebooks which can, for example, be used in AVQ systems that dynamically add or remove codewords to maintain constant distortion performance.

Because the codebook selector determines which codewords can be used by the AVQ system at any time, it is very crucial to the performance of the system. Even though the implementation of the index coder and codebook coder can greatly affect the performance of the system, they cannot improve upon the fundamental performance allowed by the codebook selector. For example, a "good" index coder will achieve a coding close to the entropy rate of the index process. However, it is the codebook selector that ultimately determines the magnitude of this entropy rate. The codebook coder is similarly dependent on the codebook selector. Consequently, if the codebook selector is performing poorly, then the performance of the entire system will suffer. We therefore consider the codebook selector to be the most important component of an AVQ system. In Chapter 7, when we describe various AVQ algorithms that have appeared in previous literature, we will concentrate our discussion mainly on the codebook selector.

6.2.3 The Codebook Coder

Once the codebook selector has determined the local codebook for time t, a description of its contents must be conveyed to the decoder. Consequently, the task of the codebook coder is to produce an efficient coding of the codebook-selection process, S_t, to transmit to the decoder. We represent the information transmitted to the decoder as random process $\tilde{S}_t$. In general, $\tilde{S}_t$ is a sequence of codewords, each of which being a string of binary digits. We consider only the case that $\tilde{S}_t$ implements a prefix code (see Section 3.3).

To measure the performance of the codebook selector, we introduce the *average codebook code length*, defined as

$$\tilde{L}(\tilde{S}_t) \triangleq \lim_{T\to\infty} \frac{1}{NT} \sum_{t=1}^{T} l(\tilde{S}_t), \tag{6.27}$$

when the limit exists. Note that $l(\tilde{S}_t)$ is the length in bits of binary string $\tilde{S}_t$ and that the average codebook code length has units of bits per vector component. The average codebook code length is often called *side information* [2] because it can be considered to be information that is "sent on the side" to the decoder to describe the adaptive updating of the local codebooks. The operation of the codebook coder and, consequently, the value of the average codebook code length depend on the universal codebook.

The most simple case is when the universal codebook is a finite set of $K^* = |\mathcal{C}^*|$ elements. Assuming that the size of all the local codebooks is the same for all time

t, there are $\binom{K^*}{K}$ possible local codebooks that can be drawn from $\mathcal{C}^*$. Consequently, we have

$$\mathcal{H}(\tilde{S}_t) \leq \tilde{L}(\tilde{S}_t) \leq \frac{1}{N}\left\lceil \log_2 \binom{K^*}{K} \right\rceil, \tag{6.28}$$

assuming that the entropy rate of $\tilde{S}_t$ exists. Equality on the left side of Equation 6.28 would require a variable-length code for $\tilde{S}_t$, the design of which would be very complex. Equality on the right side of Equation 6.28 is achieved when a fixed-length code is used for $\tilde{S}_t$; however, the design of such a fixed-length code is also a very complex problem, although solutions do exist [45]. To alleviate the complexity of the design of a fixed-length codebook coder, Zeger *et al.* [45] suggest two simple techniques that yield $\tilde{L}(\tilde{S}_t) > \frac{1}{N}\left\lceil \log_2 \binom{K^*}{K} \right\rceil$. The first of these techniques features codewords v_i that each consist of K binary strings, each string being the "address" of one of the K^* codewords in $\mathcal{C}^*$. Consequently this scheme has an average codebook code length of

$$\tilde{L}(\tilde{S}_t) = \frac{K}{N}\left\lceil \log_2 K^* \right\rceil \tag{6.29}$$

[45]. The second simple technique is to let $\tilde{S}_t = S_t$ directly. Recall that, in Section 6.1.2, we defined S_t for a finite universal codebook as

$$S_t \triangleq s_t(\mathbf{c}_1) \circ s_t(\mathbf{c}_2) \circ \cdots \circ s_t(\mathbf{c}_{K^*}), \tag{6.30}$$

where $s_t(\cdot)$ was a binary mapping indicating whether or not codeword $\mathbf{c}_i \in \mathcal{C}^*$ was in $\mathcal{C}_t$. The average codebook code length for this codebook coder is then

$$\tilde{L}(\tilde{S}_t) = \frac{K^*}{N} \tag{6.31}$$

[45]. The first of these two simple approaches will have a smaller average codebook code length when

$$K < \frac{K^*}{\log_2 K^*}, \tag{6.32}$$

a choice dependent on the relative sizes of the local codebook and the universal codebook [46].

Note that, for a finite universal codebook, the codebook-coder schemes mentioned above transmit the local codebook to the decoder without distortion. That is, referring to Figure 6.1, $\hat{\mathcal{C}}_t = \mathcal{C}_t$. However, such lossless transmission of the local codebook is not possible in the case that the universal codebook is an infinite set. In this case, it is necessary to introduce some distortion while transmitting the local codebook to the decoder.

The effect from using a local codebook $\hat{\mathcal{C}}_t$ that is different from the "true" local codebook designed by the codebook selector has been studied by Zeger *et al.* [45] and Chou and Effros [50], assuming that the VQ encoder is fixed rate (i.e., W_t is a fixed-length code). Zeger *et al.* [45] have determined that, assuming a stationary source and high-resolution-VQ approximations, the total distortion incurred by the AVQ system can be broken into a distortion associated with the vector coder and local codebook $\mathcal{C}_t$ plus a distortion due to the fact that both the encoder and the decoder use only an approximation, $\hat{\mathcal{C}}_t$, to the true local codebook (the "codebook mismatch" distortion). Zeger *et al.* further proceed to analyze two approaches for creating $\hat{\mathcal{C}}_t$ from $\mathcal{C}_t$. In the first of these two approaches, each codeword component of $\mathcal{C}_t$ is quantized with a uniform scalar quantizer. In the second approach, each codeword of $\mathcal{C}_t$ is vector quantized to a large codebook (which Zeger *et al.* call a "universal codebook" although, technically, it is not the same universal codebook $\mathcal{C}^*$ which we use in the codebook selector). For a fixed overall rate of the AVQ system, there is a tradeoff on the allocation of the available bits to the vector coder and the codebook coder, as the total distortion of the AVQ system depends on the performance of each component. Chou and Effros [50] developed a formula for the optimal tradeoff, and Zeger *et al.* [45] have analyzed the tradeoff for the two types of codebook coders, concluding that the two approaches perform equally well.

Zeger *et al.* [45] and Chou and Effros [50] have shown that the design of the codebook coder in the case of infinite codebooks is a complex problem. Additionally, the fact that they assume a stationary source and make high-resolution VQ assumptions on the vector coder mean that their work is of questionable applicability to practical AVQ algorithms. However, most designers of AVQ algorithms, when confronted with an infinite universal codebook, have chosen to use a simple codebook coder, such as quantization with a uniform scalar quantizer, and ignore the distortion incurred by using an approximation to the true local codebook. If the resolution of such a scalar quantizer is sufficiently high so that the distortion due to codebook mismatch is negligible, we can assume that $\hat{\mathcal{C}}_t = \mathcal{C}_t$. This simplifying assumption allows us to model our AVQ system more simply than Figure 6.1; our simplified AVQ-communication-system model is shown in Figure 6.3.

In this simplified AVQ-communication system model, it is assumed that $\hat{\mathcal{C}}_t = \mathcal{C}_t$. In theory, such an assumption requires that the average codebook code length, $\tilde{L}(\tilde{S}_t)$, be very large, or infinite in the case of an infinite universal codebook. As we will see in Chapter 7, most AVQ algorithms place some practical limit on the amount of side information sent to the decoder. The most common, and simplest, approach is to use a codebook coder that implements a uniform, high-resolution scalar quantizer. In such an approach, each codeword that is added to the local codebook is coded with

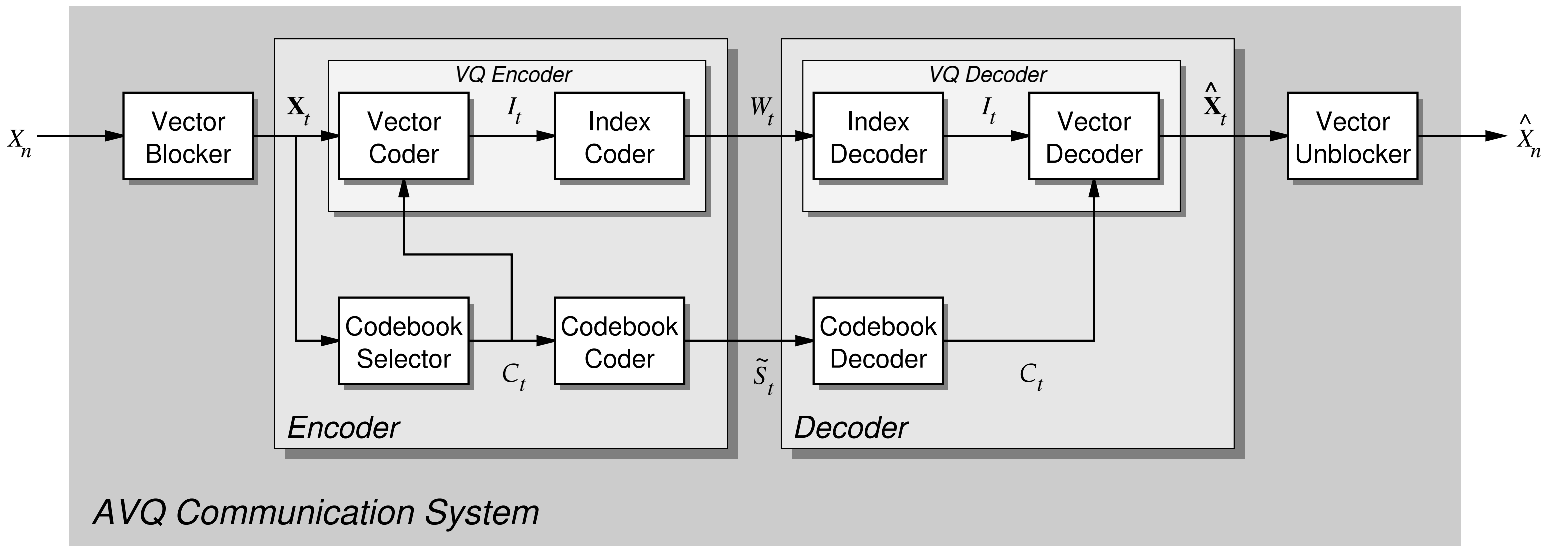


Figure 6.3: A simplified model of a communication system using AVQ

some fixed, finite number of bits per vector component. The resolution of the scalar quantizer is chosen suitably large so that negligible distortion is introduced between $\mathcal{C}_t$ and $\hat{\mathcal{C}}_t$.

In this scalar-quantizer approach, the codebook coder describes the local-codebook sequence incrementally by coding the contents of the both learning set, $\mathcal{L}_t$, and the forgetting set, $\mathcal{F}_t$, rather than by identifying each local codebook as a subset of the universal codebook. In this case, the learning set is described as a fixed number of bits per vector component by the scalar quantizer, while the forgetting set is coded by a set of indices that indicate which old codewords are removed from the local codebook. Successful implementation of this type of codebook coder relies on the assumption that both the learning and forgetting sets contain very few vectors. Under this assumption, the side information required by the codebook coder is usually small in comparison to the total rate of the AVQ system.

We emphasize that a high-resolution scalar quantizer is only one approach to the implementation of a codebook coder, and we argue that this approach is sufficient when the side information accounts for only a small portion of the total rate of an AVQ system. When the side information becomes a sizeable portion of the total rate, more effort in the design of the codebook coder is warranted. Specifically, the resolution of the scalar quantizer should account for the tradeoff, in terms of rate-distortion performance, that exists between the vector coder and the codebook coder. However, the mathematics of this tradeoff [45, 50] have been established only for stationary sources and are generally not invoked in practice. Consequently, in the experimental comparison of AVQ algorithms that follows in Chapter 9, we use a simple, high-resolution scalar-quantizer codebook coder for each algorithm. Doing so facilitates the experiments by permitting the isolation of the codebook selector, which we argued was the most important component of an AVQ system, from effects due to the codebook coder. However, we caution that, when side information is a substantial component of the total rate, more complex design techniques for the codebook coder may achieve performance more efficient than that of our assumed scalar quantizer.

6.2.4 Operation Performance Measures for AVQ Communication Systems

In Section 6.1.3, we defined the theoretical rate and distortion of AVQ based on the mathematical definition presented in Section 6.1. In this section, we define two operational counterparts that measure the performance of AVQ communication systems based on the model presented in Section 6.2. These two operational performance measures will be of utility later in Chapter 9 when we compare the performance of various AVQ algorithms.

We define the *operational average distortion* as the time average of the single-letter distortion, $d(\mathbf{X}_t, \hat{\mathbf{X}}_t)$, between the input and output random-vector processes; i.e,

$$\tilde{D}(Q_t) \triangleq \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} d(\mathbf{X}_t, \hat{\mathbf{X}}_t), \tag{6.33}$$

when the limit exists. This operational average distortion is also expressible in terms of the time-average per-letter distortion; i.e.,

$$\tilde{D}(Q_t) = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \rho(X_n, \hat{X}_n), \tag{6.34}$$

when the limit exists.

We define the *operational rate* of the AVQ communication system as

$$\tilde{L}(Q_t) \triangleq \tilde{L}(W_t) + \tilde{L}(\tilde{S}_t), \tag{6.35}$$

where the average code length, $\tilde{L}(W_t)$, of the index coder was defined in Equation 6.19, and the average codebook code length, $\tilde{L}(\tilde{S}_t)$, of the codebook coder was defined in Equation 6.27. Consequently, we see that the operational rate of the AVQ communication is composed of a rate due to the vector quantizer, $\tilde{L}(W_t)$, plus a rate due to the codebook coder, $\tilde{L}(\tilde{S}_t)$. As mentioned before, this second quantity is often called side information [2] because it can be considered to be information that is "sent on the side" to the decoder to control the adaptive updating of the local codebooks used by the vector quantizer.

As a final note, we observe that we have qualified the definitions of both the operational rate and distortion to be defined only when the limits in their definitions exist. These qualifications are a technical necessity because each definition is an infinite time average which may or may not converge. This technicality will pose us no problem as we will use these definitions solely to calculate empirical estimates of the rate and distortion obtained by AVQ algorithms. More specifically, in calculating the rate-distortion results in Chapter 9, we will necessarily have to use finite-length samples of random processes. Consequently, we use the operational rate and distortion definitions of Equations 6.35 and 6.33 for a fixed length T without the limiting operation. Of course, the operational rate and distortion thusly defined will always exist over a finite sample length.

This section concludes the discussion of the current chapter on the theoretical foundations of AVQ. In the next chapter, we review specific AVQ algorithms that have appeared in prior literature. The theory presented thus far will enable us to identify theoretically significant differences between the algorithms, upon which an algorithm taxonomy will be built.

CHAPTER 7

PREVIOUS ADAPTIVE-VECTOR-QUANTIZATION ALGORITHMS

In this chapter, we examine some specific implementations of AVQ that have appeared in previous literature. Typically, AVQ algorithms attempt to apply the rate-distortion theory developed in Chapter 4 and the theory of VQ presented in Chapter 5 as heuristic guidelines to the coding of nonstationary sources.

In Section 4.4, we discussed the source coding of stationary random processes. In that section, we identified two main approaches, constrained-distortion source coding and constrained-rate source coding. In the first approach, we considered the set of codes with distortion less than some maximum D, and stated the existence of an optimal code from this set which yielded the minimum rate. In the second approach, we considered the opposite case of the set of codes with rate less than some maximum R, and stated the existence of an optimal code from that set yielding the minimum distortion. Theoretically, both approaches are equivalent because they both provide for an optimal code that asymptotically yields performance on the rate-distortion curve for stationary random processes.

These two main approaches to source coding have also been applied to the coding of nonstationary sources, yielding AVQ algorithms of two general types. Constrained-distortion AVQ algorithms follow the example set by constrained-distortion source coding. In constrained-distortion AVQ, the codebook selector limits the distortion produced by the algorithm to some maximum value. The job of the algorithm is then to find the coding of the source that produces the smallest rate subject to this distortion constraint. Constrained-rate AVQ algorithms do the opposite, limiting the rate to be less than or equal to some maximum value and attempting to produce a coding with the smallest distortion, similar to constrained-rate source coding.

In theory, both the constrained-distortion and constrained-rate AVQ approaches are capable of optimal coding of stationary sources, assuming an asymptotically infinite codeword dimension. However, in practice, one is often faced with nonstationary sources as well as a limited vector dimension. Consequently, it has been recognized in recent literature [46, 51] that, for AVQ algorithms designed to operate in a practical setting, neither approach is sufficient by itself. The codebook selector of AVQ algorithms should ideally monitor both rate and distortion simultaneously to ensure that each updating of the local codebook is performed in a manner that is favorable to the rate-distortion performance of the algorithm. That is, one should not "greedily" lower distortion without regard to the cost in rate, or vice versa. This observation has led to the introduction of a third category of AVQ algorithms, namely rate-distortion-based AVQ algorithms, which have codebook selectors based on the minimization of rate-distortion cost functions. Additionally, rate-distortion-based AVQ algorithms usually employ vector coders implementing nearest-neighbor mappings based on these rate-distortion cost functions.

In this chapter, we present the details of AVQ algorithms that have appeared in prior literature. Many of the algorithms have a complicated structure and are usually presented with several variations. However, the discussion here will generally focus on the simplest structure available for each algorithm. Additionally, following the discussion of Section 6.2, any implementation details pertaining to the construction of the index coder or the codebook coder will usually be omitted. The reasons for these omissions are several. First, we consider the codebook selector to be the most important module of an AVQ communication system since the performance of the other modules is dependent on it. Therefore, we would like to evaluate the performance of an AVQ algorithm by considering the operation of the codebook selector alone, without any affects from the performance of the other modules. Secondly, most of the designs for index coders and codebooks coders presented with AVQ algorithms are *ad hoc* and are not suited to general data sets. Finally, as discussed in Sections 6.2.1 and 6.2.3, we can assume "ideal" index coders that achieve the first-order entropy of the index process, and "ideal" codebook coders that transmit the codebook to the decoder while suffering negligible distortion due to codebook mismatch. By assuming these "ideal" modules in our algorithms, we provide an equal basis for each AVQ algorithm under consideration, allowing us to isolate the performance of the codebook selector from that of the other components.

In Sections 7.1 through 7.3 of this chapter, we discuss each of the three categories of AVQ algorithms in greater detail. Additionally, we examine several algorithms from each category that have appeared in previous literature. Although there have been numerous AVQ algorithms reported, many are quite similar from a theoretical viewpoint; consequently, we will give only a limited number of algorithms a detailed examination. These algorithms will typically have played a historically prominent role in the AVQ literature, or will have brought interesting ideas to light. For the algorithms that we discuss in detail, we present a generic algorithm description in this chapter; the interested reader may find the source code (MATLAB M-file code) for the algorithms in Appendix B. We conclude this chapter in Section 7.4 with a taxonomy of AVQ algorithms that effectively summarizes the discussion of the chapter. Afterwards, we proceed to Chapter 8 wherein we present a new AVQ algorithm.

7.1 Constrained-Distortion AVQ Algorithms

The class of constrained-distortion AVQ algorithms includes what may be the first AVQ algorithm to appear in the literature, the speech coder developed by Paul [52] in 1982. Since then, however, there have been considerably fewer constrained-distortion AVQ algorithms in the literature than constrained-rate algorithms, perhaps due to a bias towards the minimization of distortion rather than rate.

The main idea behind constrained-distortion AVQ is to restrict the distortion to be below a certain maximum, and then concentrate on reducing as much as possible the rate achieved by the algorithm. All the constrained-distortion AVQ algorithms in previous literature use what we call *threshold replenishment*. In the threshold-replenishment technique, the codebook selector updates the local codebook only if the current source vector cannot be coded to within some threshold on the distortion using the current local codebook. In that case, the codebook selector chooses a new codeword from the universal codebook and adds it to the local codebook. The codebook selector must be designed so that this new codeword is within a distance equal to the distortion threshold from the current source vector. The vector coder for constrained-distortion algorithms then implements a distortion-based nearest-neighbor quantizer using the local codebook given to it by the codebook selector.

If we denote the distortion threshold of a threshold-replenishment algorithm as $D_{\max}$, then we note that, for each source vector $\mathbf{X}_t$, the distortion between the source vector and its quantization is at most $D_{\max}$; i.e.,

$$d(\mathbf{X}_t, \hat{\mathbf{X}}_t) \leq D_{\max}, \tag{7.1}$$

for all time t. From Equation 6.33, we have

$$\tilde{D}(Q_t) = \frac{1}{T} \sum_{t=1}^{T} d(\mathbf{X}_t, \hat{\mathbf{X}}_t) \leq D_{\max}, \tag{7.2}$$

for any finite process length of T vectors. Thus, in a threshold-replenishment algorithm, the operational average distortion, $\tilde{D}(Q_t)$, is constrained to be at most $D_{\max}$.

One can envision two methods for specifying the universal codebook for threshold-replenishment algorithms; we call these the *explicit* and *implicit* methods. In the first, the universal codebook is explicitly specified to the algorithm. That is, the codebook selector stores the universal codebook explicitly as a finite set of vectors. Then, if the nearest-neighbor codeword in the local codebook is too far from the current source vector, a nearest-neighbor search over the entire universal codebook is performed to find the closest codeword, which is then added to the local codebook. One constructs the universal codebook so as to guarantee the existence of at least one codeword within the distortion-threshold distance for every possible input vector.

Because a nearest-neighbor search through a large universal codebook can be quite computationally complex, the usual approach to threshold replenishment is to specify the universal codebook *implicitly*. A codebook selector with an implicitly defined universal codebook does not store a set of vectors. Rather, the universal codebook is implied by a function defined on $\Re^N$. This function is designed to yield the closest codeword in the universal codebook to any given point in $\Re^N$. One common approach to implicit threshold-replenishment AVQ is to use a lattice for the universal codebook (see [2, 48]).

To our knowledge, all the constrained-distortion AVQ algorithms that have been reported in previous literature have been implicit threshold-replenishment algorithms. In what follows, we present a brief survey of these algorithms. We then discuss two of these algorithms in greater detail in Sections 7.1.1 and 7.1.2.

Paul [52] developed a threshold-replenishment AVQ algorithm that was perhaps the first AVQ algorithm ever reported. The Paul algorithm uses threshold replenishment with the new codeword being simply the source vector itself (in which case the universal codebook can be considered to be $\mathcal{X}^N$). The Paul algorithm is perhaps the simplest AVQ algorithm to be published to date; however, as we will see in Chapter 9, its performance in practice is better than that of most other AVQ algorithms. Because of these advantages, as well as its historical prominence, we discuss the Paul algorithm in detail in Section 7.1.1.

Another threshold-replenishment algorithm was described by Wang *et al.* [48]. This threshold-replenishment algorithm is similar to the Paul algorithm, except that the universal codebook takes the form of a lattice vector quantizer. This algorithm is examined in more detail below in Section 7.1.2.

Finally Chen, Sheu, and Zhang [53] present an algorithm based on the Gold-Washing method originally developed by Zhang and Wei [42]. The Chen-Sheu-Zhang algorithm features a threshold-replenishment technique that uses an interpolation scheme to calculate the new codeword based on the local codebook and the current source vector. Unfortunately, this interpolation was heuristically designed for the coding of images, and is thus not applicable to general data sources. Additionally, the interpolation is not done in such a way so as to guarantee that the new codeword is within the distortion-threshold distance to the current source vector. The Gold-Washing algorithm, however, is a complicated forgetting process that seems to be theoretically superior to LRU and LFU techniques, although it is not clear from [42] how well the Gold-Washing algorithm performs in practice relative to these two simpler approaches. Because its limited applicability to general data sets, we do not consider the Chen-Sheu-Zhang algorithm further here; however, we do present the algorithms due to Paul [52] and Wang *et al.* [48] in greater detail in Sections 7.1.1 and 7.1.2, respectively.

7.1.1 The Paul Algorithm

The Paul algorithm [52] is perhaps the simplest AVQ algorithm to be reported in AVQ literature. A detailed description of the algorithm is shown in Figure 7.1.

The Paul algorithm begins by finding the closest codeword, $\mathbf{c}^*$, to the current source vector, $\mathbf{X}_t$, in the previous local codebook, $\mathcal{C}_{t-1}$, If the distance from the current source vector to this closest codeword is less than the distortion threshold, the index of this codeword is transmitted to the decoder. We also transmit a one-bit flag to the decoder as side information to signal that we are not updating the local codebook. We then set $\mathcal{C}_t = \mathcal{C}_{t-1}$ and repeat the process for the next source vector.

If the distortion between the closest codeword and the current source vector is greater than the distortion threshold, we update the local codebook. In this case,

Given: distortion threshold, $D_{\max}$
initial local codebook, $\mathcal{C}_0$
initial time, $t = 1$

Step 1: Find the closest codeword, $\mathbf{c}^*$, in the previous local codebook, $\mathcal{C}_{t-1}$:

$$\mathbf{c}^* = \arg \min_{\mathbf{c} \in \mathcal{C}_{t-1}} d(\mathbf{X}_t, \mathbf{c}).$$

Step 2: If $d(\mathbf{X}_t, \mathbf{c}^*) \leq D_{\max}$, go to Step 3; else go to Step 4.

Step 3: Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. Transmit the index of codeword $\mathbf{c}^*$ to the decoder. Transmit flag 0 to the decoder as side information. Go to Step 5.

Step 4: Let $\mathcal{C}_t = \mathcal{C}_{t-1}$. Find the LRU codeword of $\mathcal{C}_t$. Transmit the index of this LRU codeword to the decoder. Replace the LRU codeword in $\mathcal{C}_t$ with the current source vector $\mathbf{X}_t$. Transmit flag 1 and $\mathbf{X}_t$ to the decoder as side information.

Step 5: Set $t = t + 1$ and go to Step 1.

Figure 7.1: The Paul algorithm [52]

we form $\mathcal{C}_t$ by replacing the LRU codeword of $\mathcal{C}_{t-1}$ with the current source vector. As side information, we transmit the entire new codeword to the decoder, as well as a one-bit flag indicating that we are updating the codebook. We also transmit the index of the LRU codeword to the decoder, so as to specify to the decoder where in the codebook to place the new codeword.

We now make several remarks regarding the Paul algorithm with respect to the AVQ communication-system model of Figure 6.3 in Chapter 6. Regarding the codebook coder, notice that the Paul algorithm can potentially update the local codebook for each source vector. Consequently, it can adapt very quickly to abrupt changes in the source statistics. However, the amount of side information transmitted can be quite substantial, as we must send the entire new codeword to the decoder for each update. Since the new codewords are vectors appearing in the source data, the universal codebook for the Paul algorithm is the entire source alphabet, $\mathcal{X}^N$. Consequently, we must use a lossy codebook coder as discussed in Section 6.2.3. The original Paul algorithm was designed for speech-spectrum data and used a perceptually-weighted technique to quantize each codeword to about 50 bits [52]. Goodman *et al.* [54] have modified Paul's original codebook coder, introducing techniques appropriate for image-data vectors. For reasons discussed previously, we will not consider the details of these *ad hoc* codebook coders further here. Rather, when we empirically evaluate the Paul algorithm in Chapter 9, we will use a uniform, high-resolution scalar quantizer for the codebook coder and neglect any distortion due to codebook mismatch.

The original Paul algorithm used a fixed-length index coder. A more efficient method would use a variable-length index coder to achieve a coding closer to the first-order entropy of the index process. Such a variable-length index coder could very well be a static-mapped index coder, but other techniques of only marginal additional complexity can yield better practical performance. For example, Goodman *et al.* [54] present a variation of the Paul algorithm that uses a move-to-front technique to "rearrange" the local codebook after each codebook update.

The *move-to-front* algorithm [55] is a heuristic that has been shown to aid lossless coding techniques in achieving a lower rate. The move-to-front algorithm involves rearranging the index assignments of the codewords of $\mathcal{C}_t$ during coding. Equivalently, one can view this index reassignment as a shuffling of the actual positions of the codewords within the codebook, hence the name "move-to-front." In the modification to the Paul algorithm presented by Goodman *et al.* [54], the move-to-front algorithm operates as follows. After the index of the closest codeword is transmitted to the

Given: distortion threshold, $D_{\max}$
initial local codebook, $\mathcal{C}_0$
initial time, $t = 1$

Step 1: Find the closest codeword, $\mathbf{c}^*$, in the previous local codebook, $\mathcal{C}_{t-1}$:

$$\mathbf{c}^* = \arg \min_{\mathbf{c} \in \mathcal{C}_{t-1}} d(\mathbf{X}_t, \mathbf{c}).$$

Step 2: If $d(\mathbf{X}_t, \mathbf{c}^*) \leq D_{\max}$, go to Step 3; else go to Step 4.

Step 3: Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. Transmit the index of codeword $\mathbf{c}^*$ to the decoder. Transmit flag 0 to the decoder as side information. Move $\mathbf{c}^*$ to the front of $\mathcal{C}_t$. Go to Step 5.

Step 4: Let $\mathcal{C}_t = \mathcal{C}_{t-1}$. Insert current source vector $\mathbf{X}_t$ into the front of $\mathcal{C}_t$. Increment the index number of all the other codewords of $\mathcal{C}_t$. Delete the codeword with the highest index. Transmit flag 1 and $\mathbf{X}_t$ to the decoder as side information.

Step 5: Set $t = t + 1$ and go to Step 1.

Figure 7.2: The move-to-front variation of the Paul algorithm [52, 54]

decoder, the codeword is moved to the front (index number 0) of the local codebook in both the encoder and the decoder. Consequently, codewords used frequently tend to be assigned small index values. A variable-length entropy coder designed on this sequence of index values can perform substantially better than static Huffman coding designed on the sequence of indices resulting from the use of the original, unshuffled codebook [55]. In the algorithm presented by Goodman *et al.* [54], when a codeword is added to the codebook, it is added to the first slot (index number 0) of the local codebook and the indices of each of the remaining codewords are incremented. The last codeword in the codebook is deleted. This scheme inherently implements a LRU forgetting algorithm; it has the additional advantage that, since both the encoder and decoder know where to put the new codeword, transmission of an index specifying the point of insertion of the new codeword is not needed. Because of these advantages, we will use a move-to-front index coder when we evaluate the performance of the Paul algorithm in Chapter 9. This move-to-front variation of the Paul algorithm is shown in Figure 7.2.

Despite the relative simplicity of the Paul algorithm in comparison to other AVQ algorithms, it does perform quite well in practice, as we will see in Chapter 9. The Paul algorithm ensures that each source vector is coded with a distortion which is, at most, the value of distortion threshold. When the Paul algorithm updates the local codebook, it is "greedy;" that is, when it expends bits for a codebook update, it gets the best return, in terms of distortion, for that expense. Next, we examine the Wang-Shende-Sayood [48] algorithm, which is similar to the move-to-front variation of the Paul algorithm.

7.1.2 The Wang-Shende-Sayood Algorithm

The algorithm developed by Wang, Shende, and Sayood [48] and detailed in Figure 7.3 is an implicit threshold-replenishment algorithm very similar to the Paul algorithm. The main difference between the two algorithms occurs in the implementation of the universal codebook. Wang *et al.* use an infinite collection of lattice quantizer points distributed across Euclidean space $\Re^N$ (for more information on lattice quantizers, refer to [2, 56, 57]). The lattice quantizer is chosen so that the maximum distance between any two lattice points is at most D_{max}, the distortion-threshold parameter to the algorithm. As shown by Conway and Sloane [56], there exist fast algorithms that yield the closest lattice point to a given source vector for the lattices commonly used as vector quantizers. In the Wang-Shende-Sayood algorithm,

Given: distortion threshold, $D_{\max}$
lattice implementing the universal codebook, $\mathcal{C}^*$
initial local codebook, $\mathcal{C}_0 \subset \mathcal{C}^*$
initial time, $t = 1$

Step 1: Find the closest codeword, $\mathbf{c}^*$, in the previous local codebook, $\mathcal{C}_{t-1}$:

$$\mathbf{c}^* = \arg \min_{\mathbf{c} \in \mathcal{C}_{t-1}} d(\mathbf{X}_t, \mathbf{c}).$$

Step 2: If $d(\mathbf{X}_t, \mathbf{c}^*) \leq D_{\max}$, go to Step 3; else go to Step 4.

Step 3: Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. Transmit the index of codeword $\mathbf{c}^*$ to the decoder. Transmit flag 0 to the decoder as side information. Move $\mathbf{c}^*$ to the front of the local codebook. Go to Step 5.

Step 4: Find the codeword in the universal codebook that is closest to the source vector:

$$\mathbf{v}^* = \arg \min_{\mathbf{v} \in \mathcal{C}^*} d(\mathbf{X}_t, \mathbf{v}).$$

Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. Insert $\mathbf{v}^*$ into the front of $\mathcal{C}_t$. Increment the index number of all the other codewords of $\mathcal{C}_t$. Delete the codeword with the highest index. Transmit flag 1 and $\mathbf{v}^*$ to the decoder as side information.

Step 5: Set $t = t + 1$ and go to Step 1.

Figure 7.3: The Wang-Shende-Sayood algorithm [48]

the universal codebook can be considered to be implicitly specified by one of these lattice algorithms. Wang, Shende, and Sayood do not specify a particular lattice to be used with their algorithm. Later, in Chapter 9, we evaluate the performance of the Wang-Shende-Sayood algorithm using the D_N lattice [57] because of its ease of implementation. The consideration of more sophisticated lattices is beyond the scope of this work.

Similar to the move-to-front variation of the Paul algorithm, the Wang-Shende-Sayood algorithm also uses the move-to-front heuristic in the index coder. Wang *et al.* originally reported with their algorithm a lossy codebook-coder method designed to lower the amount of side information. In their codebook-coder scheme, instead of sending the entire new codeword, Wang *et al.* update only a subset of the codeword components. We, however, assume an idealized scalar-quantizer codebook coder for the reasons previously discussed.

We now consider the second category of AVQ algorithms, the constrained-rate algorithms. Contrary to the constrained-distortion algorithms considered thus far, the constrained-rate algorithms place a limit on the rate of the algorithm rather than on the distortion.

7.2 Constrained-Rate AVQ Algorithms

The class of constrained-rate AVQ algorithms has enjoyed the most attention in AVQ literature. Indeed, most of the algorithms reported fall into this category. We briefly discuss of a number of them in this section, and present the details of a few of the algorithms in Sections 7.2.2 through 7.2.6.

The main idea behind constrained-rate AVQ is to restrict the rate of the algorithm to be less than a certain maximum, and then concentrate on reducing the distortion as much as possible. The most common ways to constrain the rate are to set some limits on the number of codewords that the codebook selector can update at a given time and to restrict these updates to occur at certain time intervals.

For example, a popular paradigm for constrained-rate AVQ algorithms is as follows. Suppose we have a universal codebook of size K^* from which the codebook selector produces a new local codebook of size K after the coding of τ source vectors. That is, codebook updating occurs spaced in regular intervals of τ source vectors. Thus, we have an *adaption interval* of τ vectors. The average code length, as defined in Equation 6.19, is

$$\tilde{L}(W_t) = \frac{1}{NT}\sum_{t=1}^{T} l(\gamma_t(\alpha_t(\mathbf{X}_t))) \leq \frac{1}{N}\left\lceil \log_2 K \right\rceil, \tag{7.3}$$

for a finite process length of T vectors. Additionally, the average codebook code length, as defined in Equation 6.27, is

$$\tilde{L}(\tilde{S}_t) = \frac{1}{NT}\sum_{t=1}^{T} l(\tilde{S}_t) \leq \frac{1}{\tau N}\left\lceil \log_2 \binom{K^*}{K} \right\rceil. \tag{7.4}$$

The operational rate of this AVQ system then satisfies

$$\tilde{L}(Q_t) = \tilde{L}(W_t) + \tilde{L}(\tilde{S}_t) \leq \frac{1}{N}\left(\left\lceil \log_2 K \right\rceil + \frac{1}{\tau}\left\lceil \log_2 \binom{K^*}{K} \right\rceil \right). \tag{7.5}$$

The quantity on the right-hand side of Equation 7.5 is defined to be $R_{\max}$. In this type of constrained-rate AVQ system, the operational rate is constrained to be at most $R_{\max}$, with the operational rate equaling $R_{\max}$ when both the index coder and the codebook coder use fixed-length codes. The use of variable-length codes in either component will, in general, lower the operational rate below $R_{\max}$. Other constrained-rate AVQ algorithms feature a similar constraint on the rate.

There are two types of constrained-rate AVQ algorithms, local-context algorithms and codebook-retraining algorithms. We survey these two categories of algorithms below in Section 7.2.1 and 7.2.3, respectively.

7.2.1 Local-Context Algorithms

Local-context algorithms assume that the source possesses a certain degree of local stationarity. Consequently, local-context algorithms base the codebook updates on calculations performed over a certain length of time over which the source statistics are expected to be approximately stationary. This time interval is called the *adaption interval*. The exact length of time for which the local-stationarity assumption can be considered valid obviously varies from source to source and cannot be expected to remain constant for all time for a particular source. However, for simplicity, local-context algorithms generally fix the length of the adaption interval in advance and do not change it during coding.

The general operation of local-context algorithms is as follows. A buffer is filled with all the vectors from the current adaption interval. The codebook selector calculates certain measurements on the vectors of the adaption interval and then uses these measurements to select a local codebook from the universal codebook. This local codebook is then used by the vector coder to code one or more source vectors from the current adaption interval. The algorithm then repeats the process for the next adaption interval. Note that, in general, consecutive adaption intervals may overlap; however, most local-context algorithms use nonoverlapping adaption intervals.

Gersho and Gray survey a number of local-context AVQ algorithms in Chapter 16 of their VQ book [2]. Among these methods, the mean-adaption, gain-adaption, switched-codebook-adaption, and vector-excitation-coding algorithms can each be viewed as variants of the local-context technique in which the local codebook is determined and then used to code all the source vectors of the current adaption interval. The adaption intervals for these algorithms are nonoverlapping. We consider the gain-adaption algorithm in detail in Section 7.2.2 as an example of the class of local-context algorithms.

Other local-context algorithms are those reported by Nasrabadi and Feng [44] and Chang *et al.* [58]. In the algorithm by Nasrabadi and Feng [44], for the current source vector, a probabilistic score is calculated based on the conditional probabilities of the indices of previously encoded vectors. The universal codebook, which is trained in advance with the generalized Lloyd algorithm, is sorted using this probabilistic score. The local codebook is the first K of these sorted codewords. The current source vector is coded with this local codebook, and then the process repeats for the next source vector. The algorithm due to Chang, Chen, and Wang [58] features a universal codebook that is organized in a tree structure. The current local codebook is chosen as a subtree of the universal codebook based upon past usage statistics accumulated at the nodes of the tree of the universal codebook. Both the Nasrabadi-Feng and Chang-Chen-Wang algorithms use overlapping adaption intervals. Additionally, since the local-context statistics are accumulated based on the coded past of the source rather than the true past of the source, side information need not be sent to the decoder.

Given our mathematical definition of AVQ of Chapter 6, two other algorithms can be technically considered to be local-context AVQ algorithms. However, in the literature, finite-state VQ (Chapter 14 of [2]) and predictive VQ (Chapter 13 of [2]) are usually viewed as classes of VQ algorithms separate from adaptive VQ [2]. Consequently, we do not consider them further here.

In the next section, we describe the gain-adaption algorithm in detail. Most other local-context algorithms are similar to the gain-adaption algorithm, the main difference usually being only the type of calculation (gain, mean, etc.) that governs the local-codebook selection. Hence, the gain-adaption algorithm is considered in detail as an example of the entire class of local-context algorithms. Afterwards, in Section 7.2.3, we continue our discussion with the second class of constrained-rate AVQ algorithms, the codebook-retraining algorithms.

7.2.2 The Gain-Adaption Algorithm

Gain adaption [2] is a local-context AVQ algorithm that compensates for variations in the short-term power of nonstationary sources. Power variations are typical of many real-world signals that possess a wide dynamic range, such as speech and audio. It is assumed that such signals possess short, stationary contexts at one power level. Further, it is assumed that all the stationary contexts possess identical statistics, differing only in the short-term power of the signal. A gain-adaptive system, as shown in Figure 7.4, consists of a gain estimator which computes the local gain over a short window of source vectors which are stored in a frame buffer. In general, gain adaption can be based on vectors in the past or in the future relative to the current source vector. In forward gain adaption, which is the algorithm shown in Figure 7.4, the frame buffer stores a short sequence of source vectors which, following the gain computation and quantization, are "normalized" by the gain. These "normalized" vectors are then passed to a fixed vector quantizer. A detailed description of this algorithm is shown in Figure 7.5.

It appears initially that, since the vector quantizer is fixed, the gain-adaption algorithm possesses a single local codebook rather than a universal codebook. However,

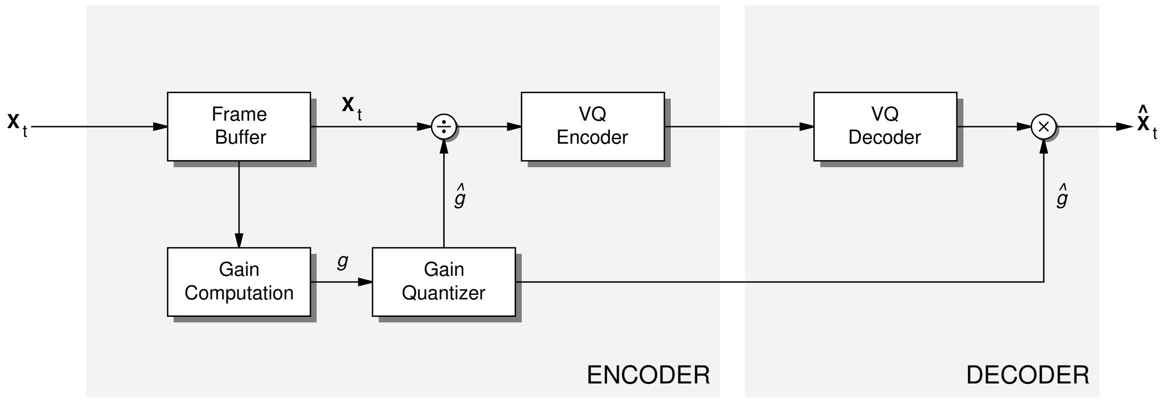


Figure 7.4: Block diagram of the gain-adaption algorithm [2]

further investigation reveals the contrary. Suppose we create a new algorithm as follows. We construct a universal codebook by multiplying each of the codewords of the fixed codebook of the gain-adaption algorithm by all the possible quantized gain values. The quantized gain then becomes the codebook-selection mechanism that selects the appropriate local codebook from this universal codebook. The buffered source vectors are no longer "normalized," but coded directly with the local codebook. This "new" algorithm functions identically to the gain-adaption algorithm and is merely an alternate definition of gain adaption consistent with the AVQ communication-system model of Chapter 6. Many of the local-context techniques presented in by Gersho and Gray [2], such as the mean-adaption and switched-codebook-adaption algorithms, also possess a structure similar to that of gain adaption that, at first sight, obscures the fact that they can be considered to have a universal codebook.

The gain-adaption algorithm presented here can be considered to be representative of the local-context algorithms. In the next section, we consider the second class of

Given: adaption-interval length, τ
fixed (nonadaptive) VQ codebook, $\mathcal{C}$
scalar gain quantizer, $q(\cdot)$
adaption-interval counter, $m = 1$

Step 1: Define set $\mathcal{T}_m$ as the current adaption interval m,

$$\mathcal{T}_m = \{\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots, \mathbf{X}_{t+\tau-1}\},$$

where $t = \tau(m-1)+1$.

Step 2: Calculate the average gain of the adaption-interval vectors:

$$g = \frac{1}{\tau} \sum_{\mathbf{X} \in \mathcal{T}_m} |\mathbf{X}|^2,$$

assuming that the process has zero mean.

Step 3: Scalar quantize the gain,

$$\hat{g} = q(g),$$

and transmit $\hat{g}$ to the decoder as side information.

Step 4: For each vector $\mathbf{X} \in \mathcal{T}_m$, do:

Step 4a: Normalize $\mathbf{X}$ with $\hat{g}$ and vector quantize; i.e., find $\mathbf{c}^*$:

$$\mathbf{c}^* = \arg\min_{\mathbf{c} \in \mathcal{C}} d(\mathbf{X}/\hat{g}, \mathbf{c}).$$

Step 4b: Send the index of c^* to the decoder.

Step 5: Set $m = m+1$ and go to Step 1.

Figure 7.5: The gain-adaption algorithm [2]

constrained-rate AVQ algorithms, the codebook-retraining algorithms. We present detailed descriptions of a few of these codebook-retraining algorithms in Section 7.2.4 through 7.2.5.

7.2.3 Codebook-Retraining Algorithms

Most of the AVQ algorithms reported in prior literature fall into the category of codebook-retraining algorithms. These algorithms are generally attempts to modify standard, nonadaptive VQ-design techniques to suit an environment with changing statistics. We discussed various VQ-design methods in Section 5.3. The AVQ algorithms discussed here use variations of either the generalized Lloyd algorithm (Section 5.3.1) or Kohonen learning (Section 5.3.2) and consequently employ distortion-based nearest-neighbor vector coders. AVQ algorithms derived from ECVQ (Section 5.3.3) fall into the category of rate-distortion-based AVQ, to be discussed in Section 7.3.

There are two types of codebook-retraining AVQ algorithms, batch and online. In general, *batch* algorithms extract a *local training set* of vectors from past or future vectors relative to the current source vector. Then a training algorithm, often quite computationally intense, is run, generally over many iterations through the local training set, to produce a new codebook. After the new codebook is generated, some or all of the new codewords replace old codewords in the previous local codebook to generate the new local codebook. Usually, the codebook-retraining process takes place at regular intervals during the coding of the source. For example, AVQ algorithms designed for image sequences often design a new codebook for each image of the sequence and use the new codebook to code that image; these algorithms are commonly called *frame adaptive* (e.g., [59]).

Because batch AVQ algorithms generally require substantial buffering and large amounts of computation for each new codebook produced, there has recently been increasing interest in the development of *online* codebook-retraining AVQ algorithms. A codebook-training algorithm is considered to be an online algorithm if the codewords are updated concurrently with the coding without any buffering or iterative computation. The distinction between online and batch algorithms can be somewhat vague, as subtle details of an algorithm may determine whether it is classed as one or the other. Artificial-neural-network algorithms, such as variations of Kohonen learning [34], are one common method for implementing online codebook-retraining AVQ algorithms.

The composition of the universal codebook for both batch and online algorithms depends on the particular method of VQ training used by the algorithm. Oftentimes, if the training algorithms can potentially produce any vector, the universal codebook will be simply the entire Euclidean space $\Re^N$. Also possible are training algorithms that are restricted to vectors meeting some criterion, such as that of lying on a lattice. In these algorithms, the universal codebook is then defined as the set of all vectors meeting the criteria.

Many of the batch codebook-retraining algorithms are based on the generalized Lloyd algorithm. The algorithm by Gersho and Yano [60], which is presented in greater detail in Section 7.2.4, monitors both the distortion and probability-of-use of each codeword and splits the codeword with the highest product of these two quantities. The generalized Lloyd algorithm is run on both of the resulting codewords. In algorithms by Goldberg and Sun [61, 59], one or more iterations of the generalized Lloyd algorithm are run over a local training set. Codewords in the trained codebook that have changed significantly from their counterparts in the previous local codebook are added to the codebook. We discuss the algorithms by Goldberg and Sun in greater detail in Section 7.2.5.

Some batch codebook-retraining algorithms feature neural-network techniques. For example, the algorithm due to Lancini, Perego, and Tubaro [49], uses competitive learning, a variation of Kohonen learning, to periodically train a new codebook, with significant new codewords being added to the local codebook. We discuss the Lancini-Perego-Tubaro algorithm more thoroughly in Section 7.2.6.

Other batch algorithms based on neural networks include algorithms presented by Fang *et al.* [62] and Chen *et al.* [63]. These algorithms use a variation of Kohonen learning in which the learning rule is modified based on how frequently a codeword has been updated.

Finally, Lee and Peterson [64] present an online codebook-retraining algorithm that is a variation of Kohonen learning in which a lattice of codewords is allowed to grow or shrink based on changes in codeword distortion and codeword probabilities induced by changing source statistics. The Lee-Peterson algorithm unfortunately requires the use of a large amount of side information which limits its use in practice.

We discuss the Gersho-Yano algorithm, the Goldberg-Sun algorithms, and the Lancini-Perego-Tubaro algorithm next in Sections 7.2.4 through 7.2.6, respectively. Afterwards, in Section 7.3, we consider the final class of AVQ algorithms, the rate-distortion-based algorithms.

7.2.4 The Gersho-Yano Algorithm

One of the most well known AVQ algorithms is due to Gersho and Yano [60]. This algorithm falls into the category of batch codebook-retraining algorithms. The Gersho-Yano algorithm is based on certain theoretical results derived for the optimal nonadaptive VQ of a random vector.

The theoretical underpinnings of the Gersho-Yano algorithm are as follows [60]. Consider the nonadaptive VQ of a random vector, $\mathbf{X}$. As discussed in Section 5.1, the VQ codebook has K codewords with corresponding partition regions, $\mathcal{R}_i$. We define the partition probabilities as

$$p(i) = \operatorname{prob}\left\{\mathbf{X} \in \mathcal{R}_i\right\}, \tag{7.6}$$

and the partition distortions as

$$d(i) = E\left[d(\mathbf{X}, \hat{\mathbf{X}}) \mid \mathbf{X} \in \mathcal{R}_i\right]. \tag{7.7}$$

Then the partial distortions for each partition region are defined as

$$D(i) = p(i) \cdot d(i) \tag{7.8}$$

[60, 29]. Gersho [29] argues that, for an asymptotically large number of codewords, the partial distortions $D(i)$ for the optimal vector quantizer approach a constant independent of i. The Gersho-Yano AVQ algorithm [60] extends this result to the coding of random processes by attempting to maintain equal partial distortions over time.

Gersho and Yano describe two similar variations of the algorithm, a basic algorithm and a slightly more complicated algorithm oriented towards image coding. We consider only the former of these two algorithms for reasons of simplicity. This

Given: adaption-interval length, τ
initial local codebook, $\mathcal{C}_0$
adaption-interval counter, $m = 1$

Step 1: Define set $\mathcal{T}_m$ as the current adaption interval m,

$$\mathcal{T}_m = \{\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots, \mathbf{X}_{t+\tau-1}\},$$

where $t = \tau(m-1)+1$.

Step 2: Calculate the partial distortions:

Step 2a: Map each of vector of $\mathcal{T}_m$ to a partition region using the previous local codebook:

$$\mathcal{R}_i = \left\{ \mathbf{X} \in \mathcal{T}_m : \arg \min_{\mathbf{c} \in \mathcal{C}_{m-1}} d(\mathbf{X}, \mathbf{c}) = \mathbf{c}_i \right\},$$

where $\mathbf{c}_i \in \mathcal{C}_{m-1}$.

Step 2b: Calculate the partition probabilities, the partition distortions, and the partial distortions:

$$p(i) = \frac{|\mathcal{R}_i|}{\tau}, \qquad d(i) = \frac{1}{|\mathcal{R}_i|} \sum_{\mathbf{X} \in \mathcal{R}_i} d(\mathbf{X}, \mathbf{c}_i),$$
$$D(i) = p(i) \cdot d(i).$$

Step 3: Let $\mathbf{c}_{i\text{max}}$ and $\mathbf{c}_{i\text{min}}$ be the codewords corresponding to the partitions with the largest and smallest partial distortions, respectively. Split $\mathbf{c}_{i\text{max}}$ into two codewords, $\mathbf{c}_{i\text{max}1}$ and $\mathbf{c}_{i\text{max}2}$, by adding small random deviations to each codeword component.

Step 4: Run one iteration of the generalized Lloyd algorithm using $\mathcal{R}_{i\text{max}}$ as the training data and $\mathbf{c}_{i\text{max}1}$ and $\mathbf{c}_{i\text{max}2}$ as the codewords.

Step 5: Set $\mathcal{C}_m = \mathcal{C}_{m-1}$. Replace $\mathbf{c}_{i\text{max}} \in \mathcal{C}_m$ with $\mathbf{c}_{i\text{max}1}$ and $\mathbf{c}_{i\text{min}} \in \mathcal{C}_m$ with $\mathbf{c}_{i\text{max}2}$.

Step 6: Send $\mathbf{c}_{i\text{max}1}$, $\mathbf{c}_{i\text{max}2}$, and their codebook indices to the decoder as side information.

Step 7: Vector quantize the vectors of $\mathcal{T}_m$ with $\mathcal{C}_m$, set $m = m+1$, and go to Step 1.

Figure 7.6: The Gersho-Yano algorithm [60]

variation of the Gersho-Yano algorithm is described in Figure 7.6. The operation of the algorithm is as follows. The source process is partitioned into consecutive sets of vectors called adaption intervals. To code the current adaption interval, first the codebook selector calculates the partial distortions of the codebook using the vectors of the adaption interval. The codebook selector "splits" the codeword of the partition region with the largest partial distortion into two new codewords. The codebook selector runs one iteration of the generalized Lloyd training algorithm for those vectors of the current adaption interval that map into this split region; this training iteration refines the location of the two new codewords. These two new codewords enter the local codebook, replacing the old codeword that was split and the codeword that belongs to the region with the smallest partial distortion. The vector coder then codes all the vectors of current adaption interval with the new local codebook. As side information, the Gersho-Yano algorithm must send the two new codewords to the decoder, as well as their addresses in the new local codebook. The second, more complicated version of the Gersho-Yano algorithm operates similarly to the basic algorithm just outlined, the main difference being that a more complicated codebook selector performs multiple codeword splittings until the all the partial distortions are below a distortion threshold.

We discuss the Goldberg-Sun algorithms in the next section. These algorithms also make use of the generalized Lloyd algorithm for codebook training.

7.2.5 The Goldberg-Sun Algorithm

Goldberg and Sun [61, 59] describe several similar algorithms that use the generalized Lloyd algorithm to train a codebook from which one or more codewords are selected to be added to the local codebook. We describe here the simplest of these algorithms.

The operation of this simple variant of the Goldberg-Sun family of algorithms is described in Figure 7.7. The codebook selector runs one or more iterations of the generalized Lloyd algorithm on the vectors of the current adaption interval, producing a codebook with the same number of vectors as the previous local codebook. Note that the previous local codebook is used as a "seed" for the generalized Lloyd algorithm. The Euclidean distance between each of the codewords of the new codebook and their counterparts in the local codebook is calculated. If this distance is greater than a prespecified threshold for a certain new codeword, the codebook selector inserts the new codeword in the place of its counterpart in the local codebook. The side information for this algorithm then consists of a set of flags specifying whether each codeword is replaced or not, as well as the entire codeword for those replaced vectors. The vector coder used in this algorithm is the standard, distortion-based nearest-neighbor mapping.

The algorithm outlined here is a technique called "codebook replenishment by mean shift" by Goldberg and Sun [61]. The algorithms presented by Goldberg and Sun [61, 59] also feature other, more *ad hoc* techniques which we do not consider here.

Both of the codebook-retraining algorithms we have seen thus far use the generalized Lloyd algorithm for codebook training. Next, we examine a codebook-retraining algorithm that features a neural-network technique.

Given: adaption-interval length, τ
replenishment threshold, δ
initial local codebook, $\mathcal{C}_0$
adaption-interval counter, $m = 1$

Step 1: Define set $\mathcal{T}_m$ as the current adaption interval m,

$$\mathcal{T}_m = \{\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots, \mathbf{X}_{t+\tau-1}\},$$

where $t = \tau(m-1)+1$.

Step 2: Run one or more iterations of the generalized Lloyd algorithm. Use $\mathcal{T}_m$ as the training data and $\mathcal{C}_{m-1}$ as the initial codebook to the generalized Lloyd algorithm. Denote the new codebook produced as $\mathcal{C}'$.

Step 3: Set $\mathcal{C}_m = \mathcal{C}_{m-1}$. For each codeword $\mathbf{c}'_i \in \mathcal{C}'$ with corresponding codeword $\mathbf{c}_i \in \mathcal{C}_m$ do

Step 3a: If

$$|\mathbf{c}'_i - \mathbf{c}_i| > \delta,$$

replace $\mathbf{c}_i$ in $\mathcal{C}_m$ with $\mathbf{c}'_i$.

Step 4: Send, as side information, a one bit flag for each $\mathbf{c}_i \in \mathcal{C}_m$ indicating whether or not the codeword was updated. For each updated codeword, send the entire new codeword.

Step 5: Vector quantize the vectors of $\mathcal{T}_m$ with $\mathcal{C}_m$, set $m = m+1$, and go to Step 1.

Figure 7.7: The Goldberg-Sun algorithm [61]

7.2.6 The Lancini-Perego-Tubaro Algorithm

A batch codebook-retraining algorithm developed by Lancini, Perego, and Tubaro [49] features a variation of Kohonen learning [34] called competitive learning [36]. A detailed description of the Lancini-Perego-Tubaro algorithm is given in Figure 7.8.

In the Lancini-Perego-Tubaro algorithm, a random set of vectors are chosen from the current adaption interval. These vectors form an initial codebook given to a competitive-learning VQ-training algorithm that iterates over the vectors of the adaption interval until convergence. Then, for each codeword in the newly trained codebook, the closest codeword in the previous local codebook is found. The codebook selector replaces the K_R LRU codewords in the previous codebook with the K_R most distant new codewords to form the current local codebook. The vectors of the adaption interval are quantized using a distortion-based nearest-neighbor vector coder with this current local codebook. Note that the only side information that needs to be sent are the K_R most distant codewords as the decoder can keep track of the frequency-of-use statistics and, consequently, knows where to place the updated codewords in the new local codebook.

The algorithm for Kohonen learning was previously presented in Figure 5.4. Competitive learning [36] is a simplified variant of Kohonen learning in which the neighborhood function is 1 for the nearest-neighbor codeword and 0 for all others. See Section 5.3.2 for more details.

Up to now, we have considered two classes of AVQ algorithms, the constrained-distortion and the constrained-rate algorithms. In the next section, we present the final class of AVQ algorithms, the rate-distortion-based algorithms. We will see that these algorithms update the local codebook using both rate and distortion, so that

Given: adaption-interval length, τ
K_T, the number of new codewords to train
K_R, the number of vectors to replace
initial local codebook, $\mathcal{C}_0$
adaption-interval counter,$m = 1$

Step 1: Define set $\mathcal{T}_m$ as the current adaption interval m,

$$\mathcal{T}_m = \{\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots, \mathbf{X}_{t+\tau-1}\},$$

where $t = \tau(m-1) + 1$.

Step 2: Randomly choose K_T vectors from $\mathcal{T}_m$ to serve as an initial codebook, $\mathcal{C}_T$, for training.

Step 3: Train a new codebook, $\mathcal{C}'$, using a competitive-learning algorithm with $\mathcal{C}_T$ as the initial codebook and $\mathcal{T}_m$ as the training data.

Step 4: For each codeword $\mathbf{c}'_i \in \mathcal{C}'$ do

Step 4a: Find the closest codeword, $\mathbf{c}^* \in \mathcal{C}_{m-1}$,

$$\mathbf{c}^* = \arg \min_{\mathbf{c} \in \mathcal{C}_{m-1}} d(\mathbf{c}'_i, \mathbf{c}).$$

Step 4b: Find the distance to the closest codeword,

$$\delta_i = d(\mathbf{c}'_i, \mathbf{c}^*).$$

Step 5: Let $\mathcal{C}_m = \mathcal{C}_{m-1}$. Insert the K_R codewords $\mathbf{c}'$ with the largest distances δ into $\mathcal{C}_m$, replacing the K_R LRU codewords of $\mathcal{C}_m$.

Step 6: Vector quantize the vectors of $\mathcal{T}_m$ with $\mathcal{C}_m$, set $m = m + 1$, and go to Step 1.

Figure 7.8: The Lancini-Perego-Tubaro algorithm [49]

a codebook update is done only if it is favorable in the rate-distortion sense. This is in contrast to the algorithms presented thus far which choose to update the local codebook to keep either the rate or the distortion alone below a certain constraint without regard to the cost incurred in the opposite quantity.

7.3 Rate-Distortion-Based AVQ Algorithms

In the two classes of AVQ algorithms presented thus far, one quantity, either the rate or the distortion, is constrained to be less than some maximum value. The codebook selector performs updates to the local codebook in order to satisfy this constraint, regardless of what the update costs in terms of the other quantity. For example, in constrained-distortion AVQ, codebook updates are performed whenever it is necessary to add a codeword to keep the distortion below the distortion threshold. However, the codebook selector of a constrained-distortion algorithm does not attempt to evaluate whether the reduction in distortion due to the update is worth its cost in rate. Similarly, constrained-rate algorithms fail to weigh the frequency of updates allowed by the rate constraint against the distortion performance of the algorithm. Additionally, the vector coders of constrained-distortion and constrained-rate algorithms implement nearest-neighbor mappings based solely on distortion. Although the rate-distortion theory of Chapter 4 justifies the approaches taken by these two classes of algorithms for stationary processes, the third class of AVQ algorithms that we present in this section is capable of better performance in the more realistic and practical case of nonstationary sources.

This new class of algorithms, rate-distortion-based AVQ, incorporates both the rate and the distortion into a cost function. The minimization of this cost function results in both quantities being considered in both the codebook selector and the vector coder. Consequently, the codebook selector of a rate-distortion-based algorithm weighs the potential for a reduction in distortion against the cost in rate for each codeword update, while the vector coder selects the codeword "closest" in the rate-distortion sense.

The only rate-distortion-based AVQ algorithm to appear in previous literature is the algorithm by Lightstone and Mitra [46, 51], although we present a new algorithm of this class in Chapter 8. The Lightstone-Mitra algorithm is based on the ECVQ method of [30] nonadaptive VQ design which was discussed in Section 5.3.3.

Recall that the ECVQ algorithm features a cost function of the form

$$J(Q) = \overline{D}(Q) + \lambda \overline{L}(Q), \tag{7.9}$$

where the rate-distortion parameter λ controls the relative weighting of the rate against the distortion in the cost function. The codebook selector of the Lightstone-Mitra algorithm uses ECVQ to generate new codewords based on the minimization of Equation 7.9. Additionally, the codebook selector uses a cost evaluation similar to Equation 7.9 to determine whether the new codewords generated with ECVQ should be allowed to update the codebook. Finally, we note that the Lightstone-Mitra algorithms employs a vector coder implementing a modified nearest-neighbor rule, which, like that of ECVQ, incorporates both rate and distortion.

A detailed description of the Lightstone-Mitra algorithm is provided in Figure 7.9. The operation of the algorithm is as follows. The codebook selector first runs the ECVQ training algorithm on the vectors of the adaption interval. The previous local codebook, $\mathcal{C}_{m-1}$, is used as the initial codebook to the ECVQ algorithm. Additionally, because the ECVQ algorithm assumes the use of a variable-length index coder,

Given: adaption-interval length, τ
rate-distortion parameter, λ
initial local codebook, $\mathcal{C}_0$
initial codeword probabilities, $p_0(i)$, for each codeword $\mathbf{c}_i \in \mathcal{C}_0$
adaption-interval counter, $m = 1$

Step 1: Define set $\mathcal{T}_m$ as the current adaption interval m,

$$\mathcal{T}_m = \{\mathbf{X}_t, \mathbf{X}_{t+1}, \ldots, \mathbf{X}_{t+\tau-1}\},$$

where $t = \tau(m-1)+1$.

Step 2: Run one iteration of ECVQ. Let $\mathcal{T}_m$ be the training set, $\mathcal{C}_{m-1}$ be the initial codebook, $l(\gamma_{m-1}(\mathbf{c}_i)) = -\log_2 p_{m-1}(i)$ be the initial codeword lengths, and λ be the rate-distortion parameter to the ECVQ algorithm. The ECVQ algorithm returns new codebook $\mathcal{C}'$ with new probabilities $p'(i)$ for each $\mathbf{c}'_i \in \mathcal{C}'$.

Step 3: Set $\mathcal{C}_m = \mathcal{C}_{m-1}$ and $p_m(i) = p'(i)$ for all i. Determine $\mathcal{R}_i$ for each $\mathbf{c}_i \in \mathcal{C}_m$:

$$\mathcal{R}_i = \left\{ \mathbf{X} \in \mathcal{T}_m : \arg\min_{\mathbf{c} \in \mathcal{C}_m} [d(\mathbf{X}, \mathbf{c}) + \lambda \cdot l(\gamma_m(\mathbf{c}))] = \mathbf{c}_i \right\}.$$

Step 4: For each codeword $\mathbf{c}_i \in \mathcal{C}_m$ with $p_m(i) \neq 0$ and corresponding codeword $\mathbf{c}'_i \in \mathcal{C}'$ do:

Step 4a: Calculate the expected distortions for $\mathbf{c}'_i$ and $\mathbf{c}_i$ and their difference:

$$d_1 = \frac{1}{|\mathcal{R}_i|} \sum_{\mathbf{X} \in \mathcal{R}_i} d(\mathbf{X}, \mathbf{c}'_i), \qquad d_2 = \frac{1}{|\mathcal{R}_i|} \sum_{\mathbf{X} \in \mathcal{R}_i} d(\mathbf{X}, \mathbf{c}_i),$$

$$\Delta d = d_1 - d_2.$$

Step 4b: Calculate the cost of a codeword update:

$$\Delta r = l(\mathbf{c}'_i)/|\mathcal{R}_i|,$$

where $l(\mathbf{c}'_i)$ is the length in bits of side information needed to send $\mathbf{c}'_i$ to the decoder.

Step 4c: If $\Delta J = \Delta d + \lambda \cdot \Delta r < 0$, set $\mathbf{c}_i = \mathbf{c}'_i$. Send $\mathbf{c}'_i$ and the index i to the decoder as side information.

Step 5: For each $\mathbf{X} \in \mathcal{T}_m$, send index i to decoder, where $\mathbf{X} \in \mathcal{R}_i$. Set $m = m+1$ and go to Step 1.

Figure 7.9: The Lightstone-Mitra algorithm [46]

the codebook selector supplies the ECVQ algorithm with initial codeword lengths, which are calculated using the current estimates of the codeword probabilities. More specifically, if $\mathbf{c}_i$ is a codeword in $\mathcal{C}_{m-1}$, we denote the probability of occurrence of $\mathbf{c}_i$ as $p_{m-1}(i)$. For this codeword, we estimate the length in bits of the output of the variable-length index coder as

$$l(\gamma_{m-1}(\mathbf{c}_i)) = -\log_2 p_{m-1}(i). \tag{7.10}$$

Finally, we note that the rate-distortion parameter passed to the ECVQ algorithm is the same parameter λ of the Lightstone-Mitra algorithm.

The ECVQ algorithm returns a set of new codewords $\mathcal{C}'$ with new codeword probabilities, $p'(i)$. The codebook selector must now decide whether to add any of the new codewords $\mathbf{c}'_i \in \mathcal{C}'$ to the local codebook. For each $\mathbf{c}'_i$, the codebook selector calculates the expected distortions, d_1 and d_2, over the partition corresponding to $\mathbf{c}'_i$ for both cases in which the codebook is and is not updated. The difference, $\Delta d = d_1 - d_2$, in these two quantities is the expected reduction in distortion due to the codebook update. The codebook selector then calculates the rate cost, Δr, as the number of bits that the codebook coder needs to transmit $\mathbf{c}'_i$ to the decoder, distributed over each of the source vectors of the adaption interval that will benefit from the codebook update. Note that this rate cost represents the amount of side information incurred by a codebook update. The cost function, ΔJ, is formed as

$$\Delta J = \Delta d + \lambda \cdot \Delta r \tag{7.11}$$

using the parameter λ to the algorithm. If this cost function is less than 0 for a codeword, then the expected reduction in distortion due to updating the codebook with this codeword outweighs the rate cost associated with this codebook update, and the codebook selector proceeds to update the codebook. Finally, after all the codeword updates are completed, the vector coder codes each vector of the adaption interval using the rate-distortion-based nearest-neighbor rule.

The algorithm of Figure 7.9 represents the implementation of the Lightstone-Mitra algorithm that we use in Chapter 9 when we compare the performance of various AVQ algorithms. However, the algorithm of Figure 7.9 is somewhat less general than that originally described by Lightstone and Mitra [46, 51]. Lightstone and Mitra originally presented an algorithm with a more general codebook coder. However, for reasons previously discussed, we consider only a high-resolution scalar-quantization codebook coder. Additionally, Lightstone and Mitra originally described an algorithm allowing multiple iterations of Steps 2 through 4 in Figure 7.9, even though they note that usually one or two iterations suffice. We use only one iteration through these steps in our simplified variation of the Lightstone-Mitra algorithm in order to reduce the computational complexity of the algorithm to a level more comparable to the other AVQ algorithms under consideration.

We note one drawback to the Lightstone-Mitra algorithm as we have presented it. Recall that the ECVQ training algorithm generally reduces the size of the codebook during training. Consequently, the codebook $\mathcal{C}'$ produced by ECVQ generally contains empty regions where $p'(i) = 0$. The codewords in $\mathcal{C}_m$ corresponding to these empty regions are subsequently removed from any future consideration of being updated since empty regions are ignored by the ECVQ algorithm. Effectively, the part of the local codebook available to codeword updating continually gets smaller due to an accumulation of these "dormant" codewords. Unfortunately, this effect occurs only in one direction—the codebook selector of the Lightstone-Mitra algorithm cannot decide at a later time to start updating the codewords once they become dormant. Consequently, even though a number of dormant codewords may suffice in the initial stages of operation, the algorithm may suffer poor performance as the future statistics of the source change. Unfortunately, the original development [46, 51] of the Lightstone-Mitra algorithm lacks any method for the "reactivation" of dormant codewords.

The Lightstone-Mitra algorithm is the only algorithm in prior literature from the class of rate-distortion-based AVQ algorithms. Similar to the terminology introduced for constrained-rate algorithms, we consider the existence of two subclasses of rate-distortion-based algorithms, *batch* and *online* algorithms. The Lightstone-Mitra algorithm, which consists of a training process run over a set of training vectors (the adaption interval) is a batch algorithm. However, in Chapter 8, we present a new algorithm which is an online rate-distortion-based algorithm. Our algorithm, called generalized threshold replenishment, uses a cost function inspired by ECVQ, like the Lightstone-Mitra algorithm. However, unlike the Lightstone-Mitra algorithm, our generalized-threshold-replenishment algorithm resembles the Paul algorithm of Section 7.1.1 in that codebook updates can be performed for any source vector. We will the consider the details of our algorithm in Chapter 8. However, first we conclude this chapter in the next section by summarizing the discussion to this point with the presentation of a taxonomy of AVQ algorithms.

7.4 A Taxonomy of AVQ Algorithms

Figure 7.10 depicts a taxonomy including each of the algorithms discussed in this chapter. Additionally, we include the generalized-threshold-replenishment (GTR) algorithm, our new rate-distortion-based AVQ algorithm that we present next in Chapter 8.

In this chapter, we have presented three classes of AVQ algorithms, two of which have received significant attention in prior literature, and the third of which is a relatively recent innovation. We have detailed algorithms from each of the three classes. We will investigate the relative performance capabilities of these algorithms later in Chapter 9. First, however, we present a new AVQ algorithm which is a member of the third and most recent AVQ category and is the topic of the next chapter.

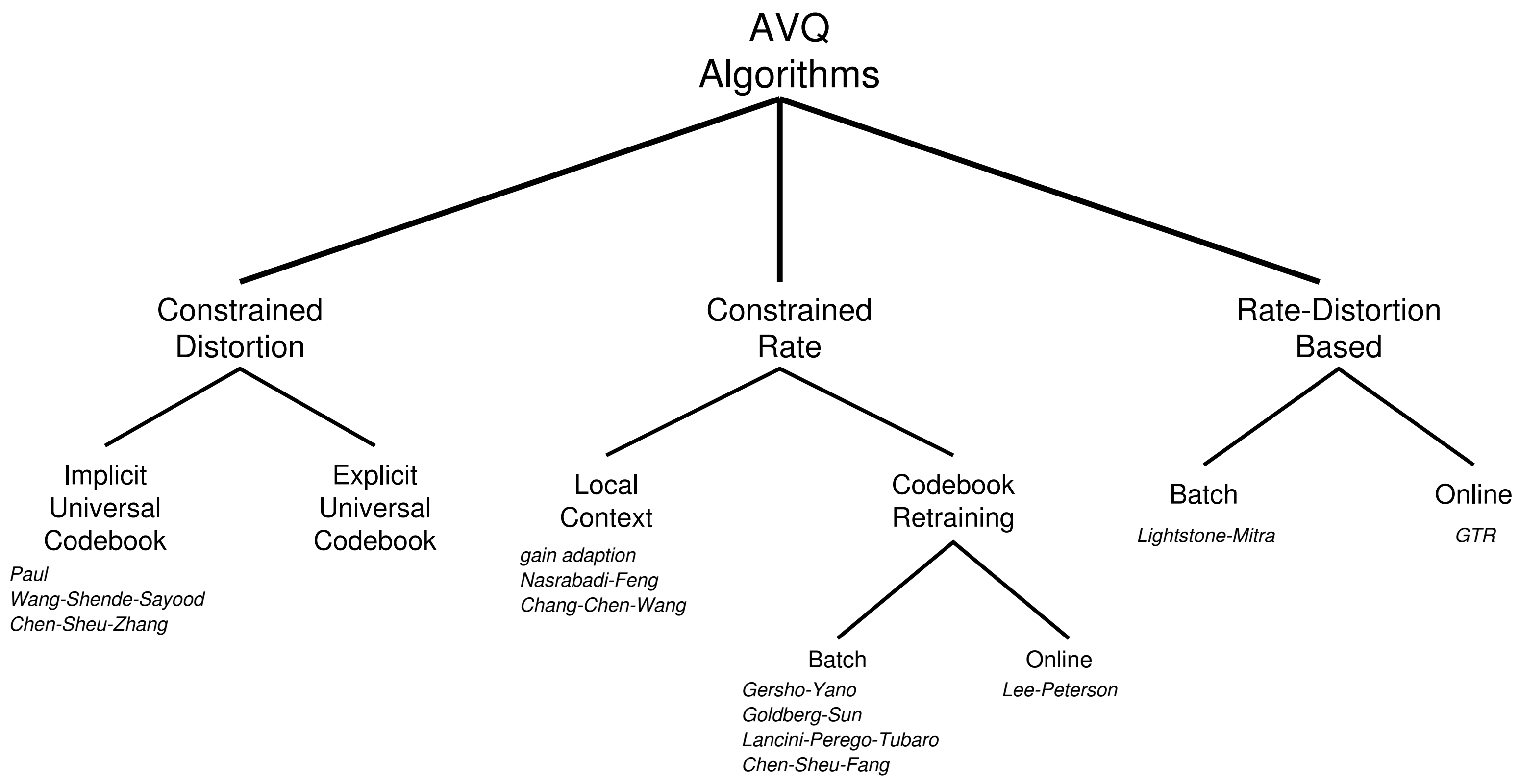


Figure 7.10: Taxonomy of AVQ algorithms. The classification of each of the algorithms discussed in Chapter 7 is indicated. The GTR algorithm refers to the generalized-threshold-replenishment algorithm presented in Chapter 8.

CHAPTER 8

THE GENERALIZED THRESHOLD REPLENISHMENT ALGORITHM

In this chapter, we present an online rate-distortion-based AVQ algorithm called generalized threshold replenishment (GTR). This algorithm has similarities to those developed by Lightstone and Mitra [46] and Paul [52]. In general terms, GTR incorporates a rate-distortion-based cost function similar to that of the Lightstone-Mitra algorithm into a framework similar to that of the Paul algorithm. However, unlike the Lightstone-Mitra algorithm, the GTR algorithm operates in an online manner rather than requiring large amounts of batch computation. That is, codebook updates, which can occur at any time, are based on quantities that are estimated dynamically. In addition, the GTR algorithm differs from the Paul algorithm in that the vector coder, as well as the codebook selector, are based on cost criteria involving both rate and distortion measures. The GTR algorithm weighs the distortion performance against the cost in rate in both the coding of the current source vector and the updating of the local codebook.

In this chapter, we describe the GTR algorithm using our AVQ communication-system model described previously in Section 6.2. To simplify the discussion, we present two versions of the GTR algorithm. In Section 8.1, we describe the basic

GTR algorithm which facilitates our discussion by laying the foundation for the move-to-front variant of the algorithm to follow in Section 8.2. Slightly more complicated than the basic algorithm, the move-to-front variant achieves a slight performance advantage in practice and, therefore, will be used extensively in Chapter 9 when we compare the performances of various AVQ algorithms. However, we begin our discussion of the GTR algorithm by considering the simpler variant first, in the next section.

8.1 The Basic Algorithm

In this section, we describe the basic GTR algorithm, which is outlined in Figure 8.1. This simple variant of GTR algorithm operates as follows.

The first sequence of steps of the algorithm determines the codeword "closest" to the current source vector in a rate-distortion sense. That is, the codebook selector finds the codeword in the local codebook that minimizes a cost function based on both the distortion between the current source vector and the codeword as well as the expected rate required to code the source vector with the codeword. More specifically, this rate-distortion-based nearest-neighbor mapping operates as follows. First, since the GTR algorithm assumes that a variable-length index coder is used, the estimated lengths of the variable-length codewords are calculated from the estimated probabilities, $p_{t-1}(i)$, of the codewords $\mathbf{c}_i \in \mathcal{C}_{t-1}$. That is, the variable-length codeword lengths are

$$l(\gamma_t(\mathbf{c}_i)) = -\log_2 p_{t-1}(i), \tag{8.1}$$

where $\gamma_t(\cdot)$ is the index coder of the AVQ system. Then, the distortion, $\delta(\mathbf{c}_i)$, between the current source vector, $\mathbf{X}_t$, and each codeword, $\mathbf{c}_i \in \mathcal{C}_{t-1}$, is calculated. These

Given: initial local codebook, $\mathcal{C}_0$
initial codeword probabilities, $p_0(i)$, for each codeword $\mathbf{c}_i \in \mathcal{C}_0$
rate-distortion parameter, λ
windowing parameter, ω
initial time, $t = 1$

Step 1: Calculate initial codeword lengths of the variable-length index coder:

$$l(\gamma_t(\mathbf{c}_i)) = -\log_2 p_{t-1}(i).$$

Step 2: Find the distortions between each codeword $\mathbf{c}_i \in \mathcal{C}_{t-1}$ and $\mathbf{X}_t$:

$$\delta(\mathbf{c}_i) = d(\mathbf{c}_i, \mathbf{X}_t).$$

Step 3: Calculate the cost function for each codeword

$$J(\mathbf{c}_i) = \delta(\mathbf{c}_i) + \lambda \cdot l(\gamma_t(\mathbf{c}_i)).$$

Step 4: Find the winning codeword:

$$\mathbf{c}^* = \arg \min_{\mathbf{c} \in \mathcal{C}_{t-1}} J(\mathbf{c}).$$

Let the index of $\mathbf{c}^*$ be denoted i^*.

Step 5: Calculate the distortion improvement and rate cost of a codebook update, as well as the rate-distortion cost function:

$$\Delta d = -\delta(\mathbf{c}^*), \qquad \Delta r = l(\mathbf{X}_t),$$
$$\Delta J = \Delta d + \lambda \cdot \Delta r.$$

Step 6: Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. If $\Delta J < 0$, go to Step 6a. Else, go to Step 6b.

Step 6a: Set $\mathbf{c}^* = \mathbf{X}_t$ in $\mathcal{C}_t$. Send to the decoder $\mathbf{X}_t$, index i^*, and a flag indicating a codebook update. Go to Step 7.

Step 6b: Send the index i^* and a flag indicating no codebook update.

Step 7: Estimate the new codeword probabilities:

$$p_t(i) = \begin{cases} [\omega p_{t-1}(i)] / (\omega + 1), & i \neq i^*, \\ [\omega p_{t-1}(i) + 1] / (\omega + 1), & i = i^*. \end{cases}$$

Step 8: Set $t = t + 1$ and go to Step 1.

Figure 8.1: The basic GTR algorithm

distortions and the variable-length codeword lengths are combined into a cost function using the rate-distortion parameter λ; i.e., the cost function, $J(\mathbf{c}_i)$, for each $\mathbf{c}_i \in \mathcal{C}_{t-1}$ is

$$J(\mathbf{c}_i) = \delta(\mathbf{c}_i) + \lambda \cdot l(\gamma_t(\mathbf{c}_i)). \tag{8.2}$$

The winning codeword, $\mathbf{c}^*$, is chosen to be the one with the lowest cost function. We denote the index of $\mathbf{c}^*$ be i^*. Note that the steps to this point implement an online version of the modified nearest-neighbor rule of the ECVQ training algorithm (Section 5.3.3).

In the next steps, the codebook selector decides whether the local codebook needs to be updated. First, the codebook selector estimates the improvement in distortion to be gained if the codebook is updated. If the codebook is updated, the current source vector will be coded with zero distortion. However, if the codebook is not updated, the current source vector will be coded with a distortion of $\delta(\mathbf{c}^*)$. Thus, the estimated distortion improvement due to a codebook update is

$$\Delta d \triangleq -\delta(\mathbf{c}^*). \tag{8.3}$$

Next, the codebook selector estimates the cost in rate, Δr, of the codebook update. This estimate is the amount of side information needed to send $\mathbf{X}_t$ to the decoder,

$$\Delta r \triangleq l(\mathbf{X}_t), \tag{8.4}$$

where $l(\mathbf{X}_t)$ is the number of bits used by the codebook coder to transmit $\mathbf{X}_t$ to the decoder.

The codebook selector uses the expected distortion improvement and the expected rate cost in a rate-distortion-based cost function, defined as

$$\Delta J \triangleq \Delta d + \lambda \cdot \Delta r, \tag{8.5}$$

where λ is the rate-distortion parameter given to the GTR algorithm. If $\Delta J < 0$, then the expected improvement in distortion outweighs the expected cost in rate, and the codebook selector inserts $\mathbf{X}_t$ into the local codebook by replacing $\mathbf{c}^*$ with $\mathbf{X}_t$. The codebook selector transmits to the decoder a flag indicating whether the codebook was updated. Additionally, if the codebook was updated, the codebook selector instructs the codebook coder to send $\mathbf{X}_t$ as side information.

The final step of the codebook selector is to estimate the new codeword probabilities using a time average. Our method of estimation assumes that the source possesses some degree of local stationarity; i.e., the statistics over a window of ω source vectors are approximately stationary. The codebook selector estimates how many of the past ω source vectors have mapped to codeword $\mathbf{c}_i \in \mathcal{C}_{t-1}$ as

$$n_{t-1}(i) = \omega \cdot p_{t-1}(i). \tag{8.6}$$

When the codebook selector has determined the winning codeword, $\mathbf{c}^*$, for the current source vector, the new counts are calculated as

$$n_t(i) = \begin{cases} n_{t-1}(i), & i \neq i^*, \\ n_{t-1}(i) + 1, & i = i^*. \end{cases} \tag{8.7}$$

The codebook selector estimates the new partition probabilities using the new codeword counts; i.e.,

$$p_t(i) = \frac{n_t(i)}{\sum_{\mathbf{c}_j \in \mathcal{C}_t} n_t(j)}, \tag{8.8}$$

for each $\mathbf{c}_i \in \mathcal{C}_t$. Plugging Equations 8.6 and 8.7 into Equation 8.7 yields

$$p_t(i) = \begin{cases} \left[\omega p_{t-1}(i)\right] / (\omega + 1), & i \neq i^*, \\ \left[\omega p_{t-1}(i) + 1\right] / (\omega + 1), & i = i^*. \end{cases} \tag{8.9}$$

After the codeword probabilities are calculated, the vector coder codes the current source vector, and the whole algorithm repeats for the next source vector.

The vector coder codes the current source vector using the new local codebook determined by the codebook selector. The vector coder operates identically to Steps 1 through 4 of Figure 8.1, calculating the nearest neighbor to the current source vector with a rate-distortion-based cost criterion. The vector coder transmits the index, i^*, of the winning codeword to the decoder. We note that the vector coder does not appear explicitly as a separate step in Figure 8.1 as its operation is included in the codebook selector (as Steps 1 through 4). That is, the codebook selector calculates the winning codeword using the rate-distortion-based nearest-neighbor mapping of the vector coder as the initial part of its operation. The index of this winning codeword does not change during a codebook update. Consequently, the implementation of the GTR algorithm depicted in Figure 8.1 does not need to include the vector coder as a separate step.

The parameter λ given to the GTR algorithm is called the *rate-distortion* parameter as it controls the tradeoff between rate and distortion within the algorithm. The value of λ affects not only the selection of the "closest" codeword by the vector coder but also the codebook-update decision in the codebook selector. Like ECVQ, GTR features a vector coder that chooses a codeword to represent the current source vector based on both the distortion between the two vectors as well as the rate needed to specify the codeword to the decoder. The parameter λ controls the relative weighting of these two quantities in this rate-distortion-based cost function. Additionally, λ controls the codebook selector by establishing the rate-distortion tradeoff in the codebook-update decision. Consequently, the value of λ ultimately determines the performance of the algorithm, in terms of rate and distortion. Indeed, as we will see in Chapter 9, varying the value of λ traces out the rate-distortion performance curve of the algorithm. Larger values of λ will focus the efforts of the algorithm on minimizing rate over distortion, whereas smaller values of λ will result in performance with a lower distortion and a higher rate.

As a final note, we observe that the value of the estimated cost in rate, Δr, depends on the codebook coder. In general, the codebook coder will allocate a variable number of bits to each $\mathbf{X}_t$ sent to the decoder. However, for reasons of simplicity discussed in Section 6.2.3, we consider only the case in which the codebook coder is a uniform, high-resolution scalar quantizer. In this case, Δr is a constant independent of $\mathbf{X}_t$, and the codebook-update decision rule (Equation 8.5) is identical to that of the Paul algorithm. However, even though the update decisions rules are the same in this case, codebook updates of the GTR algorithm are still performed in consideration of rate and distortion since the codeword to be updated, $\mathbf{c}^*$, is chosen with a rate-distortion-based vector coder rather than with a vector coder operating on distortion alone, as in the Paul algorithm.

8.2 The Move-to-Front Variant of the Algorithm

In the basic GTR algorithm presented in the previous section, when a codebook update was performed, the codebook selector added the current source vector to the local codebook by replacing the codeword that was determined to be the closest in a rate-distortion sense. In this section, we describe the move-to-front GTR algorithm. In this variant of the algorithm, codewords are added to the local codebook in a move-to-front fashion, effectively replacing the LRU codewords. The move-to-front GTR algorithm replaces Steps 6 and 7 of the basic algorithm (Figure 8.1) with the steps shown in Figure 8.2 and operates as follows.

Step 6: Set $\mathcal{C}_t = \mathcal{C}_{t-1}$. If $\Delta J < 0$, go to Step 6a. Else, go to Step 6b.

Step 6a: Insert $\mathbf{X}_t$ in the front of $\mathcal{C}_t$. Increment the indices of all the other codewords. Delete the codeword with the highest index. Send to the decoder $\mathbf{X}_t$ and a flag indicating a codebook update. Go to Step 7.

Step 6b: Send the index i^* and a flag indicating no codebook update. Move $\mathbf{c}^*$ to the front of $\mathcal{C}_t$.

Step 7: Estimate the new codeword probabilities. If $\Delta J \geq 0$, go to Step 7a. Else, go to Step 7b.

Step 7a: Estimate the new codeword probabilities as:

$$p_t(i) = \begin{cases} [\omega p_{t-1}(i)] / (\omega + 1), & i \neq i^*, \\ [\omega p_{t-1}(i) + 1] / (\omega + 1), & i = i^*. \end{cases}$$

Rearrange the indices of the probabilities to match the move-to-front rearrangement of $\mathcal{C}_t$; i.e., move $p_t(i^*)$ to index 1, shifting all the other probabilities. Go to Step 8.

Step 7b: Form the new codeword counts as:

$$n_t(i) = \begin{cases} \omega p_{t-1}(i), & i \neq i^* \\ \omega p_{t-1}(i)/2, & i = i^*. \end{cases}$$

Let K be the size of the local codebook; i.e., $K = |\mathcal{C}_t|$. Set $n_t(K) = n_t(i^*)$. Calculate the new codeword probabilities as:

$$p_t(i) = \frac{n_t(i)}{\sum_{1 \leq j \leq K} n_t(j)}.$$

Rearrange the indices of the probabilities to match the move-to-front rearrangement of $\mathcal{C}_t$; i.e., move $p_t(K)$ to index 1, shifting all the other probabilities.

Figure 8.2: The move-to-front GTR algorithm. These steps replace the corresponding steps in the basic GTR algorithm of Figure 8.1.

If the codebook is not updated, the codebook selector simply moves $\mathbf{c}^*$ to the front of the codebook. However, if the codebook selector updates the codebook, i.e., if $\Delta J < 0$, the codebook selector adds current source vector $\mathbf{X}_t$ to the local codebook. In this case, instead of replacing $\mathbf{c}^*$, the codebook selector places $\mathbf{X}_t$ in the front of $\mathcal{C}_t$. All the other codewords are shifted to the next highest index and the one with the highest index (the LRU codeword) is deleted.

The estimation of the new codeword probabilities is the same as in the basic algorithm, if the codebook was not updated. However, in the case of codebook update, the move-to-front insertion of $\mathbf{X}_t$ necessitates a modified calculation of the codeword probabilities. In this case, the codebook selector "splits" the probability of partition i^* between the new codeword, $\mathbf{X}_t$, and $\mathbf{c}^*$. Figure 8.2 gives the details of how this probability "split" is accomplished. After the new codeword probabilities are calculated, they are rearranged so as to match the move-to-front rearrangement of $\mathcal{C}_t$.

Like the move-to-front variant of the Paul algorithm (Figure 7.2 of Section 7.1.1), the move-to-front GTR algorithm effectively implements a LRU-replacement strategy with the added advantage that the index of the LRU codeword does not need to be transmitted to the decoder. As we will see in Chapter 9, the move-to-front variant of the GTR algorithm provides a slight improvement in rate-distortion performance over the basic algorithm, although the additional computational cost is negligible. Consequently, we will use the move-to-front variant when we compare the performance of the GTR algorithm with that of the AVQ algorithms presented in Chapter 7. This performance evaluation is the topic of the next chapter.

CHAPTER 9

EXPERIMENTAL RESULTS

In this chapter, we compare the performance of the AVQ algorithms presented in Chapters 7 and 8. We will consider two sets of sources, a nonstationary Wiener process and an image sequence. For each, we evaluate the performance of the AVQ algorithms for a range of different rates and distortions. In the case of the Wiener process, we compare the performance of the algorithms to the theoretical rate-distortion function given in Section 4.5.2.

To evaluate the rate-distortion performance of an AVQ algorithm, we use the operational rate and distortion measures introduced in Equations 6.35 and 6.33 of Section 6.2.4. The operational distortion is simply the time average of the distortion between the original source process and the coded output. In this chapter, we use the mean squared error (MSE) as the operational distortion measure.

The operational rate used is the sum of the time-average rates of the codings produced by the index coder and the codebook coder. As discussed in Section 6.2.1, we estimate the average code length of the index coder, $\tilde{L}(W_t)$, as the first order entropy of the index process, I_t. As discussed in Section 6.2.3, we assume each algorithm uses a uniform, high-resolution scalar quantizer for the codebook coder.

Consequently, the average codebook code length, $\tilde{L}(\tilde{S}_t)$, is a fixed number of bits per vector component for all time. The operational rate, $\tilde{L}(Q_t)$, is then the sum of $\tilde{L}(W_t)$ and $\tilde{L}(\tilde{S}_t)$.

In this chapter, we present performance results obtained for the Wiener process next in Section 9.1. We follow with similar results obtained for an image sequence in Section 9.2. We conclude this chapter with a summary of these experimental results in Section 9.3.

9.1 Results for the Wiener Process

In this section, we evaluate the rate-distortion performance of various AVQ algorithms on the Wiener process, one of the few nonstationary sources with a known rate-distortion function. In Section 4.5.2, we presented an expression for that rate-distortion function; in this section, we compare the performance of AVQ algorithms to this theoretically optimal performance.

In this section, we consider a Wiener process with variance $\sigma^2 = 1$. All results are averaged over 5 trials; i.e., we use a testing data set consisting of five different instances of the Wiener process. Each instance is 80,000 samples long. We use a sixth instance of the Wiener process to train initial local codebooks using the generalized Lloyd algorithm. This same training instance is used to train an initial local codebook for each combination of vector dimension and codebook size under consideration.

For each algorithm, we use a codebook coder with a uniform scalar quantizer of 512 levels; i.e., for each vector transmitted as side information, we send 9 bits per vector component. In Section 9.1.1, we present extensive results for our GTR algorithm. Then, in Section 9.1.2, we compare these results to those of other AVQ algorithms.

9.1.1 Results for the GTR Algorithm

In this section, we present the performance results obtained for the GTR algorithm for the coding of the Wiener-process data. Initially, we consider only the move-to-front variant of the GTR algorithm, and we fix the windowing parameter ω at 100; we consider both the basic algorithm and other values of ω later. Using the testing data set, we evaluate the operational rate and distortion of the GTR algorithm for a variety of different rate-distortion parameters, λ. We generate these λ values via a geometric sequence.

In Figure 9.1, we show the rate-distortion performance for the GTR algorithm for a fixed vector dimension and various local-codebook sizes. We see from this figure that increasing the codebook size yields performance closer to the theoretical rate-distortion curve, particularly in the low-rate, low-distortion area of the curve (the so-called "knee" of the curve). However, the fact that the 128- and 256-codeword curves are coincident suggests that there is a limit to the increase in performance obtainable by increasing the codebook size. As it seems that, from Figure 9.1, a local codebook of 256 codewords is sufficiently large, we use this codebook size for the remaining results of this section.

In Figure 9.2, we plot the rate-distortion performance for the GTR algorithm for a fixed local-codebook size and various vector dimensions. From rate-distortion theory, we would expect that increasing the vector dimension would yield performance increasingly close to the rate-distortion function. However, for practical implementations of AVQ, such as the GTR algorithm, the burden in rate due to side information varies with vector dimension. As illustrated in Figure 9.2, when the distortion is high, increasing the vector dimension yields performance increasingly close to the

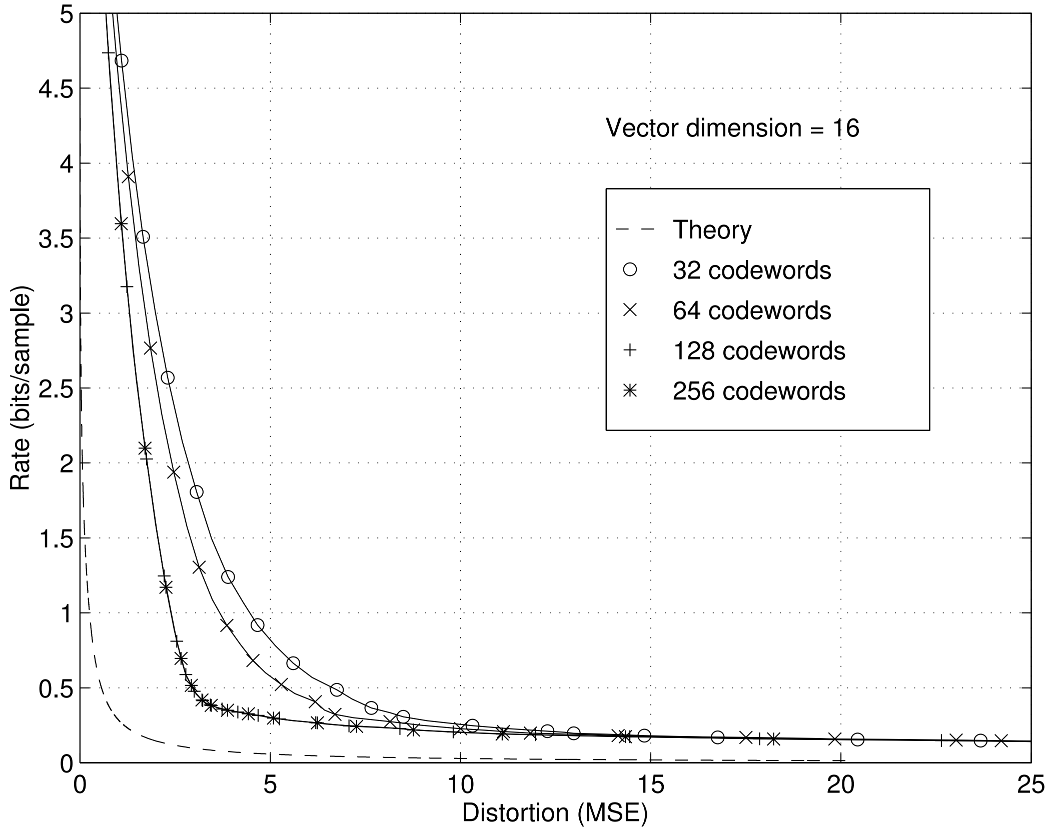


Figure 9.1: Rate-distortion performance of the GTR algorithm on the Wiener process using 16-dimensional vectors and various local-codebook sizes. Each curve is generated from left to right using an increasing, geometric sequence of λ values. For each curve, $\omega = 100$.

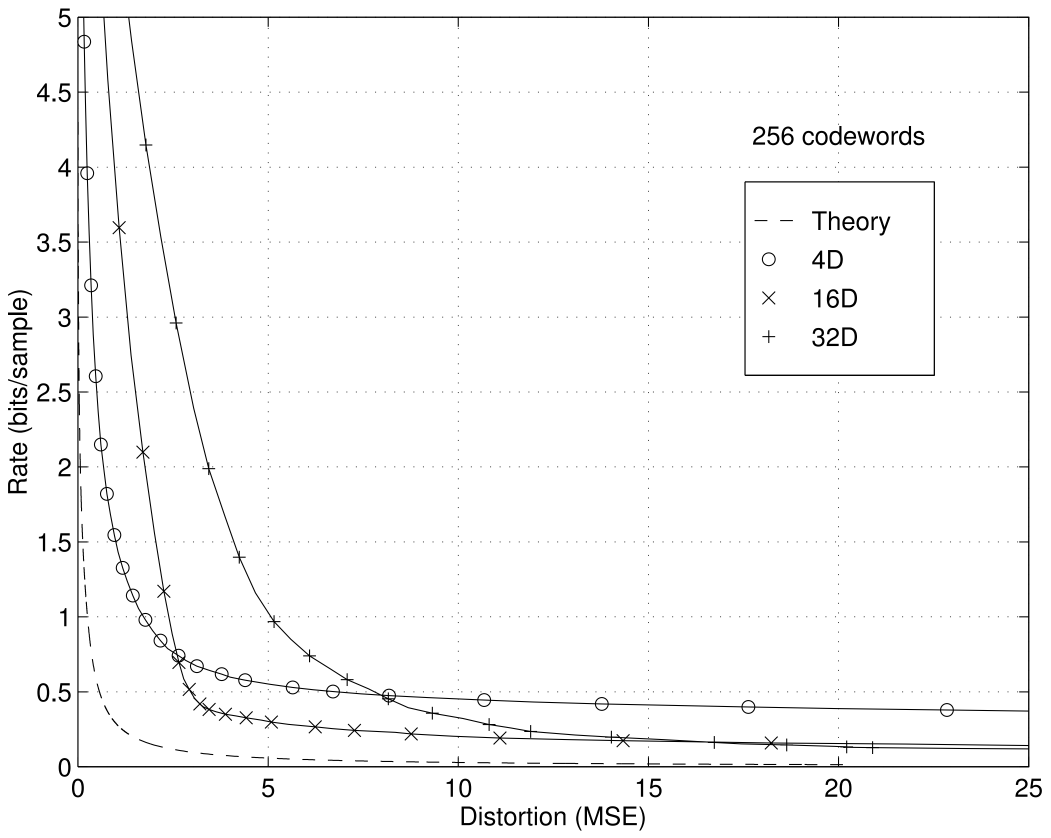
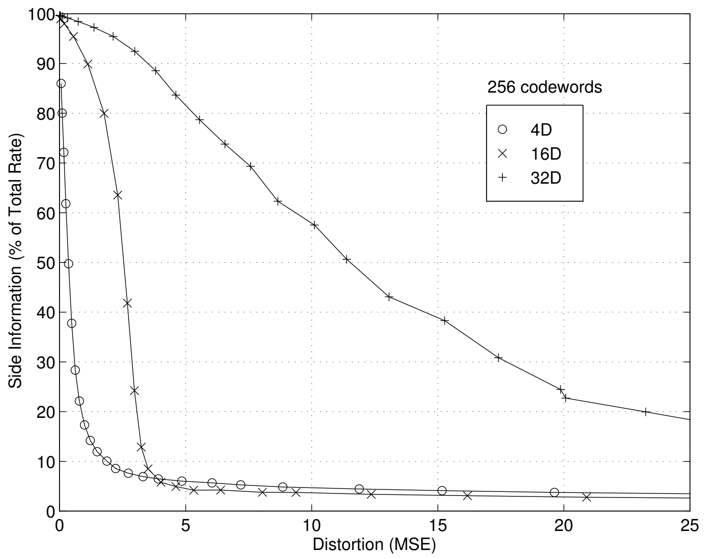
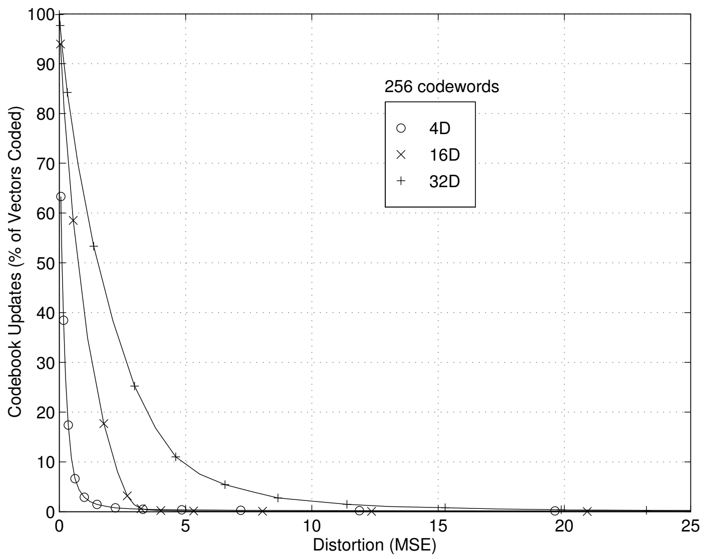


Figure 9.2: Rate-distortion performance of the GTR algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. As in Figure 9.1, each curve is generated from left to right with an increasing, geometric sequence of λ values. For each curve, $\omega = 100$.



(a) Amount of side information



(b) Frequency of codebook updates

Figure 9.3: Side information for the GTR algorithm using various codeword dimensions. For each vector dimension, we use a local codebook of 256 codewords and a windowing parameter of $\omega = 100$. (a) The amount of side information is expressed as a percentage of the total rate reported in Figure 9.2. (b) The frequency of codebook updates expressed as a percentage of the number of vectors coded.

theoretical rate-distortion curve as expected. However, we see from the same figure that, when the distortion is low, the opposite is true; that is, using smaller vector dimensions yields performance closer to the rate-distortion curve. This effect is explained in Figure 9.3. Figure 9.3a plots the amount of side information as a percentage of the total rate for the corresponding data in Figure 9.2. In Figure 9.3b, the frequency of codebook updates is plotted as a percentage of the number of vectors coded. From Figure 9.3b, we see that an increasing number of codebook updates are needed to achieve a lower distortion for each vector dimension considered. However, when the vector dimension is small, each codeword update needs fewer bits. Therefore, for low distortion levels, the side information accounts for less of the total rate when the vector dimension is small than when the vector dimension is large, as verified in Figure 9.3a. The algorithm consequently achieves better rate-distortion performance at low distortion levels with smaller vector dimensions.

The results for the GTR algorithm to this point have all used a windowing parameter $\omega = 100$. Figure 9.4 presents the rate-distortion performance curves for the algorithm using various values of ω. We conclude from Figure 9.4 that the windowing parameter has little effect on the rate-distortion performance of the algorithm. In Figure 9.5, we compare the basic GTR algorithm to the move-to-front variant. We see that the move-to-front variant has a slight performance advantage, especially when the distortion is low. We now continue to the next section which includes a comparison of the performance of the GTR algorithm as reported in this section to that of other AVQ algorithms.

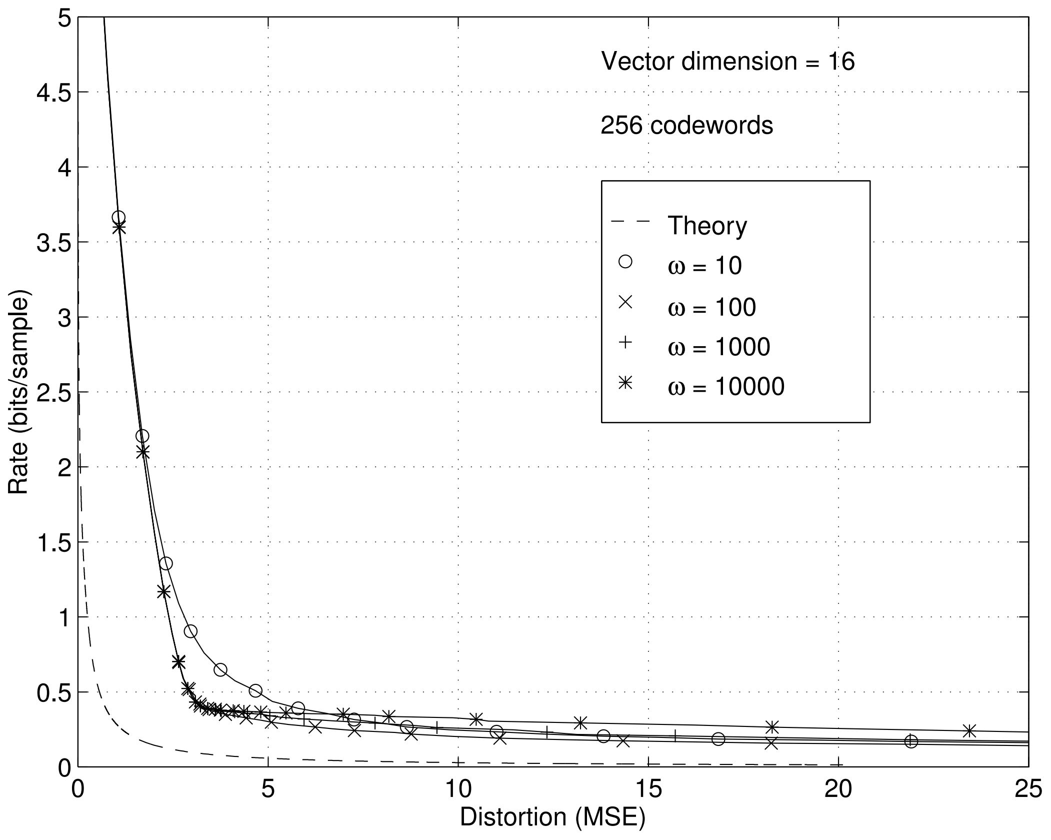


Figure 9.4: Rate-distortion performance of the GTR algorithm on the Wiener process using a local-codebook size of 256, a vector dimension of 16, and various values of the windowing parameter, ω.

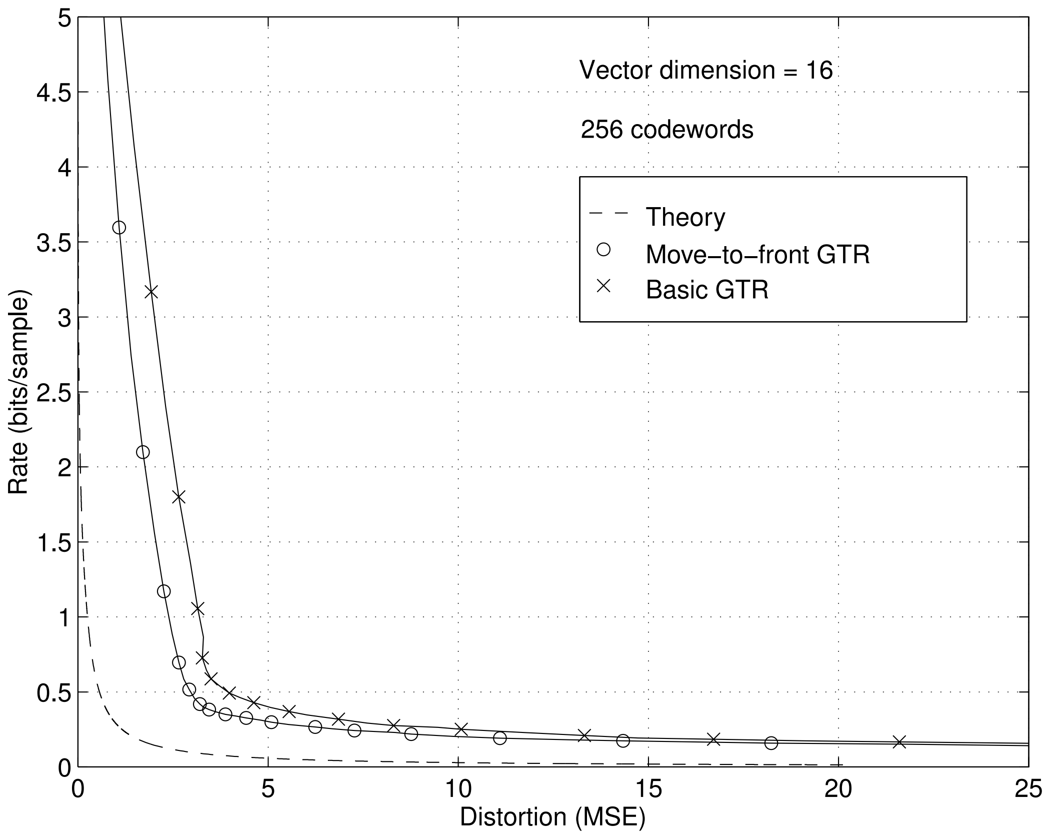


Figure 9.5: Comparison of the basic and move-to-front variants of the GTR algorithm. Results are run for the Wiener process using 16-dimensional vectors, a local-codebook size of 256 codewords, and a windowing parameter $\omega = 100$.

9.1.2 Comparisons to Previous AVQ Algorithms

In this section, we compare the rate-distortion performance of the GTR algorithm to that of other AVQ algorithms. To facilitate the discussion, we present only the results obtained for a local codebook of 256 vectors although similar results have been observed for different local-codebook sizes. In Figure 9.2, we presented results for the GTR algorithm using a local-codebook size of 256 codewords and various vector dimensions; we now give corresponding results for each of the other AVQ algorithms under consideration. At the end of this section, we compare these results to those of the GTR algorithm.

In Figure 9.6, we show the rate-distortion performance for the Paul algorithm on the Wiener-process testing data set for a variety of vector dimensions. We use the move-to-front variant of the algorithm. Note that, like the GTR algorithm, the Paul algorithm also achieves better rate-distortion performance at low distortion when the vector dimension is small.

The rate-distortion performance of the Gersho-Yano algorithm on the same testing data is given in Figure 9.7. We use the basic algorithm as described in Section 7.2.4. The size of the adaption interval, τ, used by the algorithm dictates the operating point on the rate-distortion curve; we create the plots in Figure 9.7 by varying τ on the order of 1 to 1,000 vectors.

Similar results are shown in Figure 9.8 for the Lancini-Perego-Tubaro algorithm. The competitive-learning network used by the algorithm trains a new codebook of size $K_T = 8$ codewords of which $K_R = 2$ are chosen to add to the local codebook. The learning rule of the competitive-learning network is

$$\mathbf{c}_i = \mathbf{c}_i + \epsilon \cdot e^{-u_i/T} \left[\mathbf{X} - \mathbf{c}_i\right], \tag{9.1}$$

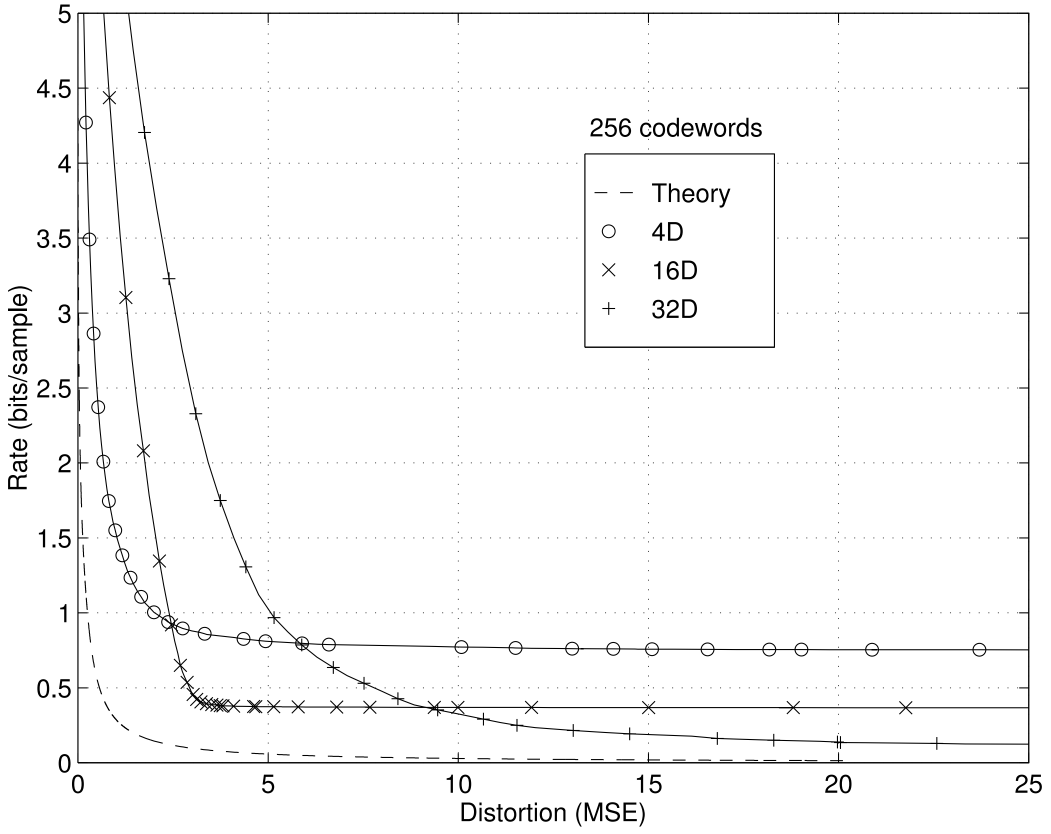


Figure 9.6: Rate-distortion performance of the move-to-front Paul algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. Each curve is drawn from the left to the right by increasing the distortion-threshold parameter to the algorithm, $D_{\max}$.

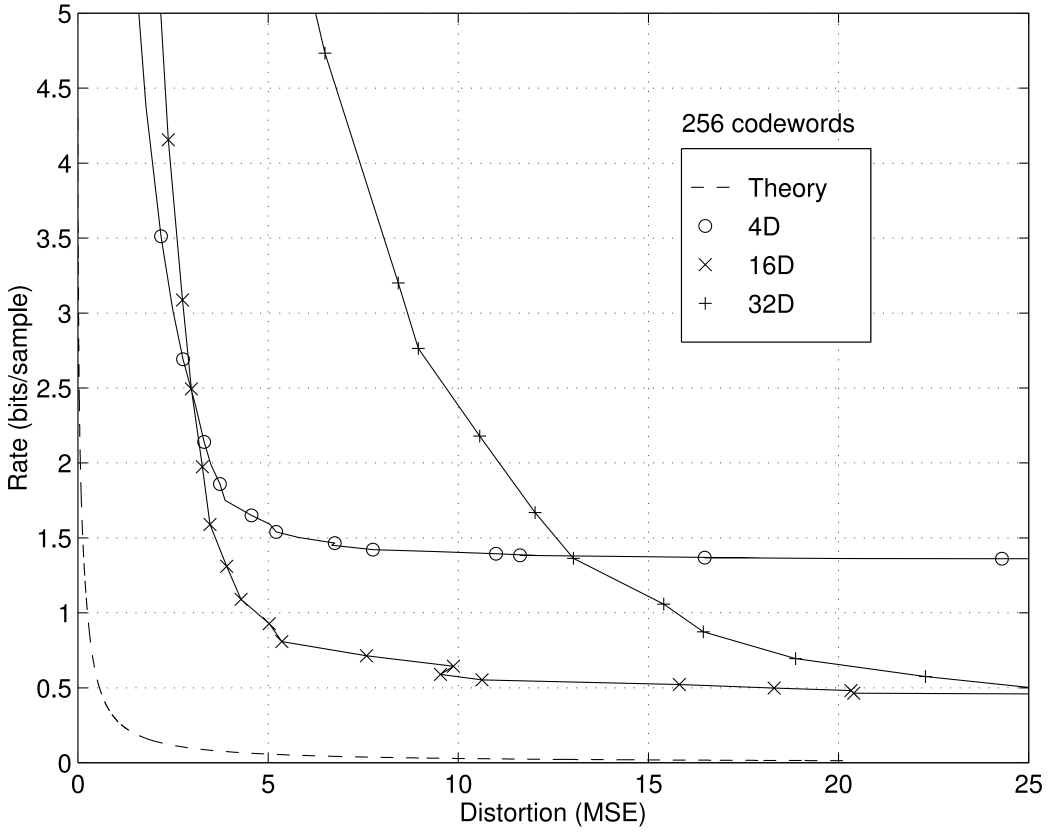


Figure 9.7: Rate-distortion performance of the Gersho-Yano algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. Each curve is drawn from the left to the right by increasing the adaption interval, τ, used by the algorithm.

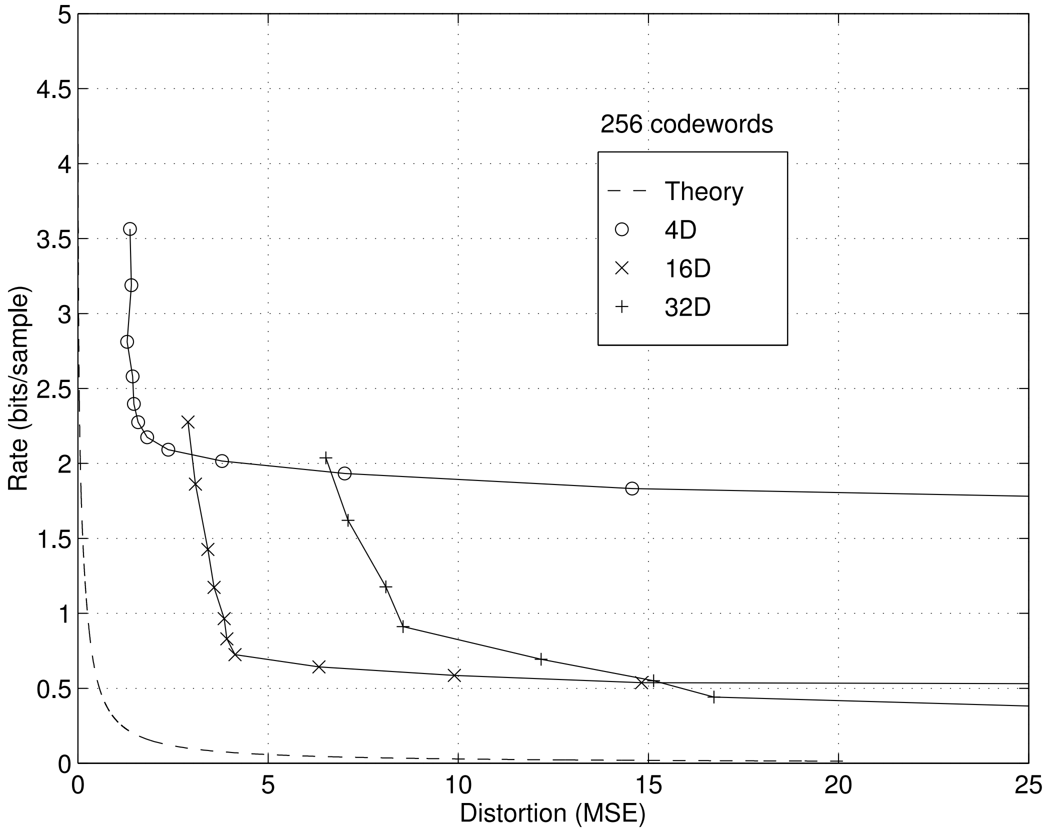


Figure 9.8: Rate-distortion performance of the Lancini-Perego-Tubaro algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. Each curve is drawn from the left to the right by increasing the adaption interval, τ, used by the algorithm.

where u_i is a counter of the number of times $\mathbf{c}_i$ has won the competition. We choose $\epsilon = 0.02$ and $T = 100$. We trace out the rate-distortion performance curves in Figure 9.8 by varying the adaption interval τ from 10 to 1,000 vectors.

Figure 9.9 depicts the Wiener-process results for the Wang-Shende-Sayood algorithm. For these results, we use the D_N lattice as the universal codebook, where N is the vector dimension. As discussed by Conway and Sloane [57], of the root lattices in 4-dimensional space, the D_4 lattice features the smallest second moment, a quantity related to the expected distortion of the lattice (see, for example, [29]). However, for higher dimensions, other lattices perform better than the D_N lattice [57]. For this reason, the Wang-Shende-Sayood algorithm using the D_N lattice suffers in rate-distortion performance relative to other AVQ algorithms for dimensions $N > 4$, although it is competitive for $N = 4$.

The results for the Goldberg-Sun algorithm are shown in Figure 9.10. A replenishment threshold of $\delta = 0.001$ was used for each plot. As can be seen from the figure, the Goldberg-Sun algorithm is heavily burdened with side information, resulting in a rate-distortion performance much worse than that of the other algorithms considered thus far. Although we do not present them here, results for several other values of δ were calculated. While higher values of δ were successful in reducing the amount of side information to levels more comparable to those of other algorithms, the resulting distortion for these large δ values was much higher than that of the other AVQ algorithms.

We note that we do not present Wiener-process results for either the Lightstone-Mitra or gain-adaption AVQ algorithms. The Lightstone-Mitra algorithm had a severe problem with "dormant" codewords. This effect, discussed in Section 7.3, results from

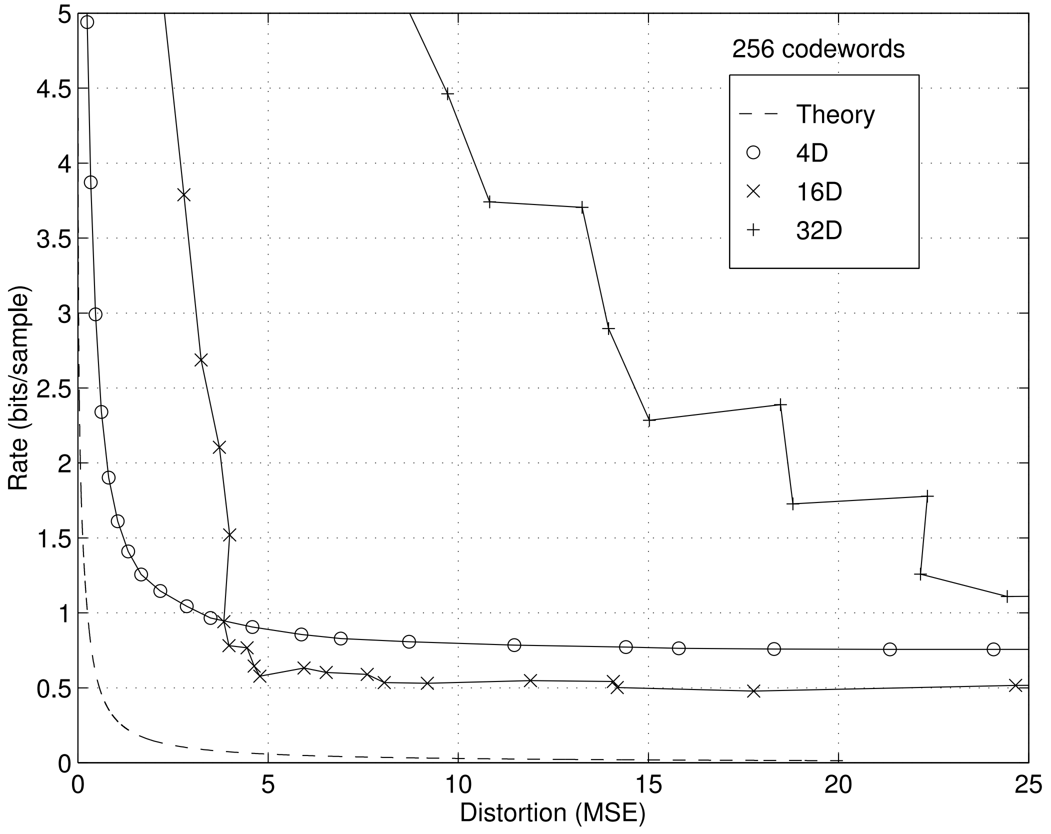


Figure 9.9: Rate-distortion performance of the Wang-Shende-Sayood algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. The D_N lattice is used for the universal codebook. Each curve is plotted from left to right by increasing the distortion threshold, $D_{\max}$.

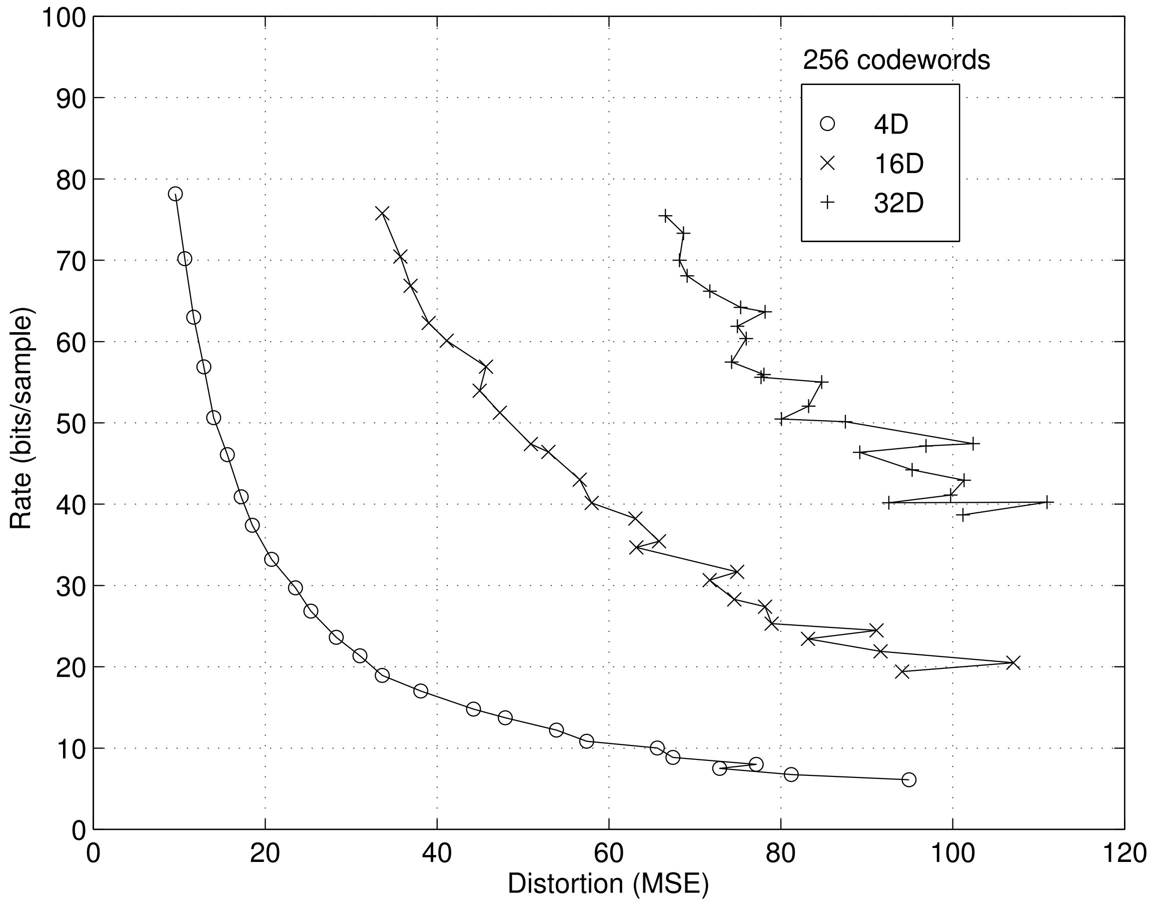


Figure 9.10: Rate-distortion performance of the Goldberg-Sun algorithm on the Wiener process using a local codebook of 256 codewords and various vector dimensions. A replenishment threshold of $\delta = 0.001$ was used, and the adaption interval, τ, was increased to trace out each curve from the left to the right.

the reduction of the size of the training codebook by each ECVQ training iteration. For the Wiener process, the training codebook size quickly reduced to one codeword, essentially rendering the Lightstone-Mitra algorithm equivalent to nonadaptive VQ with a higher rate due to nonzero side information. For this reason, the Lightstone-Mitra algorithm is not competitive with the other AVQ algorithms on the Wiener-process data and so is not considered further in this section. Additionally, we do not present any results for the gain-adaption algorithm. The gain-adaption algorithm requires, in practice, a much different codebook-design paradigm (see [2]) than that of the other AVQ algorithms under consideration here. This design procedure involves the training of a “gain-normalized” VQ codebook and the establishment of a suitable gain-estimation mechanism. As the performance of the gain-adaption algorithm is highly dependent on these design procedures [2], it is difficult to make a meaningful comparison to the other AVQ algorithms which all use the same initial codebook generated by the generalized Lloyd algorithm. Thus, we do not consider the gain-adaption algorithm further here.

To facilitate the comparison of the performances of the AVQ algorithms, we replot, in Figures 9.11 through 9.13, the results of this section (Figures 9.6 through 9.9), combined with the corresponding results from the last section for the GTR algorithm (Figure 9.2), using a separate figure for each value of vector dimension. We see from Figures 9.11 through 9.13 that the GTR and Paul algorithms perform consistently better than any other AVQ algorithm. Additionally, the performance of the GTR algorithm is always at least as good as that of the Paul algorithm. In most cases, the GTR algorithm is closer to the theoretical rate-distortion curve than the Paul algorithm for low-rate, high-distortion coding, while the algorithms have equivalent

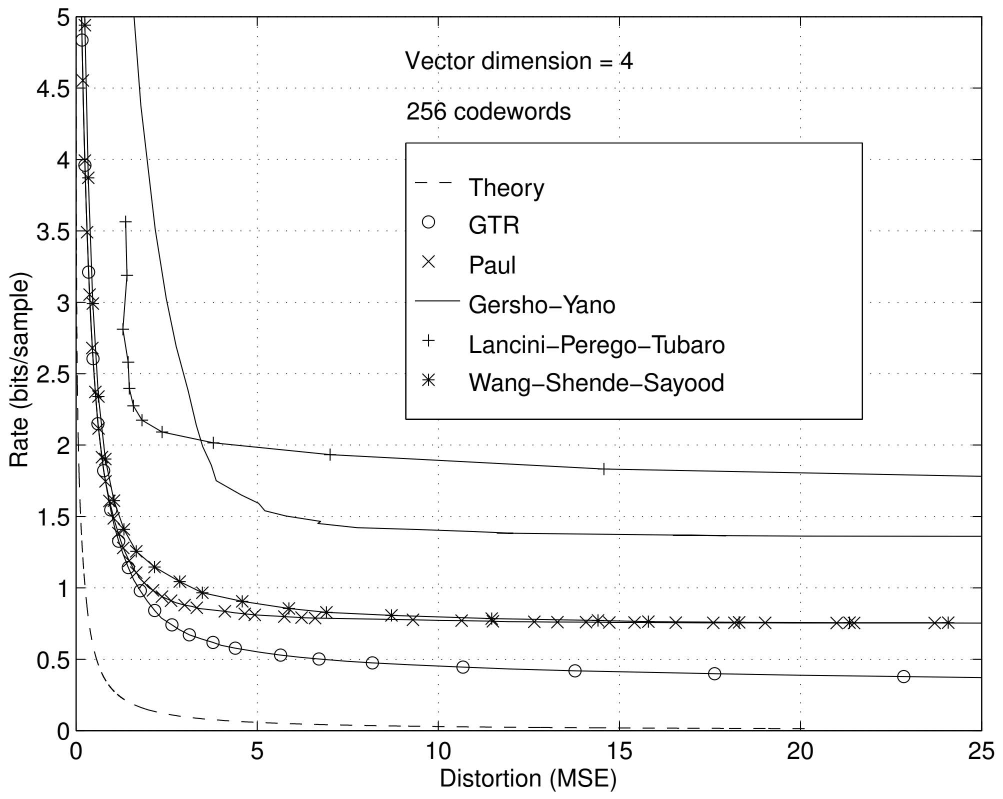


Figure 9.11: Rate-distortion performance of various AVQ algorithms on the Wiener process with 4-dimensional vectors

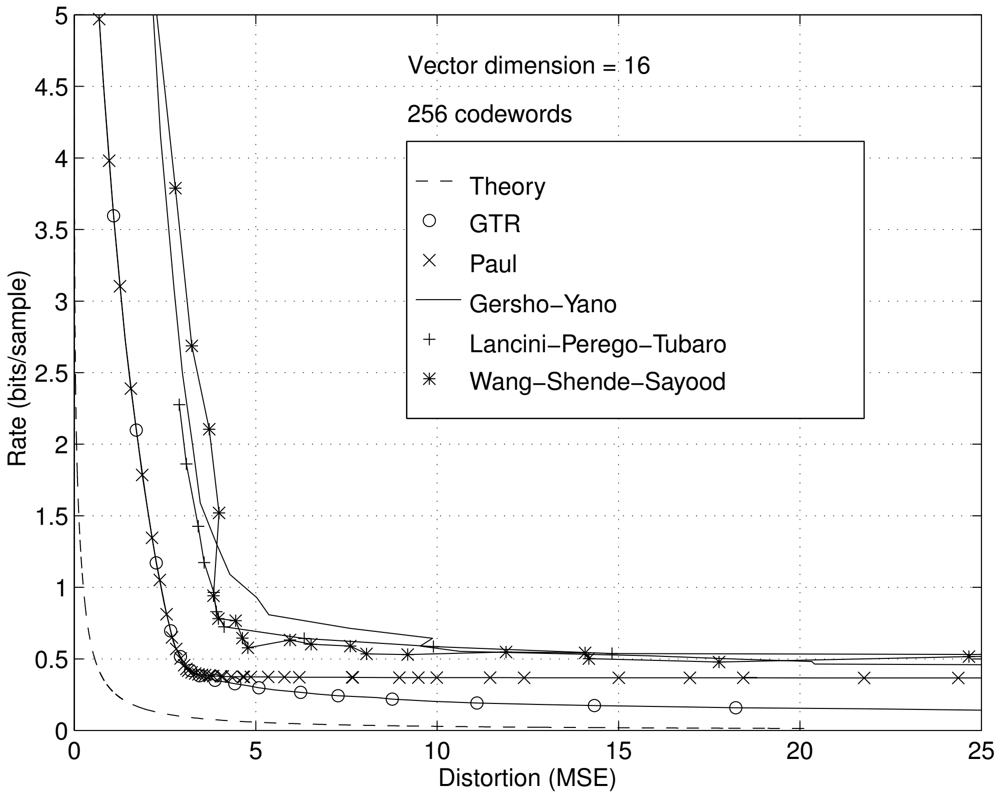


Figure 9.12: Rate-distortion performance of various AVQ algorithms on the Wiener process with 16-dimensional vectors

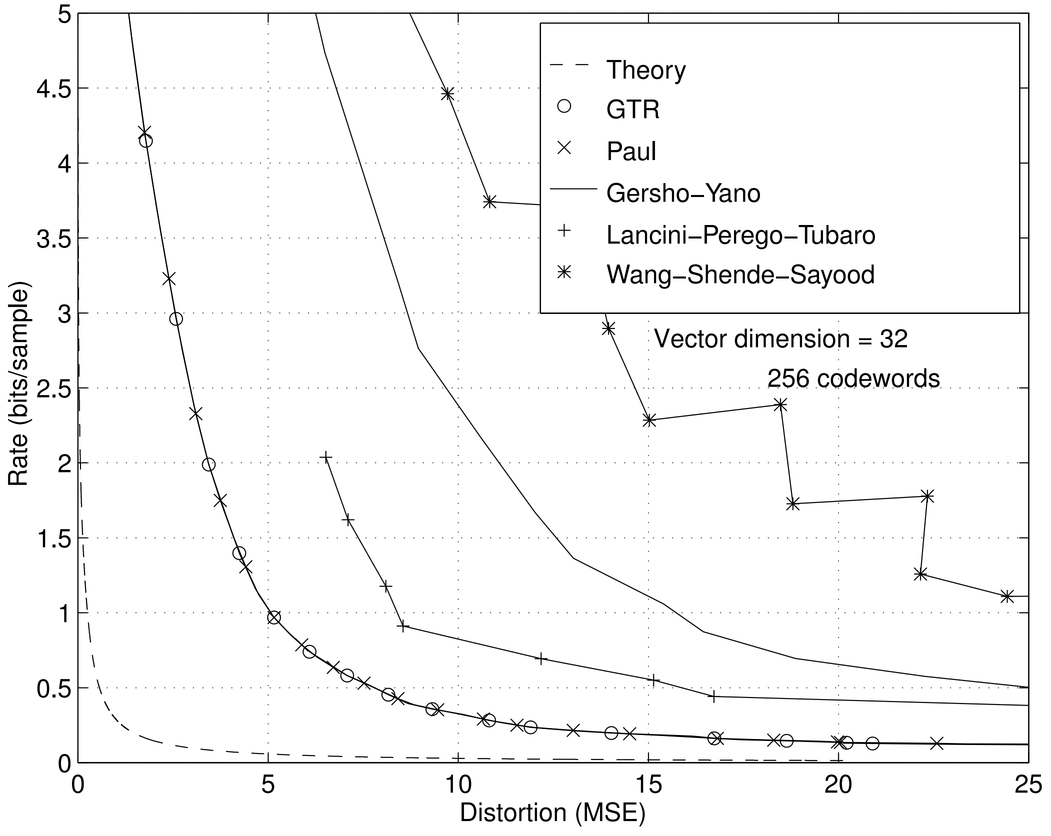


Figure 9.13: Rate-distortion performance of various AVQ algorithms on the Wiener process with 32-dimensional vectors

Dimension	Rate (bits/sample)	Distortion (MSE)
4	1.275	345.0
16	0.419	316.0
32	0.133	439.0

Figure 9.14: Rate-distortion performance for nonadaptive VQ on the Wiener process using a VQ codebook of 256 codewords and various vector dimensions. The VQ codebook is the initial codebook used by the AVQ algorithms.

performance for high-rate, low-distortion coding. We will further elaborate on the coding performance of each AVQ algorithm in these different regions of the rate-distortion plane in Section 9.3.

Finally, in Figure 9.14, we present the rate and distortion obtained for nonadaptive VQ of the Wiener-process data. For the nonadaptive VQ codebook, we use the initial codebook that we trained for the AVQ algorithms using the generalized Lloyd algorithm. The rate reported for the nonadaptive vector quantizer is the measured first-order entropy of the VQ indices. We see that the nonadaptive vector quantizer results in a much higher distortion when compared at the same rate to the AVQ algorithms.

These observations conclude our consideration of the Wiener process. We now continue the discussion in the next section by examining the performance of the AVQ algorithms for an image sequence.

9.2 Image-Sequence Results

In this section, we consider the results obtained for the AVQ of the image sequence shown in Figure 9.15. This sequence, which we will use as a testing data set, consists

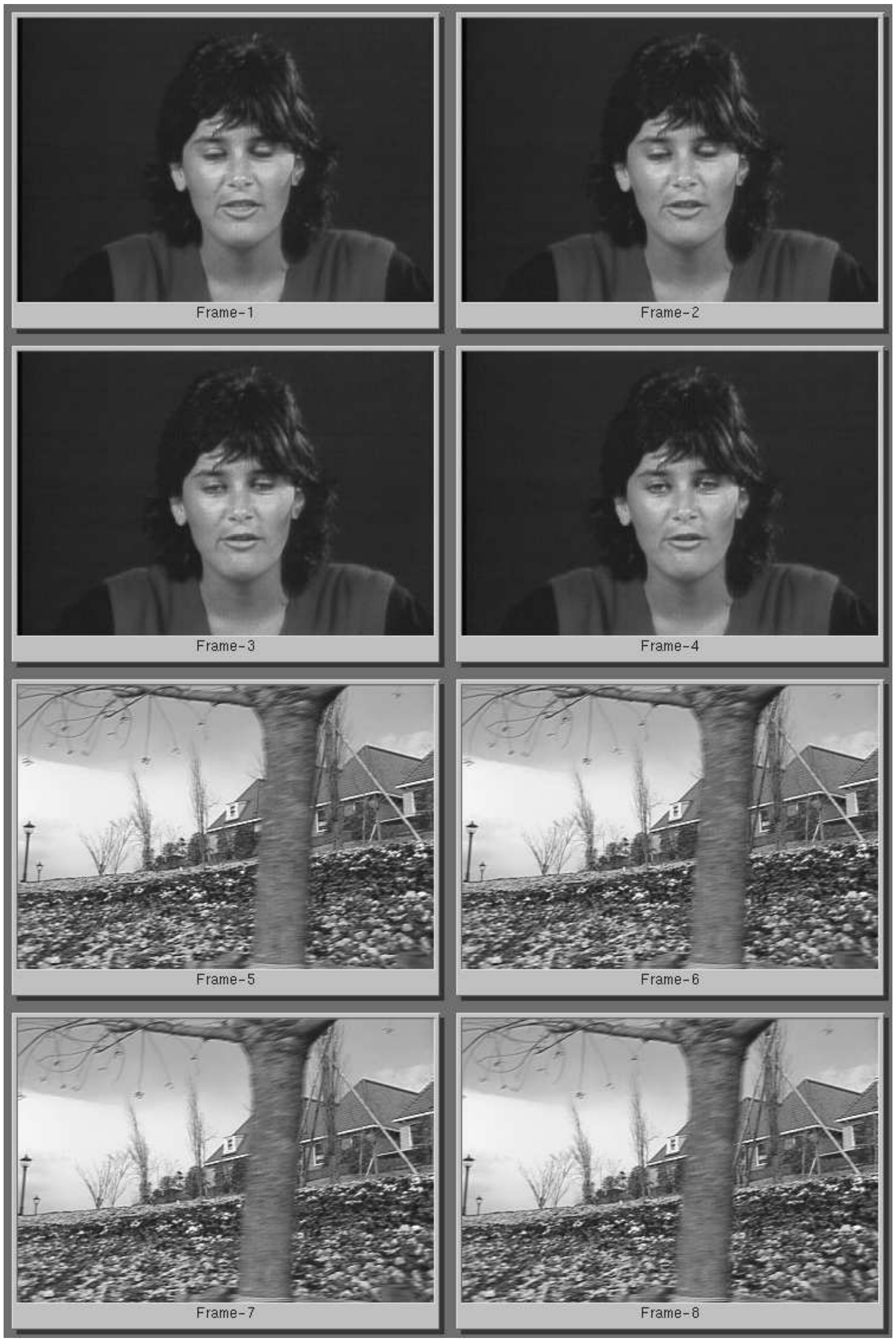


Figure 9.15: Original image sequence

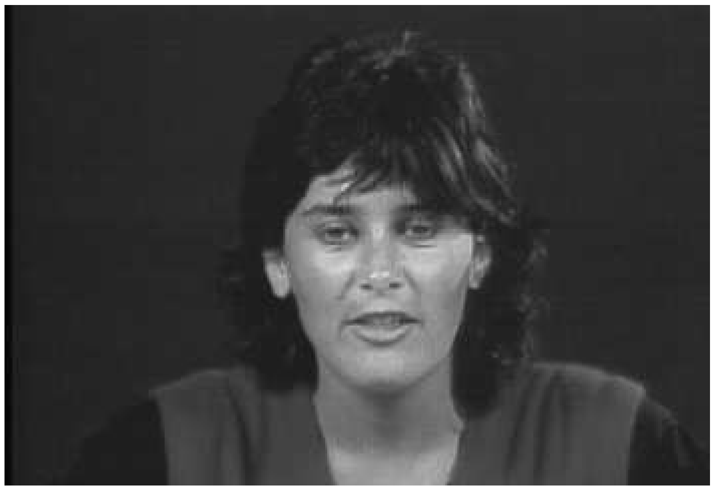


Figure 9.16: Training image used to generate initial local codebooks

of 8 image frames, 4 frames from the image sequence "Miss America" followed by 4 frames from the "Garden" sequence. Each image of the sequence is grayscale with 256 levels and has a resolution of 352×240 pixels. We use an additional frame, shown in Figure 9.16, from the "Miss America" sequence as a training data set. The generalized Lloyd algorithm is run on this training data to produce an initial local codebook for each vector dimension and codebook size under consideration. For each AVQ algorithm under consideration, we will use a codebook coder that transmits 8 bits per vector component for side information.

In the next section, we investigate the performance of the GTR algorithm on the image sequence of Figure 9.15. Then, in Section 9.2.2, we make some comparisons between the performance of the GTR algorithm and that of other AVQ algorithms.

9.2.1 Results for the GTR Algorithm

In Figure 9.17, we plot the rate-distortion performance of nonadaptive VQ versus that of the GTR algorithm for the image sequence using a local-codebook size of 256 codewords and 4-dimensional vectors. For the nonadaptive-VQ result, the VQ codebook used over the entire image sequence is the same initial local codebook given to the GTR algorithm, and the rate is the first-order entropy of the VQ indices. For the GTR algorithm, we vary the rate-distortion parameter λ to obtain the rate-distortion performance curve for the algorithm. In Figures 9.18 and 9.19, we show quantized image sequences produced by the GTR algorithm and nonadaptive VQ, respectively. To allow for a fair, albeit subjective, comparison of the perceptual image quality of each quantized sequence, we set $\lambda = 16$ to produce Figure 9.18; consequently, the GTR algorithm operates at approximately the same rate as the nonadaptive vector quantizer.

We see from Figure 9.17 that the GTR algorithm features much better rate-distortion performance than nonadaptive VQ when the performance over the entire process is considered. However, when we visually compare the image quality at equal rates by examining Figures 9.18 and 9.19, we observe that nonadaptive VQ performs slightly better on the first 4 frames of the sequence, while the GTR algorithm is substantially better on the last 4 frames of the sequence. This subjective observation is quantitatively verified in Figure 9.20 where we plot the average distortion and rate obtained for each frame. Since the nonadaptive vector quantizer is using a codebook trained on data very similar to that of the first 4 frames of the sequence, its performance is quite good on those early frames. However, since the nonadaptive-VQ codebook has very few codewords that are suited for coding the latter frames of the

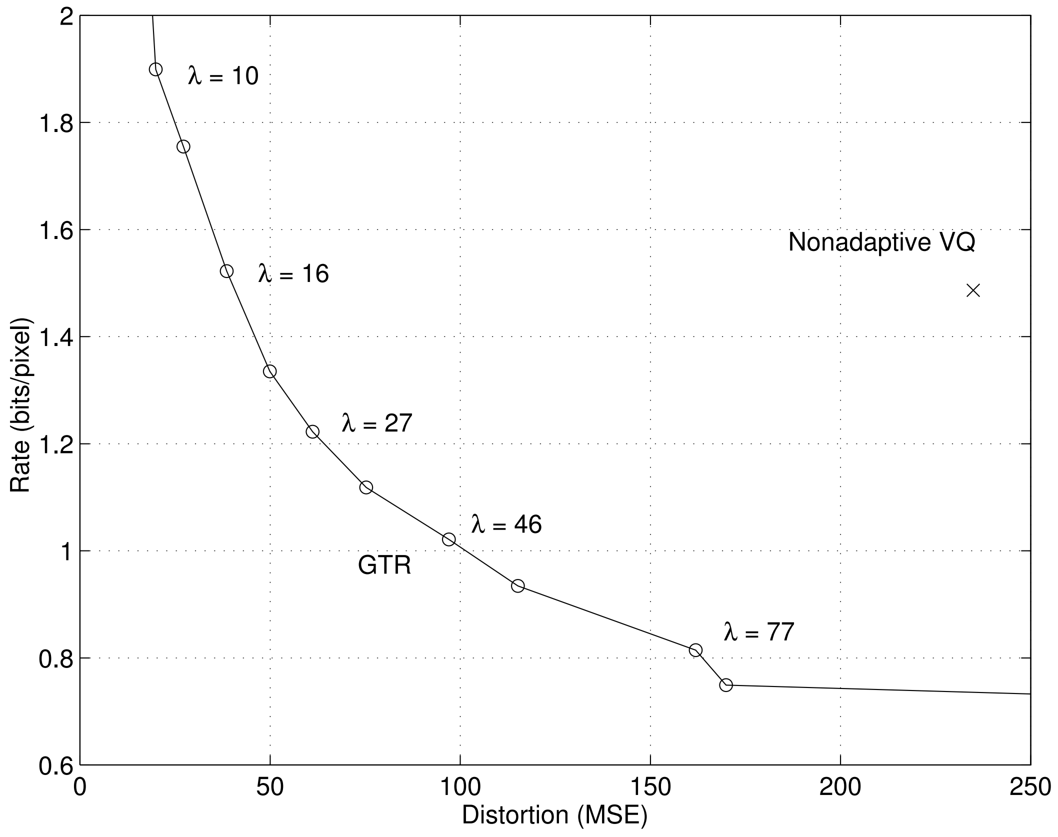


Figure 9.17: The GTR algorithm versus nonadaptive VQ for the image sequence using 4-dimensional vectors and a local codebook of 256 codewords. The GTR algorithm operates at approximately the same rate as the nonadaptive vector quantizer when $\lambda = 16$.

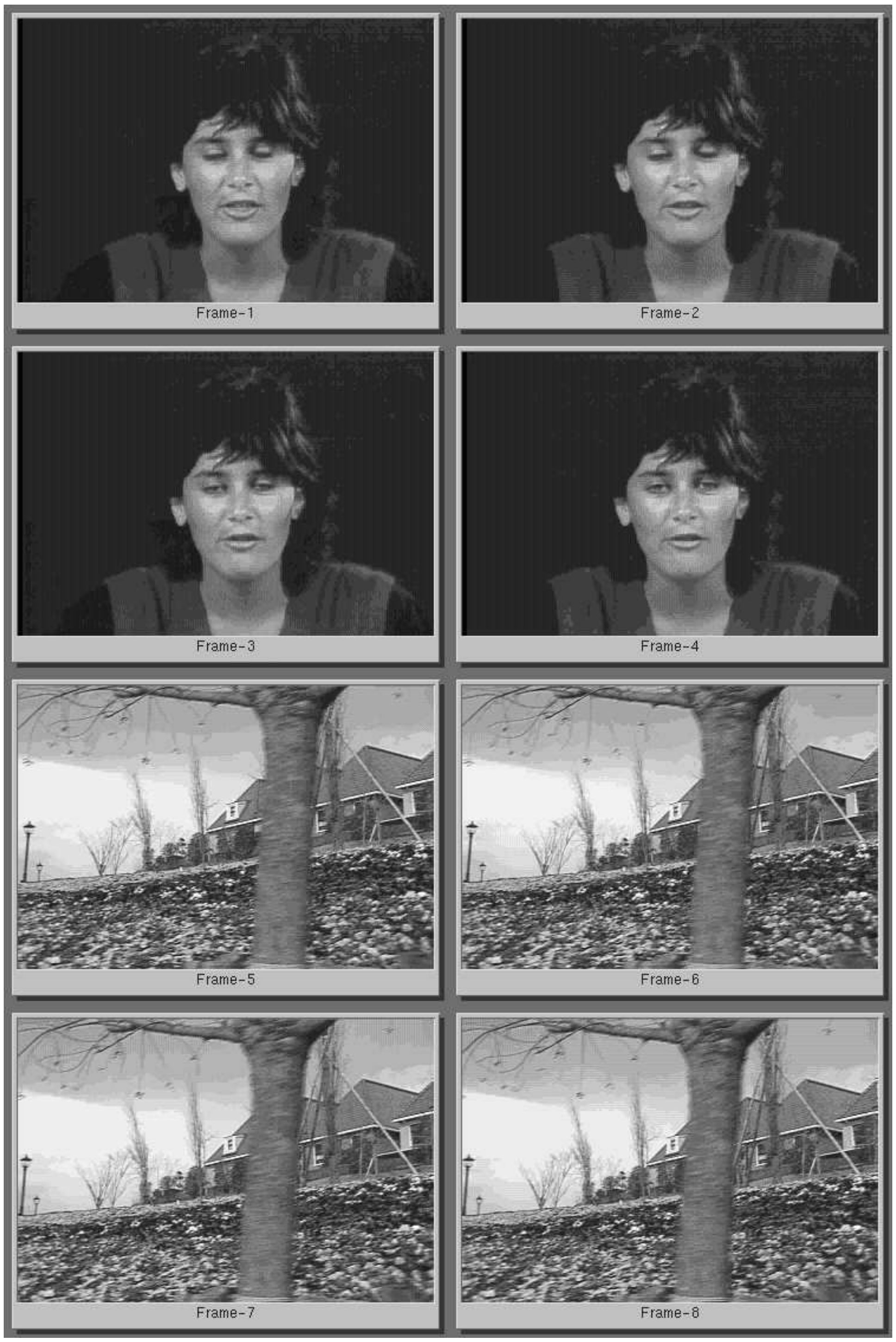


Figure 9.18: Quantized image sequence for the GTR algorithm for 4-dimensional vectors and a local codebook of 256 codewords ($\lambda = 16$, MSE = 38.6, rate = 1.522 bits/pixel).

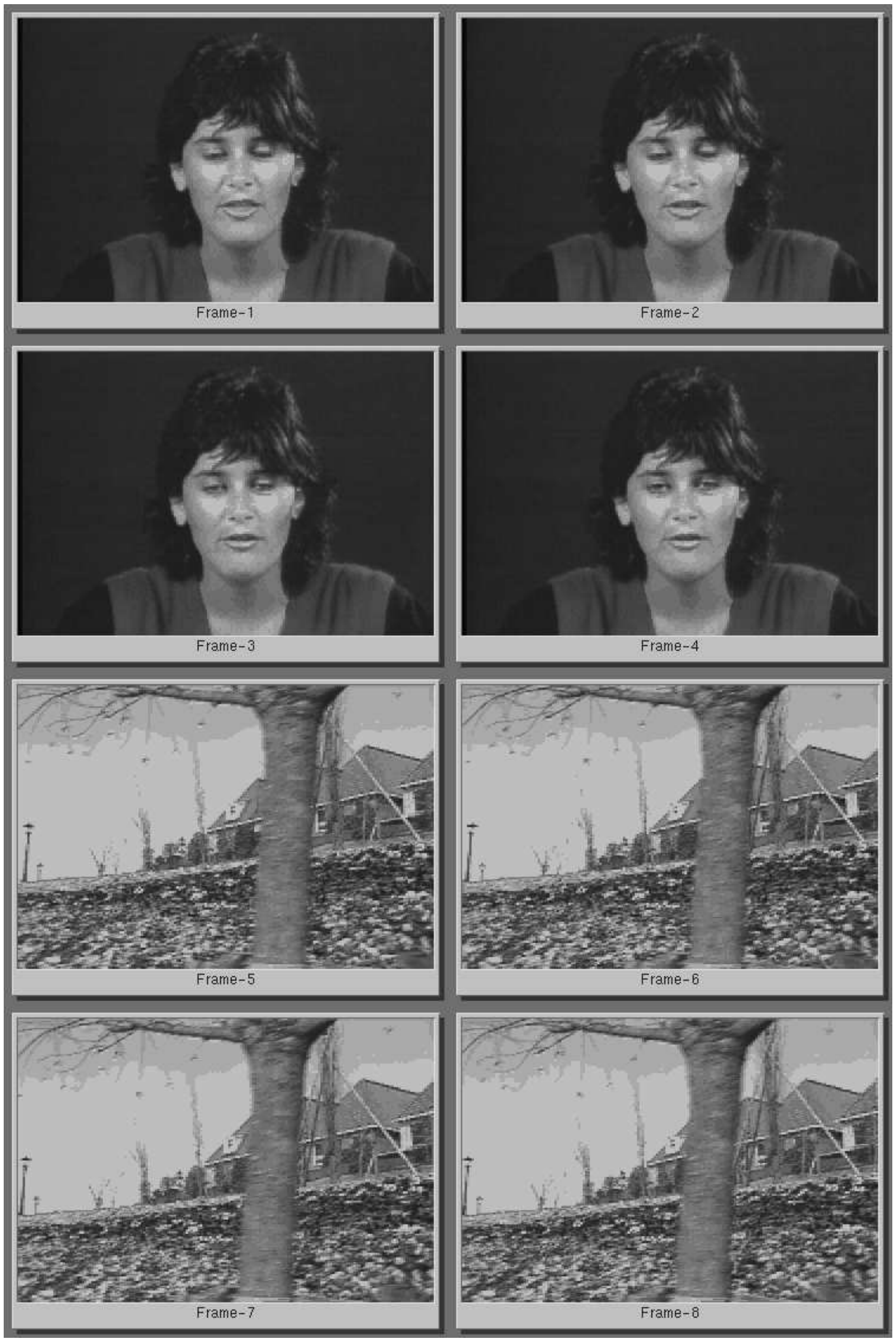
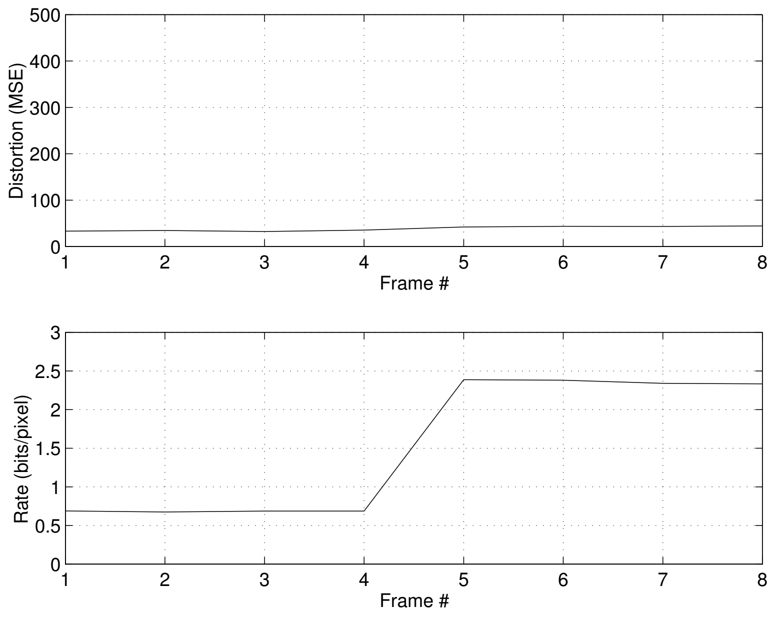
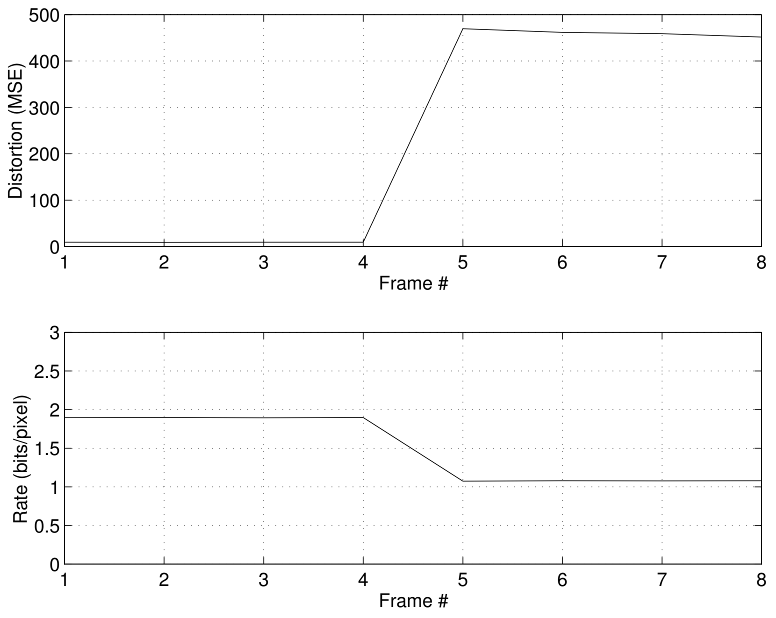


Figure 9.19: Quantized image sequence for nonadaptive VQ for 4-dimensional vectors and a VQ codebook of 256 codewords (MSE = 234.9, rate = 1.487 bits/pixel).



(a) The GTR algorithm ($\lambda = 16$)



(b) Nonadaptive VQ

Figure 9.20: Average distortion and rate per frame of the image sequence for 4-dimensional vectors and a local codebook of 256 codewords

sequence, poor distortion performance is observed for the nonadaptive vector quantizer on these latter frames. On the other hand, the adaptive nature of the GTR algorithm allows it to increase the rate on the latter frames in order to maintain roughly even distortion performance over the entire sequence. Additionally, we observe that the GTR algorithm preserves edges and other areas of high detail much better than nonadaptive VQ, as it is these areas that are likely to induce a codebook update within the GTR algorithm. Since these areas, which compose a majority of the regions in the latter frames of the sequence, are crucial to perceptual image quality, not only does the GTR algorithm have a much better empirically measured average distortion, it also achieves better subjective performance on the latter frames of the sequence when compared to the nonadaptive vector quantizer operating at the same rate.

For completeness, in Figures 9.21 through 9.24, we present for a vector dimension of 16 the same results that have been given thus far for a vector dimension of 4. To allow a subjective comparison of image quality at equal rates, we use a rate-distortion parameter of $\lambda = 209$ in the GTR algorithm to produce the quantized image sequence depicted in Figure 9.22. We see in Figure 9.21 that the rate-distortion performance of nonadaptive VQ is closer to that of the GTR algorithm for this vector dimension. However, Figure 9.23 indicates that the nonadaptive vector quantizer operates at a distortion unacceptably high for most applications in this case.

In the next section, we continue the discussion of image-coding results by comparing the performances of other AVQ algorithms to that of the GTR algorithm.

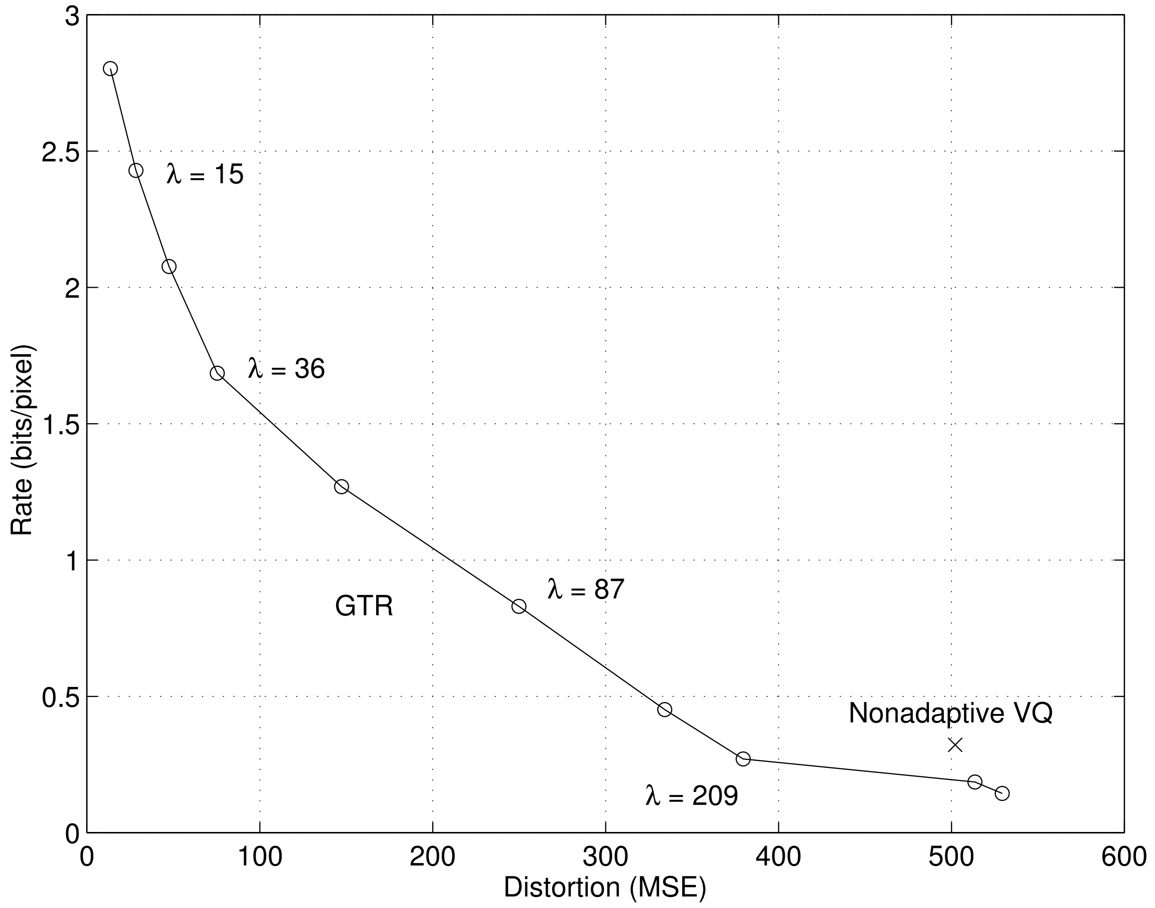


Figure 9.21: The GTR algorithm versus nonadaptive VQ for the image sequence using 16-dimensional vectors and a local codebook of 256 codewords. The GTR algorithm operates at approximately the same rate as the nonadaptive vector quantizer when $\lambda = 209$.

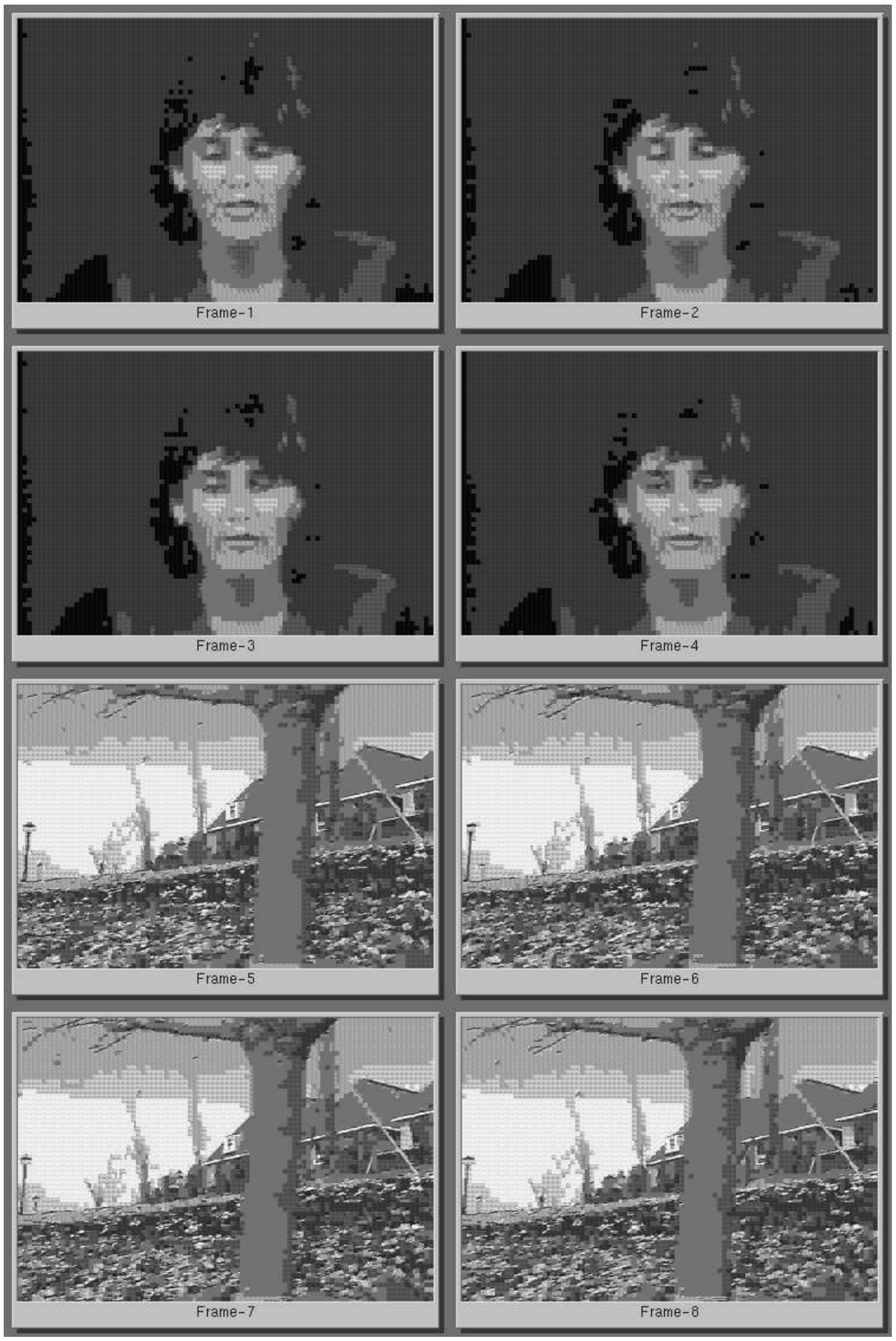


Figure 9.22: Quantized image sequence for the GTR algorithm for 16-dimensional vectors and a local codebook of 256 codewords ($\lambda = 209$, MSE = 379.5, rate = 0.271 bits/pixel).

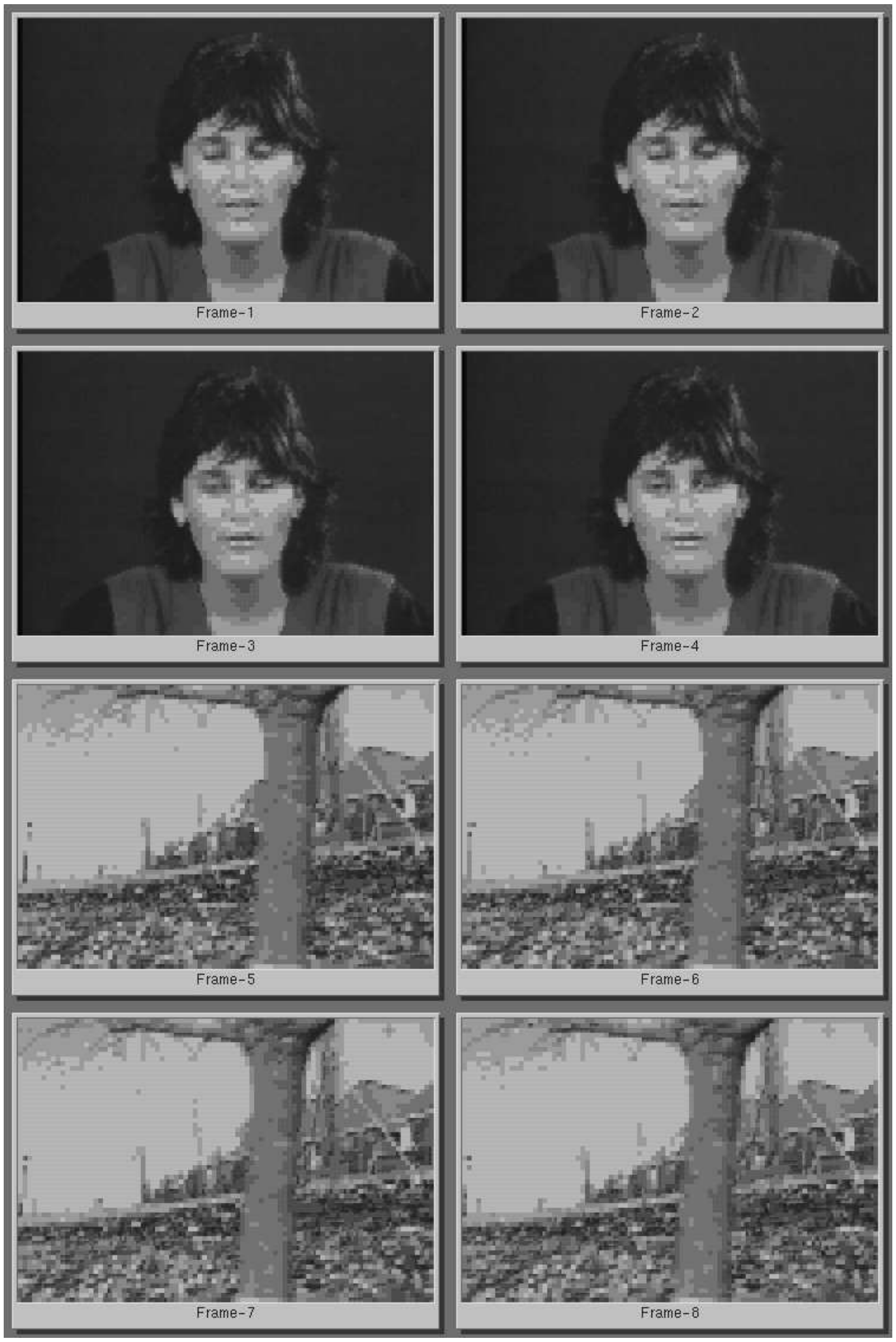
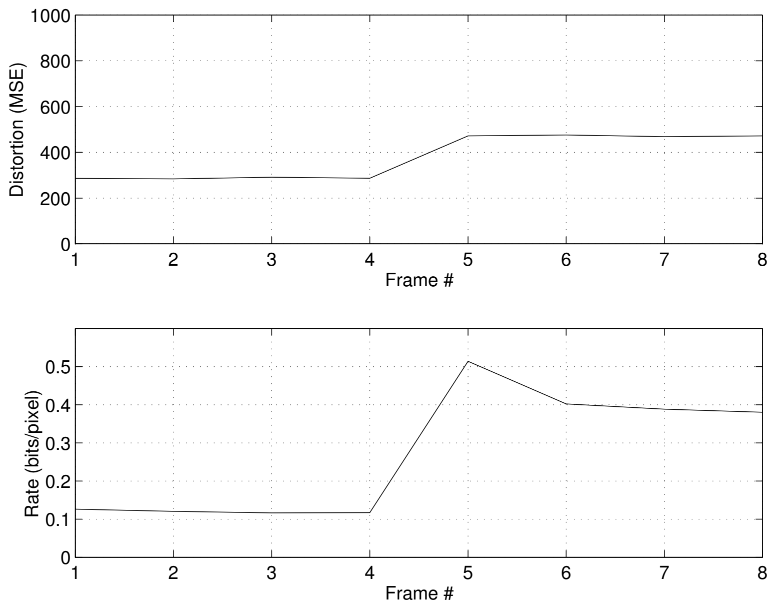
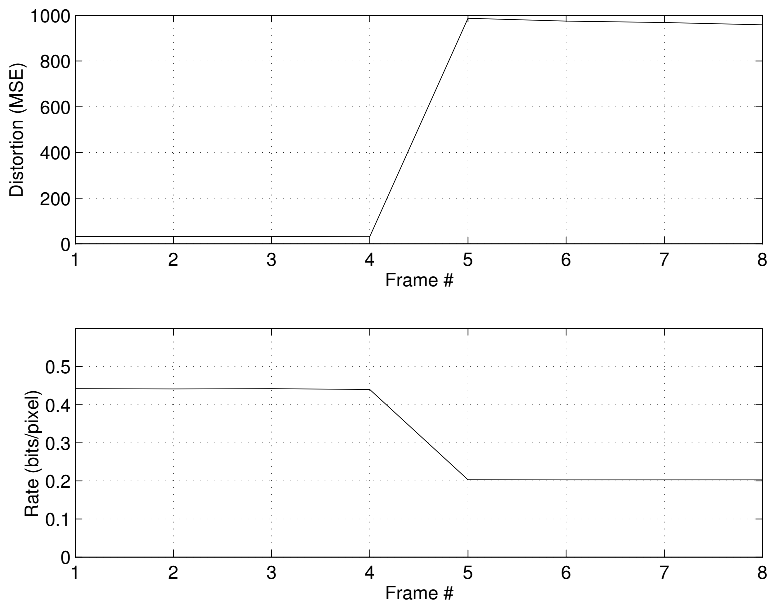


Figure 9.23: Quantized image sequence for nonadaptive VQ for 16-dimensional vectors and a VQ codebook of 256 codewords (MSE = 502.1 , rate = 0.322 bits/pixel).



(a) The GTR algorithm ($\lambda = 209$)



(b) Nonadaptive VQ

Figure 9.24: Average distortion and rate per frame of the image sequence for 16-dimensional vectors and a local codebook of 256 codewords

9.2.2 Comparisons to Previous AVQ Algorithms

In this section, we use the image sequence of Figure 9.15 to compare the relative performances of AVQ algorithms in image-coding applications. The rate-distortion performance curves for various AVQ algorithms for 4-dimensional vectors and a local codebook of 256 codewords are plotted in Figure 9.25. The Goldberg-Sun algorithm used a replenishment threshold of $\delta = 0.0001$; all other constant parameters to the algorithms were the same as those used in Section 9.1.2. Again, we do not present results for the Lightstone-Mitra algorithm as it was as hindered by dormant codewords for the image data as it was for the Wiener process. Neither do we consider the gain-adaption algorithm in this section because of its design process being substantially different than that of the other AVQ algorithms, as discussed in Section 9.1.2.

For the constrained-rate AVQ algorithms, namely the the Gersho-Yano, Goldberg-Sun, and Lancini-Perego-Tubaro algorithms, the adaption interval was increased to a maximum value of $\tau = 2112$ vectors (i.e., one image frame) to trace out the rate-distortion curve. This value of τ was chosen to be the maximum allowed adaption-interval size since, due to buffering limitations occurring in real implementations of image coding, one image frame would normally be the maximum amount of data to which an algorithm could have access at any given time. The lowest rate obtained for these algorithms occurred at this maximum τ value. Note that for each of the constrained-rate algorithms, the rate-distortion curve has more or less a vertical slope. This effect occurs due to the fact that the constrained-rate algorithms concentrate on reducing distortion without considering the corresponding increase in rate. We see in Figure 9.25 that, for the constrained-rate algorithms considered here, a substantial cost in rate is incurred for any significant reduction in distortion.

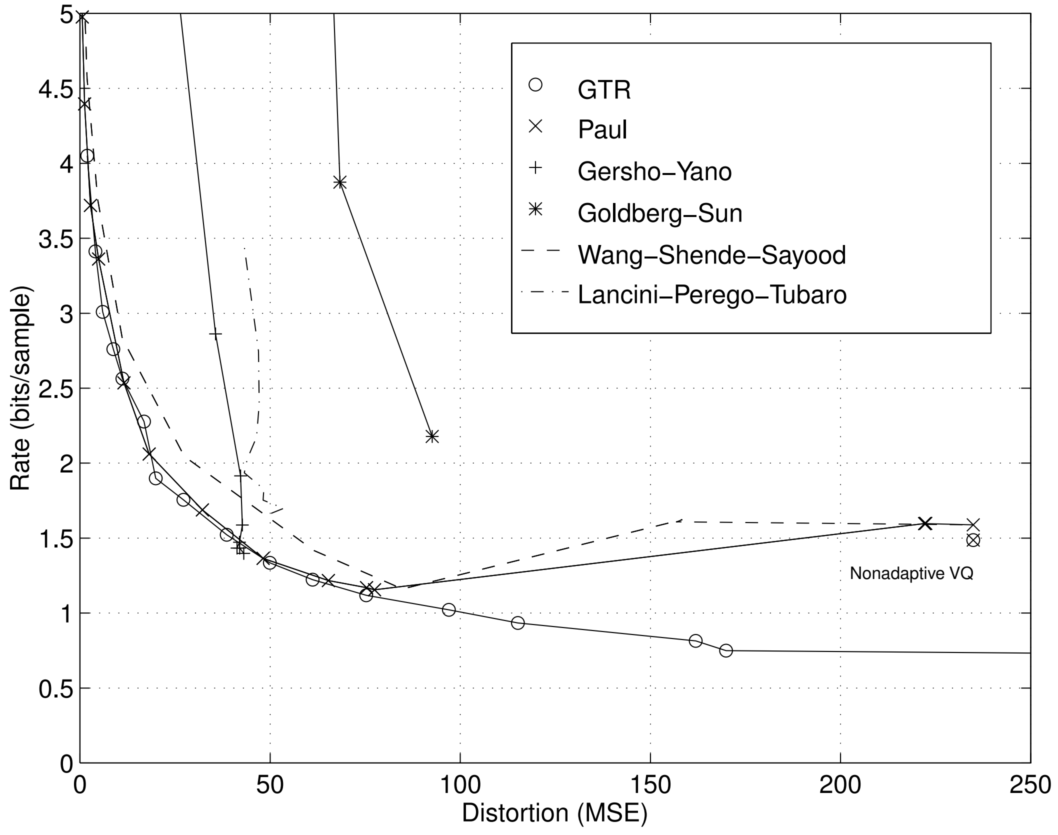


Figure 9.25: The rate-distortion performance of various AVQ algorithms for the image sequence using 4-dimensional vectors and a local codebook of 256 codewords. The circled "×" represents the operation point of the nonadaptive vector quantizer for the same data.

The constrained-distortion algorithms, namely the Paul and Wang-Shende-Sayood algorithms, and the sole rate-distortion-based algorithm considered, namely the GTR algorithm, fair substantially better than the constrained-distortion algorithms. We see from Figure 9.25, that each of the Paul, Wang-Shende-Sayood, and GTR algorithms feature nearly equivalent rate-distortion performance for most of the low-distortion, high-rate region of the rate-distortion plane. However, as the distortion gets larger, the rate associated with the constrained-distortion algorithms first decreases, and then start to grow. In fact, as their distortion thresholds go to infinity, both the Wang-Shende-Sayood and Paul algorithms would converge to the nonadaptive vector quantizer were it not for the move-to-front index coder and the codebook-update flags associated with the algorithms. It seems that efficiency of the move-to-front index coder decreases as the codebook updates become fewer, eventually resulting in rate-distortion performance at a point equal in distortion to that of the nonadaptive vector quantizer but at a slightly higher rate. On the other hand, the GTR algorithm allows the rate to continually decrease as the distortion increases. In fact, for $\lambda = \infty$, the performance of the GTR algorithm converges to a rate of $1/N$ bits/sample, where N is the vector dimension, and a distortion equal to the average power of the original source. Consequently, the GTR algorithm features rate-distortion performance superior to that of the other AVQ algorithms for low-rate image coding.

For high-rate coding (a rate of 1.5 bits/pixel or greater), most of the AVQ algorithms have distortion performance significantly better than that of nonadaptive VQ. For example, at a rate equal to that of the nonadaptive vector quantizer (about 1.5 bits/pixel), most of the AVQ algorithms achieve an MSE of around 50, which results in very little visual distinction between their quantized images. However, the AVQ algorithms achieve substantially less distortion than nonadaptive VQ at this rate.

More distinction between the AVQ algorithms is observed for low-rate coding. Particularly, the constrained-rate algorithms were unable to achieve rates below about 1.5 bits/pixel. Of those algorithms that were able to produce a coding at a rate below 1.25 bits/pixel, only the GTR algorithm was able to maintain a monotonic decrease in rate for increasing distortion. As a consequence, GTR was the only algorithm to achieve a coding at a rate less than 1.0 bits/pixel.

In this section, we have presented experimental results for the AVQ of an image sequence. We have noted that the performance of each AVQ algorithm relative to the others varies depending upon which region of the rate-distortion plane we focus our attention. In the next section, we conclude the current chapter by summarizing the performance observed for the algorithms in certain regions of the rate-distortion plane.

9.3 Summary of Experimental Results

In the experimental results presented in Sections 9.1 and 9.2, the rate-distortion performance of various AVQ algorithms was evaluated for two types of data, an artificial nonstationary process and real image-sequence data. In each case, varying algorithm parameters produced a spectrum of rate-distortion performance pairs for each algorithm. In this section, we identify two regions of the rate-distortion plane as being especially important in practical coding applications and summarize the performance observed for the algorithms in these regions.

The two regions of the rate-distortion plane that are of particular interest are the low-rate (LR) region and the low-distortion (LD) region. The first of these regions corresponds to performance yielding a compact representation of the source using few bits. The natural tradeoff between rate and distortion usually implies a high distortion for this region. In data-compression terminology, the LR region corresponds to algorithm performance with a high compression ratio. LR-region performance is of importance in applications whose available rate, or, in information-theory terms, whose channel capacity, is severely limited. As many practical applications, such as network and wireless communications, fall into this category, LR performance is of particular interest in current research.

The second region of interest is the LD region which corresponds to performance yielding a high fidelity between the coded output and the original source. Generally, performance in this region will require a high rate. Although the LR performance of an algorithm is usually considered more important, LD performance is of interest in certain scientific applications that perform measurements and calculations on coded data and, consequently, need to maintain the integrity of the original data source.

In this section, we compare the LR and LD performances observed previously for various AVQ algorithms. As representative of the results of this chapter, we choose Figure 9.11, which plotted the rate-distortion curves of various AVQ algorithms for the Wiener process using a vector dimension of 4 and a local-codebook size of 256 codewords. Figure 9.26 repeats Figure 9.11 with the LR and LD regions explicitly labeled. In Figure 9.26, we identify, somewhat subjectively, the LR region as the lower portion of the plot where the rate-distortion curves are more or less horizontal,

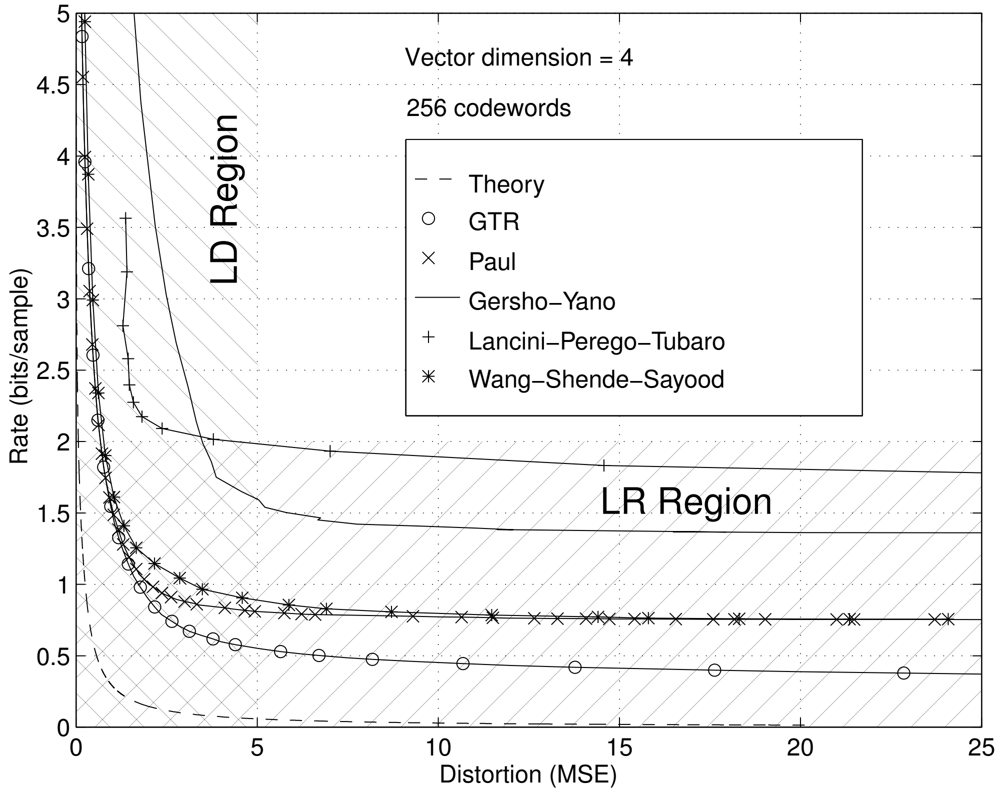


Figure 9.26: The LR and LD regions of the rate-distortion plane. The regions are marked on the plot that previously appeared in Figure 9.11. Rate-distortion curves are for the Wiener process using 4-dimensional vectors and a local-codebook size of 256 codewords.

i.e., the lower half plane where the rate is less than 2.0 bits/sample. Similarly, we identify the LD region as the portion of the plot where the rate-distortion curves are roughly vertical, i.e., the left half plane where the MSE is less than 5.

Figure 9.27 summarizes the LR and LD performance associated with each of the AVQ algorithms. To determine LR performance, we select a value of distortion for which all the rate-distortion curves lie in the LR region in Figure 9.26. We then measure the rate of each algorithm for that value of distortion and plot it in Figure 9.27 as the measure of LR performance. Similarly for LD performance, we choose a rate for which the rate-distortion curves all lie in the LD region and then measure the distortion for each algorithm for that rate.

On examining Figure 9.27, we see that GTR achieves the best performance of all the AVQ algorithms in both the LR and LD regions. In fact, in the LR region the GTR algorithm achieves performance almost 50% better than the next best algorithms (the Paul and Wang-Shende-Sayood algorithms). In the LD region, the GTR algorithm achieves performance nearly equivalent to that of the Paul and Wang-Shende-Sayood algorithms but substantially better than that of the other two algorithms. Although Figure 9.27 was created from only one set of experimental results, performance plots similar to Figure 9.27 corresponding to the remaining results (Figures 9.12, 9.13, and 9.25) would yield similar conclusions.

The cost associated with the superior LR and LD performance of the GTR algorithm is a relatively slower execution speed than that of the other AVQ algorithms. A measure of execution times of the AVQ algorithms is included in Figure 9.27. These times are presented as ratios to the execution time of nonadaptive VQ on the same data. These execution-time figures are intended to give only a rough estimate of

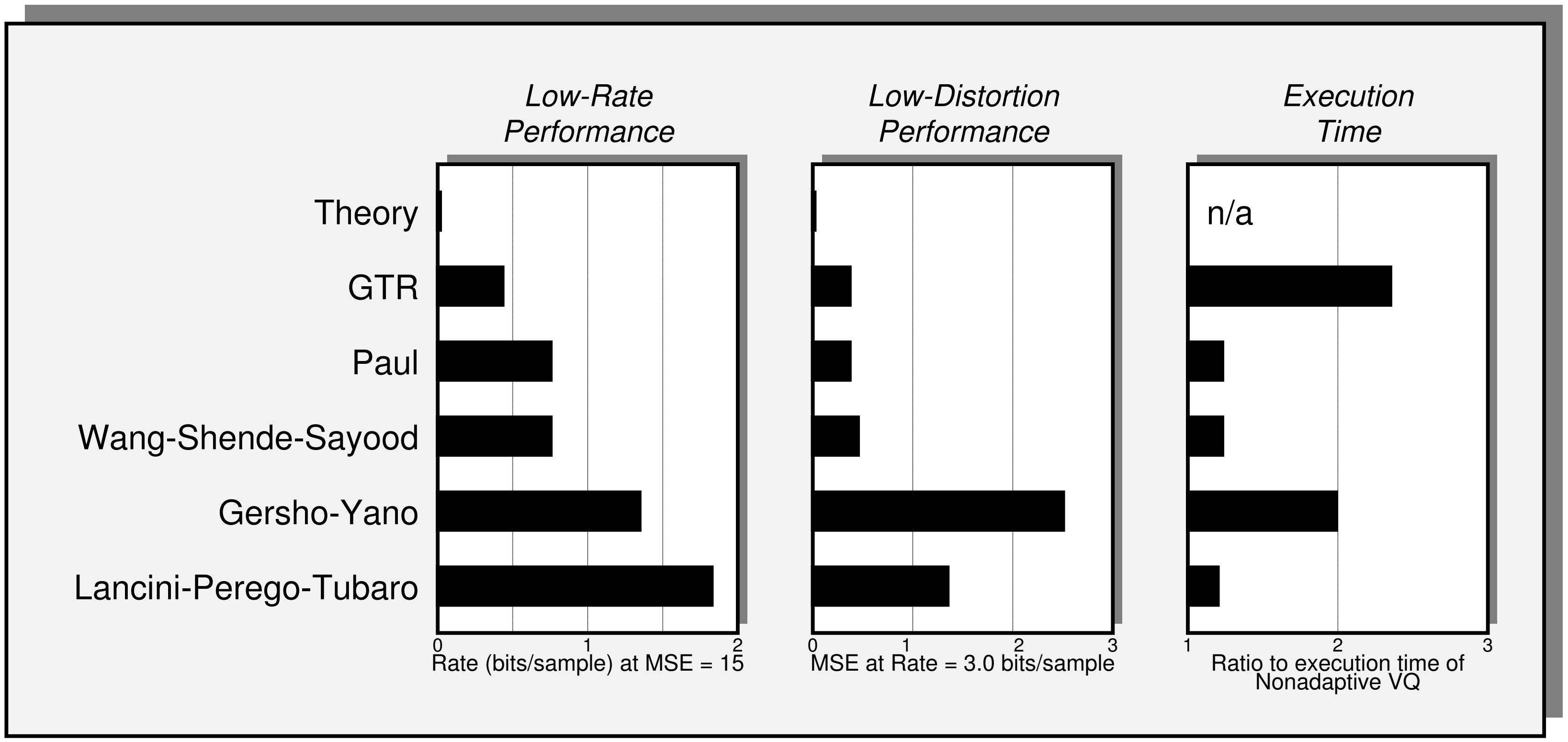


Figure 9.27: Summary of experimental results. For the LR performance, we plot the rates from Figure 9.26 of each algorithm for a MSE distortion of 15 . For the LD performance, we plot the MSE distortions from Figure 9.26 of each algorithm for a rate of 3.0 bits/sample. For the execution time, we plot the time for the algorithm divided by the time for nonadaptive VQ; the parameters of each algorithm were adjusted so to yield performance on the "knee" of the corresponding rate-distortion curve in Figure 9.26. Better performance is indicated by smaller bars for all three plots.

the relative speed of the algorithms as they will vary somewhat for different local-codebook sizes, vector dimensions, and algorithm parameters. Additionally, these times were obtained for the algorithm implementations given in Appendix B, which have not been optimized for execution speed.

The execution-time results presented in Figure 9.27 indicate that nonadaptive VQ is just over twice as fast as the GTR algorithm. The other AVQ algorithms, with the exception of the Gersho-Yano algorithm, require only about one third more execution time than the nonadaptive vector quantizer, and consequently are somewhat faster than the GTR algorithm. The GTR algorithm is slower than the Paul algorithm, which has a similar structure, due to the recalculation of codeword probabilities and codeword lengths which are done for every source vector.

Before leaving this chapter, we make an observation in relation to the AVQ-communication-system model described in Chapter 6. The results of this chapter have been obtained for a codebook coder implemented by a uniform, high-resolution scalar quantizer such that negligible codebook mismatch was observed. We argued in Section 6.2.3 that this simple approach to codebook-coder design is sufficient when the side information accounts for a small part of the total rate of the AVQ system. Regarding Figure 9.3a, we see that, for the GTR algorithm with 4-dimensional vectors, the side information accounts for about 5% of the total rate for most of the LR region. Consequently, we argue that the use of our high-resolution scalar quantizer is justified for LR region operation. However, as seen in Figure 9.3a, the percentage of side information increases sharply with decreasing distortion in the LD region, accounting for a sizeable portion of the total rate there. Consequently, when circumstances call for operation in the LD region, a more sophisticated approach to the design of the codebook coder is warranted. In this case, we should account for the tradeoff, in terms of rate-distortion performance, that exists between the vector coder and the codebook coder. We hypothesize that this same observation could be extended to algorithms other than GTR.

In summary, our results indicate that GTR is the best known AVQ algorithm for coding in the LR region. In addition, GTR is one of three algorithms that achieve equivalently good coding in the LD region. This superior coding performance is achieved with computational complexity somewhat higher than that of other AVQ algorithms. Finally, the high-resolution scalar quantizer used as a codebook coder by the AVQ algorithms is sufficient for operation in the LR region, but more sophisticated design methods incorporating the rate-distortion tradeoff between the codebook coder and the vector coder should be considered for possibly more efficient operation in the LD region. These observations conclude our experimental evaluation of AVQ algorithms. We now continue to Chapter 10, the final chapter of this text, to summarize the discussion presented to this point and to make some concluding remarks.

CHAPTER 10

CONCLUSION

In this dissertation, we have provided an extensive discussion of AVQ, starting with its theoretical underpinnings in information and rate-distortion theories, continuing with the development of a general mathematical model of AVQ communication systems, and concluding with an extensive presentation of experimental results establishing the superiority of AVQ techniques over nonadaptive VQ. We included the development of a new AVQ algorithm, GTR, which was shown to have rate-distortion performance better than that of the other AVQ algorithms. We saw that the GTR algorithm achieved particularly good performance for low-rate coding.

Of the contributions presented here, perhaps the most important is our model of AVQ communication systems. This model allows not only the precise definition of what we consider to be an AVQ algorithm, but also the classification of AVQ algorithms according to the major components in their implementations. In the creation of our taxonomy of AVQ algorithms, we endeavored to isolate the major conceptual distinctions of each algorithm while removing *ad hoc* techniques in order that our experiments evaluated the performance differences due to the theoretically significant portions of each algorithm.

We identified the codebook selector as the most important component of an AVQ system. This component monitors the changing source statistics to select appropriate local codebooks from a large universal codebook. As only the codewords in the local codebook are used to code the source, the composition of the local codebook is paramount to the performance of the algorithm. As a consequence, the design of the codebook selector was central in the development of our AVQ taxonomy.

Our taxonomy identified three major categories of AVQ: constrained-distortion, constrained-rate, and rate-distortion-based algorithms. Of the three, the first two categories can be considered to be online adaptions of the two general types of source coding originating in rate-distortion theory. The third category represents relatively recent developments in AVQ algorithms based on the philosophy that coding should progress under the consideration of both rate and distortion simultaneously.

This third class of algorithms, rate-distortion-based AVQ, incorporates both the rate and the distortion into a cost function. The minimization of this cost function results in both quantities being considered in both the codebook selector and the vector coder. Consequently, the codebook selector of a rate-distortion-based algorithm weighs the potential for a reduction in distortion against the cost in rate for each codeword update, while the vector coder selects the codeword "closest" in the rate-distortion sense.

Our GTR algorithm belongs to this third category of AVQ algorithms. The rate-distortion cost criteria associated with our GTR algorithm enables it to consistently outperform algorithms of the other two categories of AVQ. Performance surpassing that of the other AVQ algorithms was observed in experimental results reported for both an artificial nonstationary process as well as for an image sequence.

We observed that the performance of each AVQ algorithm relative to the others varied depending the particular region of the rate-distortion plane containing the current operating point. Consequently, we identified two regions of the rate-distortion plane, i.e., the low-rate (LR) and low-distortion (LD) regions, as being especially important in practical coding applications. Our results established that, in comparison to other AVQ algorithms, GTR is particularly well suited to LR-coding applications. The GTR algorithm was unrivaled in LR-region performance while its LD performance equaled that attained by the best algorithms in that region.

The performance results reported here indicate that AVQ algorithms have significant potential in source-coding applications. However, several issues remain open for further investigation; we briefly mention a few of these as potential topics for further research. First, all the AVQ algorithms examined here require the specification of one or more parameters before coding begins. However, little is known about how one should should determine suitable values for these parameters for a given application. Consequently, there would be considerable use for techniques that provide online, dynamic estimations for "optimal" parameter values. For example, it would be useful to have an online method of adjusting the rate-distortion parameter λ of the GTR algorithm so as to automatically choose a particular operating point, such as operation in the LR region or on the knee of the rate-distortion curve, for example.

Secondly, it would be useful to have some measure of local stationarity since most AVQ algorithms rely, to some degree, on an assumption of slowly varying source statistics. An estimation of the length of time over which source statistics are approximately stationary would have utility in dynamic parameter-estimation methods as mentioned above. More immediately, it would serve as a good basis for the static setting of the windowing parameter ω in the GTR algorithm as well as the adaption interval τ of the constrained-rate AVQ algorithms.

Thirdly, we recommend a more thorough consideration of the tradeoff involved between the codebook coder and the vector coder. As discussed in Section 9.3, the high-resolution scalar quantizer that we used for the codebook coder of each AVQ algorithm is sufficient for LR coding. However, because of the sizeable amount of side information in the LD region, the investigation of other codebook coders is warranted for systems designed to operate in this region.

Finally, throughout this dissertation the problem at hand was generally one of statistical source coding. That is, given a random process as input, the goal of our communication system was to produce an efficient coding in terms of rate and distortion. However, what is known of real information sources is often more complex than this simple probabilistic random-process model. We have neglected the issue of *source modeling* here. Clearly, for the most efficient operation of a communication system, the source model should exploit all information known about the source. For example, most real sources can be considered to be low-pass in nature; that is, they are mainly composed of low- rather than high-frequency signals. One common approach to handling such sources is to embed a quantizer within a prediction loop. In the case that the quantizer is scalar, this technique is differential pulse-code modulation (DPCM) [65]; for nonadaptive vector quantizers, it has been called vector DPCM [66] or differential VQ (DVQ) [67, 68]. The performance of these algorithms depend on both the design of the predictor (source modeling) as well as the design of the quantizer (source coding).

To facilitate the design of communication systems, it is usually convenient to consider source modeling as an issue separate from source coding. As stated before, this dissertation has concentrated on source coding. The AVQ algorithms that we have examined here are simply source-coding techniques that are not necessarily intended to be stand-alone solutions in real applications. Indeed, we could combine them with various source models; for image coding, such source modeling could consist of motion compensation, transform coding, or DPCM-like predictions, for example. Although it was not our focus in this dissertation, it would be of interest to evaluate the performance of AVQ algorithms in conjunction with these and other source models that are commonly used in practice.

Although we disregard the issue of source modeling, we believe that the performance comparisons reported here are a significant advancement in the field as we know of no prior literature that has attempted to establish the relative rate-distortion performance capabilities of AVQ algorithms. Our results show that the GTR algorithm consistently achieves low-rate-coding performance superior to that of other AVQ algorithms. Since many applications of current interest, such as network and wireless communications, have severe rate limitations, low-rate coding techniques will play a vital role in the successful development of systems for such applications. Our GTR algorithm will be instrumental in the incorporation of AVQ into practical, low-rate coding techniques at the heart of future communication systems designed for these demanding applications.

APPENDIX A

THE CODEBOOK SELECTION PROCESS FOR INFINITE UNIVERSAL CODEBOOKS

In Section 6.1.2, we defined the codebook-selection process, S_t, for the cases of a finite or countably infinite universal codebook, $\mathcal{C}^*$. In this appendix, we present a definition of the same random process using measure theory. This definition is valid for all universal codebooks, finite, countably infinite, or uncountably infinite. We start by introducing some necessary concepts from measure theory.

The following definitions from measure theory are given by Berger (Chapter 7 of [6]). Additionally, Halmos [5] provides an extensive discussion of measure theory. An *abstract alphabet*, A, is a set of finitely, countably, or uncountably many elements. A *σ-algebra*, $\mathcal{A}$, is a collection of subsets of A that is closed under complementation and the formation of countable unions. A *measurable space* is the pair $(A, \mathcal{A})$. A *probability space*, $(\Omega, \mathcal{B}, P)$, is a measurable space coupled with a *probability measure*, P, which is a nonnegative, countably additive function defined on the sets of $\mathcal{B}$ with $P(\Omega) = 1$. We will define *random variable* X as a measurable function from Ω to A and the *probability distribution* of X, p_X, as a probability measure defined on $\mathcal{A}$ such that

$$p_X(Z) = P(\{w \in \Omega : X(w) \in Z\}), \qquad Z \in \mathcal{A}. \tag{A.1}$$

That is, we can consider the probability space $(\Omega, \mathcal{B}, P)$ as the "event space" that drives the random variable X whose values are members of the set A.

Suppose that we have a universal codebook, $\mathcal{C}^*$, that is a finite, countably infinite, or uncountably infinite set. Define A as the power set of $\mathcal{C}^*$; i.e., A is an abstract alphabet that is the class of all subsets of $\mathcal{C}^*$. Let $\mathcal{A}$ be a σ-algebra of subsets of A. Then $(A, \mathcal{A})$ form a measurable space. Suppose we have an event space driving random variable S; i.e., $(\Omega, \mathcal{B}, P)$ is a probability space,

$$S : \Omega \to A, \tag{A.2}$$

and we define a probability measure p_S on $(A, \mathcal{A})$ such that

$$p_S(Z) = P(\{w \in \Omega : S(w) \in Z\}), \qquad Z \in \mathcal{A}. \tag{A.3}$$

Let S_t be a sequence of such random variables. S_t forms a codebook-selection process analogous to the one defined in Section 6.1.2. We note that we define the probability distribution for S_t in A.3 so that

$$p_{S_t}(\{\mathcal{C}_t\}) = \text{prob}\{S_t = \mathcal{C}_t\}, \qquad \mathcal{C}_t \in A, \quad \{\mathcal{C}_t\} \in \mathcal{A} \tag{A.4}$$

for each time t.

The definition of S_t given in this appendix is valid for all types of universal codebooks, finite, countably infinite, or uncountably infinite. Consequently, we can assume the existence of this codebook-selection process for all possible universal codebooks.

APPENDIX B

SOURCE-CODE IMPLEMENTATIONS OF THE AVQ ALGORITHMS

In the following sections, we present the implementations of various AVQ algorithms used to run the results of Chapter 9. The implementations given here are in MATLAB M-file code and have not been optimized for execution speed. In Section B.1, we present some utility routines used by the algorithm implementations. Then, in Sections B.2 through B.8, we present the algorithm implementations themselves.

B.1 Source Code for Various Utility Routines

B.1.1 Competitive Learning

```
function [Y, D] = cl(X, K, epsilon, T, I, t, y)
%
% [Y, D] = cl(X, K, epsilon, T, I, t, y)
%
% X: (k x N), training vectors (N column vectors of dimension k)
% K: codebook size (scalar)
% epsilon: learning rule constant (scalar)
% T: learning rule constant (scalar)
% I: number of iterations to run (scalar)
% t: distortion threshold stopping criterion
% y: (k x K) initial codebook (optional)
%
% Y: codebook (k x K)
% D: sequence of distortions (I x 1)
%
% Implements competitive learning.  The learning rule is specified with
% epsilon and T:
%    y = y + epsilon*exp(-u/T)*(x - y)
% where y is the winning codeword, x is the current input vector, and
% u is the number of times codeword y has been updated.
%
% Either I or t must be specified.  If I is nonzero, then training stops
% after the specified number of iterations.  If I is zero training stops
% when (D(i-1) - D(i))/D(i-1) is less than t, where D(i) is the average
% distortion for the codebook of iteration i.  The sequence of
% distortion values obtained during training is returned in D (only
% if t rather than I is specified).

% k: vector dimension, N = # of training vectors
[k, N] = size(X);

if I == 0
  if exist('t') == 0
    disp('cl: I or t must be specified')
    return
  end
end
```

```
if exist('y') ~= 0
  Y = y;
else
  % Setup random initial codebook
  m1 = max(X(:));
  m2 = min(X(:));
  Y = rand(k, K)*(m1 - m2) + ones(k, K)*((m1 + m2)/2 - (0.5*(m1-m2)));
end

% Normalize to unit cube
X_max = max(X(:));
X_min = min(X(:));
X2 = (X - X_min)./(X_max - X_min);
Y = (Y - X_min)./(X_max - X_min);

partition = zeros(1,N);

u = zeros(1, K);

if I ~= 0
  for iteration = 1:I
    for n = 1:N
      [distortion, partition] = vq(X2(:, n), Y);
      Y(:, partition) = Y(:, partition) + ...
          epsilon*exp(-u(partition)/T)*(X2(:, n) - Y(:, partition));
      u(partition) = u(partition) + 1;
    end
  end
else
  iteration = 1;
  done = 0;

  [distortion, partition] = vq(X2, Y);
  D = mean(distortion);
  if (D == 0)
    done = 1;
  end

  while done ~= 1
    iteration = iteration + 1;
    D = [D(:); 0];
    for n = 1:N
      [distortion, partition] = vq(X2(:, n), Y);
      Y(:, partition) = Y(:, partition) + ...
          epsilon*exp(-u(partition)/T)*(X2(:, n) - Y(:, partition));
      u(partition) = u(partition) + 1;
    end

    [distortion, partition] = vq(X2, Y);
    D(iteration) = mean(distortion);

    if (D(iteration) == 0)
      done =1;
    elseif (abs(D(iteration - 1) - D(iteration)))/D(iteration - 1) < t
      done = 1;
    end
  end

end

% Unnormalize codebook
Y = Y.*(X_max - X_min) + X_min;
D = D.*((X_max - X_min).^2);
```

B.1.2 ECVQ Encoder

```
function [Distortion, Partition] = ecvq(InputProcess, Codebook, ...
    Probs, Lambda);
%
% [Distortion, Partition] = ecvq(InputProcess, Codebook, Probs, Lambda)
%
% InputProcess: (k x N), N k-dimensional vectors to be vector quantized
% Codebook: (k x M), M k-dimensional vectors for codebook
% Probs: (1 x M), codeword probabilities
% Lambda: (scalar), rate-distortion parameter
%
% Distortion: (1 x N), squared error for each input vector
% Partition: (1 x N), the number of the partition to which each input
%                     vector is assigned
%
% Implements the ECVQ encoding algorithm, featuring a modified
% nearest-neighbor rule.
%
% P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-Constrained Vector
%   Quantization," IEEE Trans. Acoust. Speech and Sig. Proc., vol 37,
%   no. 1, Jan. 1989, pages 31-42.

[vector_dimension, num_input_vectors] = size(InputProcess);
codebook_size = size(Codebook, 2);

if (vector_dimension ~= size(Codebook, 1))
  disp('ecvq: Codebook and InputProcess must have same codeword dimension')
  return
end

if (codebook_size ~= size(Probs, 2))
  disp('ecvq: Codebook and Probs must indicate the same codebook size')
  return;
end

Partition = zeros(1, num_input_vectors);
Distortion = zeros(1, num_input_vectors);

% Calculate estimated variable-length codeword lengths for current
% probabilities
const = -1/log(2);
codewordlengths = zeros(1, codebook_size);
P = Probs > 0;
codewordlengths(P) = const*log(Probs(P));
codewordlengths(~P) = Inf*ones(1,sum(~P));

for current_vector = 1:num_input_vectors

  % Get distortion btw each codeword and the current source vector
  [distortion, dummy] = vq(Codebook, InputProcess(:, current_vector));

  % Calculate cost function and choose the codeword with lowest cost
  J = distortion + Lambda*codewordlengths;
  [dummy, winner] = min(J);
  Partition(current_vector) = winner(1);
  Distortion(current_vector) = distortion(winner);
```

```
end
```

B.1.3 ECVQ Training Algorithm

```
function [FinalCodebook, FinalProb, FinalPartition] = ecvq_train(InputData, ...
    InitialCodebook, InitialProb, Lambda, Iterations, StoppingThreshold);
%
% [FinalCodebook, FinalProb, FinalPartition] =
%         ecvq_train(InputData, InitialCodebook, InitialProb, Lambda,
%              Iterations, StoppingThreshold)
%
% InputData: (k x N), input vector process of N k-dimensional vectors
% InitialCodebook: (k x M)
% InitialProb: (1 x M) initial codeword probabilities
% Lambda: (scalar), algorithm parameter controlling the relative weighting
%   of the rate into the minimizing cost function
% Iterations: (scalar)
% StoppingThreshold: (scalar)
%
% FinalCodebook: (k x M')
% FinalProb: (1 x M)
% FinalPartition: (1 x N), final partitions for InputData
%
% This subroutine implements the entropy-constrained vector
% quantization (ECVQ) training algorithm of Chou, Lookabaugh, and Gray.
%
% Either I or t must be specified.  If I is nonzero, then training stops
% after the specified number of iterations.  If I is zero training stops
% when (J(i - 1) - J(i))/J(i) is less than StoppingThreshold, where J(i)
% is the cost function for iteration i.
% Lambda controls the weighting of the rate in the cost function:
% J = E[D + Lambda*R].  This cost function is minimized by the
% algorithm.
%
% Because the ECVQ training algorithm generally reduces the codebook
% size during training, the FinalCodebook will generally be smaller than
% the InitialCodebook, i.e., M' will be less than M.
%
% P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-Constrained Vector
%   Quantization," IEEE Trans. Acoust. Speech and Sig. Proc., vol 37,
%   no. 1, Jan. 1989, pages 31-42.
```

```
%

if (Iterations == 0)
  if (~exist('StoppingThreshold'))
    disp('Either Iterations or StoppingThreshold must be specified')
    return
  end
end

numvectors = size(InputData, 2);

[vector_dimension, codebook_size] = size(InitialCodebook);
codebook = InitialCodebook;

const = -1/log(2);
codewordlengths = zeros(1, codebook_size);
P = InitialProb > 0;
codewordlengths(P) = const*log(InitialProb(P));
% Use infinite codeword lengths for codewords with probability = 0
codewordlengths(~P) = Inf*ones(1,sum(~P));

if (Iterations ~= 0)

  % Iteration-based training
  for iteration = 1:Iterations

    %Determine new partition
    partition = zeros(1, numvectors);
    for vector = 1:numvectors
      % Determine MSE for each partition, note seemingly inverse use of vq()
      [distortion, dummy] = vq(codebook, InputData(:,vector));
      if (Lambda ~= 0)
        cost = distortion + Lambda*codewordlengths;
      else
        cost = distortion;
      end
      [dummy, winner] = min(cost);
      partition(vector) = winner(1);
    end

    cnt = zeros(1, codebook_size);
    keep = ones(1, codebook_size);

    % Update partition centroids
    for codeword = 1:codebook_size
      part = (partition == codeword);
      cnt(codeword) = sum(part);

      % Compute partition centroids
      if (cnt(codeword) > 0)
        if (cnt(codeword) == 1)
          codebook(:,codeword) = InputData(:,part);
        else
          codebook(:,codeword) = mean(InputData(:,part).').';
        end
      else
        % Remove codeword from codebook
        keep(codeword) = 0;
      end
    end

    % Remove unused codewords from codebook
    codebook = codebook(:,keep);
    cnt = cnt(keep);
    codewordlengths = codewordlengths(keep);
    codebook_size = sum(keep);

    % Update codeword lengths
    prob = cnt / size(partition, 2);
    P = (prob > 0);
    codewordlengths(P) = const*log(prob(P));
    % Use infinite codeword lengths for codewords with probability = 0
    codewordlengths(~P) = Inf*ones(1,sum(~P));

  end

else

  % Stopping-threshold-based training
  iteration = 0;
  done = 0;
  previous_J = Inf;
```

```
while (~done)
  iteration = iteration + 1;

  %Determine new partition
  partition = zeros(1, numvectors);
  for vector = 1:numvectors
    % Determine MSE for each partition, note seemingly inverse use of vq()
    [distortion, dummy] = vq(codebook, InputData(:,vector));
    if (Lambda ~= 0)
      cost = distortion + Lambda*codewordlengths;
    else
      cost = distortion;
    end
    [dummy, winner] = min(cost);
    partition(vector) = winner(1);
  end

  cnt = zeros(1, codebook_size);
  keep = ones(1, codebook_size);

  % Update partition centroids
  for codeword = 1:codebook_size
    part = (partition == codeword);
    cnt(codeword) = sum(part);

    % Compute partition centroids
    if (cnt(codeword) > 0)
      if (cnt(codeword) == 1)
        codebook(:,codeword) = InputData(:,part);
      else
        codebook(:,codeword) = mean(InputData(:,part).').';
      end
    else
      keep(codeword) = 0;
    end
  end

  % Remove unused codewords from codebook
  codebook = codebook(:,keep);
  cnt = cnt(keep);
```

```
    codewordlengths = codewordlengths(keep);
    codebook_size = sum(keep);

    % Update codeword lengths
    prob = cnt / size(partition, 2);
    P = (prob > 0);
    codewordlengths(P) = const*log(prob(P));
    % Use infinite codeword lengths for codewords with probability = 0
    codewordlengths(~P) = Inf*ones(1,sum(~P));

    [D, dummy] = vq(InputData, codebook);
    R = entropy(prob);
    J = mean(D) + Lambda*R;
    if (J == 0)
      done = 1;
    else
      if ((previous_J - J)/J < StoppingThreshold)
        done = 1;
      end
      previous_J = J;
    end
  end
end

FinalCodebook = codebook;
FinalProb = prob;
FinalPartition = partition;
```

B.1.4 Entropy

```
function H = entropy(P);
%
% H = entropy(P)
%
% P: (1 x N)
%
% H: scalar
%
% Computes the entropy of the probability mass function given by
% P.
%
```

```
P = P(P>0);
H = -sum(P.*log(P));
H = H./log(2);
```

B.1.5 The Generalized Lloyd Algorithm

```
function [Y, D] = lbg(X, K, I, t, y);
%
% [Y, D] = lbg(X, K, I, t, y)
%
% X: (k x N), training vectors (N column vectors of dimension k)
% K: codebook size (scalar)
% I: number of iterations to run (scalar)
% t: distortion threshold stopping criterion
% y: (k x K) initial codebook (optional)
%
% Y: codebook (k x K)
% D: sequence of distortions (I x 1)
%
% This subroutine implements the LBG algorithm (generalized
% Lloyd algorithm) for VQ design as described in
% Gersho and Gray, Vector Quantization and Signal Compression.
% Kluwer Academic Publishers, 1992.
% If no initial codebook is specified, the initial codebook
% is randomly generated.
%
% Either I or t must be specified.  If I is nonzero, then training stops
% after the specified number of iterations.  If I is zero training stops
% when (D(i-1) - D(i))/D(i-1) is less than t, where D(i) is the average
% distortion for the codebook of iteration i.  The sequence of
% distortion values obtained during training is return in D (only
% if t rather than I is specified).

% k: vector dimension, N = # of training vectors
[k, N] = size(X);

if I == 0
  if exist('t') == 0
    disp('lbg: I or t must be specified')
    return
  end
end

if exist('y') ~= 0
  Y = y;
else
  % Setup random initial codebook
  m1 = max(X(:));
  m2 = min(X(:));
  Y = rand(k, K)*(m1 - m2) + ones(k, K)*((m1 + m2)/2 - (0.5*(m1-m2)));
end

partition = zeros(1,N);

if I ~= 0
  for iteration = 1:I,

    old_Y = Y;

    [distortion, partition] = vq(X, Y);

    for i = 1:K,
      part = (partition == i);
      if (part == 0)
        % random codeword from training set if empty cell
        Y(:, i) = X(:,floor(rand(1)*(N-1)) + 1);
      else
        if (sum(part) <= 1)
          Y(:, i) = X(:,part);
        else
          Y(:, i) = mean(X(:,part).').';
        end
      end
    end

  end

else
  iteration = 1;
  done = 0;
  D = 0;
  [distortion, partition] = vq(X, Y);
  D = mean(distortion);

  while done ~= 1

    iteration = iteration+1;
    D = [D(:); 0];

    old_Y = Y;

    [distortion, partition] = vq(X, Y);

    for i = 1:K,
      part = (partition == i);
      if (part == 0)
        % random codeword from training set if empty cell
        Y(:, i) = X(:,floor(rand(1)*(N-1)) + 1);
      else
        if (sum(part) <= 1)
          Y(:, i) = X(:,part);
        else
          Y(:, i) = mean(X(:,part).').';
        end
      end
    end

    [distortion, partition] = vq(X, Y);
    D(iteration) = mean(distortion);

    change = (D(iteration-1) - D(iteration))/D(iteration-1);
    if  ((change >= 0) & (change < t))
      return
    end
  end
end
```

B.1.6 The D_n Lattice Quantizer

```
function Y = lattice_dn(X)
%
```

```
% Y = lattice_dn(X)
%
% X: (n x N)
%
% Y: (n x N)
%
% Calculates the Dn lattice quantization of each n-dimensional input
% vector given in X.
%
% J. H. Conway and N. J. A. Sloane, "Voronoi Regions of Lattices, Second
%   Moments of Polytopes, and Quantization", IEEE Trans. Info. Theory, vol. 28,
%   no. 2, March 1982, pp. 211-226.
%
% J. H. Conway and N. J. A. Sloane, "Fast Quantizing and Decoding Algorithms
%   for Lattice Quantizers and Codes", IEEE Trans. Info. Theory, vol. 28,
%   no. 2, March 1982, pp. 227-232.
%

vector_dimension = size(X,1);
if (vector_dimension < 4)
  disp('lattice_dn: X must be at least 4-dimensional vectors');
  return
end

N = size(X, 2);

f = zeros(vector_dimension, N);

% Round to nearest integer (except for m + 0.5 should round down to m)
part = (abs(X - fix(X)) == 0.5);
f(~part) = round(X(~part));
f(part) = fix(X(part));

w = abs(f - X);
[dummy, i] = max(w);
g = f;
for n = 1:N
  g(i(n), n) = round(X(i(n), n) - 0.5*sign(f(i(n), n) - X(i(n), n)));
end

select = ones(vector_dimension, 1)*abs(rem(sum(f), 2));
```

```
Y = (~select).*f + (select).*g;
```

B.1.7 Partition Probabilities

```
function P = partitionprob(X, Y);
%
% P = partitionprob(X, Y)
%
% X: (k x N)
% Y: (k x K)
%
% P: (1 x K)
%
% Calculates the probabilities of each of the K VQ partitions
% of codebook Y based on the set of input values X.
%

[k, N] = size(X);
K = size(Y,2);

P = zeros(1, K);

for n = 1:N,
  diff = X(:,n)*ones(1,K) - Y;
  dist = diag(diff.' * diff);
  [mindist,winner] = min(dist);
  P(winner) = P(winner)+1;
end

P = P/N;
```

B.1.8 VQ Encoder

```
function [Distortion, Partition] = vq(InputData, Codebook);
%
% [Distortion, Partition] = vq(InputData, Codebook)
%
% InputData: (k x N), N k-dimensional vectors to be vector quantized
% Codebook: (k x M), M k-dimensional vectors for codebook
%
% Distortion: (1 x N), squared error for each input vector
% Partition: (1 x N), the number of the partition to which each input
%                     vector is assigned
%

[k, N] = size(InputData);
[k2, M] = size(Codebook);

if k ~= k2
  disp('vq: Codebook and InputData must have same codeword dimension')
  return
end

Partition = zeros(1,N);
Distortion = zeros(1,N);
for n=1:N
  diff = InputData(:,n)*ones(1,M) - Codebook;
  if (M < 1024)
    dist = diag(diff.' * diff);
  else
    % Have to save space if codebook is large
    dist = zeros(1, M);
    for codeword = 1:M
      dist(codeword) = diff(:,codeword).' * diff(:,codeword);
    end
  end
  [Distortion(n), winner] = min(dist);
  Partition(n) = winner(1);
end
```

B.2 The Paul Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, OutputProcess] = ...
    paul82(InputProcess, DistThreshold, InitialCodebook, ComponentBits);
%
% [Distortion, Rate, FinalCodebook, OutputProcess] =
%   paul85(InputProcess, DistThreshold, InitialCodebook, ComponentBits)
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% DistThreshold: (scalar), maximum allowable VQ distortion before
%   adaption takes place
% IntialCodebook: (k x M)
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% OutputProcess: (k x N), quantized output process
%
% Implements Paul's 1982 algorithm for adaptive vector quantization.  If the
% current input vector is greater than the specified DistThreshold,
% the current input vector is added to the codebook, replacing the
% old codeword that was least-recently used.
%
% Least-recently-used algorithm is implemented as move-to-front reordering
% of the codebook and the VQ rate is calculated as the entropy of the
% resulting indices.
%
% D. B. Paul, "A 500-800 bps Adaptive Vector Quantization Vocoder Using a
%   Perceptually Motivated Distance Measure", Conference Record, IEEE
%   Globecom, pp. 1079-1082, 1982.
%

[vector_dimension, num_input_vectors] = size(InputProcess);
codebook_size = size(InitialCodebook, 2);
codebook = InitialCodebook;
index_length = log(codebook_size)/log(2);
Distortion = zeros(1, num_input_vectors);
Index = [];
OutputProcess = zeros(vector_dimension, num_input_vectors);
```

```
total_replaced = 0;

for current_vector = 1:num_input_vectors

  [distortion, partition] = vq(InputProcess(:,current_vector), codebook);

  if (distortion > DistThreshold)
    codebook = [ InputProcess(:,current_vector) ...
          codebook(:,1:(codebook_size - 1)) ];
    % Rate equals a flag bit plus the entire new codeword
    total_replaced = total_replaced + 1;
  else
    Distortion(current_vector) = distortion;
    % Rate equals a flag bit plus the VQ index
    Index = [Index partition];
    % Move to front
    codebook = codebook(:,[partition 1:(partition - 1) (partition + ...
          1):(codebook_size)]);
  end
  OutputProcess(:, current_vector) = codebook(:, 1);

end

prob = zeros(1, codebook_size);
for codeword = 1:codebook_size
  prob(codeword) = sum(Index == codeword);
end
prob = prob / size(Index, 2);

total_bits = total_replaced*(1 + vector_dimension*ComponentBits) + ...
    (num_input_vectors - total_replaced)*(1 + entropy(prob));
Rate = total_bits/num_input_vectors/vector_dimension;
FinalCodebook = codebook;
```

B.3 The Wang-Shende-Sayood Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, OutputProcess] = ...
    wss94(InputProcess, DistortionThreshold, InitialCodebook, ComponentBits);
%
% [Distortion, Rate, FinalCodebook, OutputProcess] =
%   wss94(InputProcess, DistortionThreshold, InitialCodebook, ComponentBits)
```

```
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% DistortionThreshold: (scalar), maximum allowable VQ distortion before
%   adaption takes place (squared error)
% IntialCodebook: (k x M)
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% OutputProcess: (k x N), quantized output process
%
% Implements Wang, Shende, and Sayood's 1994 algorithm for AVQ.  The
% universal codebook is the Dn lattice, where n is the vector
% dimension.  The specified DistortionThreshold
% gives the scaling of the lattice so that the maximum squared error of
% any input vector is less than DistortionThreshold.  If the closest
% codeword to the current input vector is greater than DistortionThreshold,
% the closest codeword on the Dn lattice is found and is added to the
% codebook, replacing the least-recently used codeword.
%
% Note: the vector dimension specified by InputProcess and
% InitialCodebook must be at least 4.
%
% Note: Implements the baseline version of the algorithm - the
% scalar-quantized vector-component updating is not done.
%
% X. Wang, S. Shende, K. Sayood, "Online Compression of Video Sequences
%   Using Adaptive VQ Codebooks," Proc. DCC, March 1994, pp. 185-194.
%

[vector_dimension, num_input_vectors] = size(InputProcess);

if (vector_dimension < 4)
  disp('wss94: vector dimension must be at least 4');
  return;
end

codebook_size = size(InitialCodebook, 2);
% Normalize to Dn lattice which has maximum distance =
% sqrt(vector_dimension/4)
covering_radius = sqrt(vector_dimension/4);
normalization = covering_radius/sqrt(DistortionThreshold);
codebook = InitialCodebook*normalization;
InputProcess = InputProcess*normalization;
index_length = log(codebook_size)/log(2);
Distortion = zeros(1, num_input_vectors);
Index = [];
OutputProcess = zeros(vector_dimension, num_input_vectors);
total_vectors_replaced = 0;

for current_vector = 1:num_input_vectors

  [distortion, partition] = vq(InputProcess(:,current_vector), codebook);

  if (distortion > covering_radius)
    % There is a Dn lattice point that is closer
    new_codeword = lattice_dn(InputProcess(:,current_vector));
    codebook = [ new_codeword codebook(:,1:(codebook_size - 1)) ];
    % Rate equals a flag bit plus the entire new codeword
    total_vectors_replaced = total_vectors_replaced + 1;
    diff = InputProcess(:, current_vector) - new_codeword;
    Distortion(current_vector) = diff.' * diff;
  else
    Distortion(current_vector) = distortion;
    % Rate equals a flag bit plus the VQ index
    % Rate equals a flag bit plus the VQ index
    Index = [Index partition];
    % Move to front
    codebook = codebook(:,[partition 1:(partition - 1) (partition + ...
          1):(codebook_size)]);
  end
  OutputProcess(:, current_vector) = codebook(:, 1);
end

prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
```

```
end

total_bits = total_vectors_replaced*(1 + ...
    vector_dimension*ComponentBits) + (num_input_vectors - ...
    total_vectors_replaced)*(1 + entropy(prob));

Rate = total_bits/num_input_vectors/vector_dimension;
FinalCodebook = codebook/normalization;
Distortion = Distortion/normalization/normalization;
```

B.4 The Gersho-Yano Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, OutputProcess] = ...
    gy85(InputProcess, AdaptionInterval, InitialCodebook, ComponentBits);
%
% [Distortion, Rate, FinalCodebook, OutputProcess] =
%       gy85(InputProcess, AdaptionInterval, InitialCodebook, ComponentBits)
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% AdaptionInterval: (scalar), number of vectors per adaption interval
% IntialCodebook: (k x M)
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% OutputProcess: (k x N), quantized output process
%
% Implements Gersho and Yano's 1985 simple algorithm. The partition with
% the greatest partial distortion is split.  An LBG iteration is run to
% refine the locations of the two new codewords.  One codeword replaces
% the split codeword; the other replaces the codeword with the smallest
% partial distortion.
%
% A. Gersho and M. Yano, "Adaptive Vector Quantization by Progressive
%   Codevector Replacement," ICASSP 1985, 133-136.
%

[vector_dimension, num_input_vectors] = size(InputProcess);
codebook_size = size(InitialCodebook, 2);
```

```
num_intervals = floor(num_input_vectors/AdaptionInterval);
codebook = InitialCodebook;
final_distortion = zeros(1, num_intervals*AdaptionInterval);
Index = zeros(1, num_intervals*AdaptionInterval);
OutputProcess = zeros(vector_dimension, num_input_vectors);

for current_interval = 1:num_intervals

  X = InputProcess(:, ((current_interval - 1)*AdaptionInterval + ...
      1):(current_interval*AdaptionInterval));

  [distortion, partition] = vq(X, codebook);
  partitiondistortion = zeros(1,codebook_size);
  partitionprob = zeros(1,codebook_size);
  for p = 1:codebook_size
    part = partition == p;
    if sum(part) ~= 0
      partitiondistortion(p) = mean(distortion(part));
      partitionprob(p) = sum(part)/AdaptionInterval;
    end
  end

  partialdistortion = partitiondistortion .* partitionprob;
  [a,b] = max(partialdistortion);
  winner = b(1);
  [a,b] = min(partialdistortion);
  loser = b(1);

  winners = (partition == winner);

  if (any(winners))
    signalpower = sqrt(max(diag(X(:, winners).' * X(:, winners))));
    split = zeros(vector_dimension, 2);
    split(:,1) = codebook(:,winner) + randn(vector_dimension, ...
        1)*0.0001*signalpower;
    split(:,2) = codebook(:,winner) + randn(vector_dimension, ...
        1)*0.0001*signalpower;
    split2 = lbg(X(:, winners), 2, 1, 0, split);
    codebook(:, winner) = split2(:, 1);
    codebook(:, loser) = split2(:, 2);
    [distortion, partition] = vq(X, codebook);
  end

  final_distortion(((current_interval - 1)*AdaptionInterval + ...
      1):(current_interval*AdaptionInterval)) = distortion;
  Index(((current_interval - 1)*AdaptionInterval + ...
      1):(current_interval*AdaptionInterval)) = partition;
  OutputProcess(:, ((current_interval - 1)*AdaptionInterval + ...
      1):(current_interval*AdaptionInterval)) = codebook(:, partition);

end

Distortion = final_distortion;
FinalCodebook = codebook;

index_length = log(codebook_size)/log(2);

% Variable VQ rate
prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
end
Rate_VQ = entropy(prob);

% Side information involves sending each vector component of the new codeword
% plus the index of its location in the codebook
Rate_Side = 2*(index_length + ComponentBits*vector_dimension)/AdaptionInterval;
Rate = (Rate_VQ + Rate_Side)/vector_dimension;
```

B.5 The Goldberg-Sun Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, OutputProcess] = ...
    gs86(InputProcess, Interval, InitialCodebook, ...
    ReplenishmentThreshold, ComponentBits);
%
% [Distortion, Rate, FinalCodebook, OutputProcess] =
%       gs86(InputProcess, Interval, InitialCodebook,
%            ReplenishmentThreshold, ComponentBits)
%
```

```
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% Interval: (scalar), number of vectors per adaption interval
% IntialCodebook: (k x M)
% ReplenishmentThreshold: (scalar), threshold at which
%   new trained vectors are added to codebook (expressed as a fraction
%   of the maximum vector squared magnitude (signal power)
%   of the adaption interval)
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% OutputProcess: (k x N), quantized output process
%
% Implements Goldberg & Sun's 1986 AVQ algorithm.  One iteration of LBG
% is run using vectors of the adaption interval.  Codewords in the
% new codebook that have changed greater than the specified
% ReplenishmentThreshold replace their corresponding vectors in the
% old codebook.
%
% Note: this implementation is a forward-adaptive version of the algorithm
% "Codebook Replenishment by Mean Shift" as originally described by
% Goldberg & Sun.  That is, coding of the vectors of the adaption
% interval takes place after the codebook is updated, rather than
% afterwards as originally described.
%
% M. Goldberg and H. Sun, "Image Sequence Coding Using Vector Quantization",
%   IEEE Trans. Comm., vol. 34, no. 7, July 1986, pp. 703-710.
%

[vector_dimension, num_input_vectors] = size(InputProcess);
codebook_size = size(InitialCodebook, 2);
num_intervals = floor(num_input_vectors/Interval);
codebook = InitialCodebook;
index_length = log(codebook_size)/log(2);
Distortion = zeros(1, num_intervals*Interval);
Index = zeros(1, num_intervals*Interval);
OutputProcess = zeros(vector_dimension, num_input_vectors);
num_replacements = 0;
```

```
for adaption_interval = 1:num_intervals

  X = InputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval));
  max_dist = max(diag(X.' * X));
  new_codebook = lbg(X, codebook_size, 1, 0, codebook);
  diff = new_codebook - codebook;
  distance = diag(diff.' * diff);
  replace = (distance > ReplenishmentThreshold*max_dist);
  codebook(:, replace) = new_codebook(:, replace);
  num_replacements = num_replacements + sum(replace);

  [distortion, partition] = vq(X, codebook);

  Distortion(((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = distortion;
  Index(((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = partition;
  OutputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = codebook(:, partition);
end

FinalCodebook = codebook;

% Variable-rate VQ
prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
end
Rate_VQ = entropy(prob);

% Side information involves sending each vector component of the new codeword
% plus the index of its location in the codebook, for each codeword added
% to codebook
Rate_Side = num_replacements*(index_length + ...
    vector_dimension*ComponentBits)/(num_intervals*Interval);

Rate = (Rate_VQ + Rate_Side)/vector_dimension;
```

B.6 The Lancini-Perego-Tubaro Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, OutputProcess] = ...
    lpt92(InputProcess, Interval, InitialCodebook, NewCodebookSize, ...
    ReplacementNum, ComponentBits);
%
% [Distortion, Rate, FinalCodebook, OutputProcess] =
%       lpt92(InputProcess, Interval, InitialCodebook,
%             NewCodebookSize, ReplacementNum, ComponentBits)
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% Interval: (scalar), number of vectors per adaption interval
% InitialCodebook: (k x M), M k-dimensional codewords
% NewCodebookSize: (scalar), number of codewords to train during each
%      adaption interval
% ReplacementNum: (scalar), number of local codewords to add to codebook
%      during each adaption interval (ReplacementNum <= NewCodebookSize)
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate of encoding, bits per sample
% FinalCodebook: (k x M), M k-dimensional codewords
% OutputProcess: (k x N), quantized output process
%
% During each adaption interval, a local codebook of NewCodebookSize
% codewords is trained using competitive learning with random initialization
% from the training set.  The ReplacementNum most significant codewords
% from the local codebook replace the ReplacementNum least frequently used
% codewords in the codebook, and then the vectors of the adaption interval
% are quantized.
%
% R. Lancini, F. Perego, and S. Tubaro, "Neural Network Approach for
%   Adaptive Vector Quantization of Images," ICASSP 1992, 389-392.
%

if (ReplacementNum > NewCodebookSize)
  disp('lancini_92: ReplacementNum must be no more than NewCodebookSize')
  return
end
```

```
[vector_dimension,N] = size(InputProcess);
codebook_size = size(InitialCodebook, 2);
num_intervals = floor(N/Interval);
codebook = InitialCodebook;
Distortion = zeros(1, num_intervals*Interval);
Index = zeros(1, num_intervals*Interval);
OutputProcess = zeros(vector_dimension, num_input_vectors);
num_used = zeros(1, codebook_size);

for adaption_interval = 1:num_intervals

  X = InputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval));

  % Training set is either whole adaption interval or 10 times size of
  % local codebook.
  training_set_size = min([size(X, 2) NewCodebookSize*10]);
  if (training_set_size  < NewCodebookSize)
    disp('lancini_92: Adaption interval too small for training local codebook')
    return
  end
  % Choose training vectors randomly from adaption interval
  training_set = X(:, rem(round(rand(1,training_set_size)*1000), ...
      training_set_size)+1);

  % Initialization of CL: random vectors from training set
  initialcodebook = training_set(:, ...
      rem(round(rand(1,NewCodebookSize)*1000), NewCodebookSize)+1);
  [new_local_codebook, distortion] = cl(training_set, ...
      NewCodebookSize, 0.2, 100, 0, 0.05, initialcodebook);

  % Get indices of least frequently used codewords in old codebook
  [dummy, lfu] = sort(num_used);
  lfu = lfu(1:ReplacementNum);

  % Find closest codewords in old codebook to codewords in local codebook
  [distortion, partition] = vq(new_local_codebook, codebook);
  [distortion, index] = sort(distortion);
  % Choose local codewords that are farthest away from corresponding old
  % codewords.
  index = index(size(index, 2):-1:1);
  codebook(:, lfu) = new_local_codebook(:, index(1:ReplacementNum));
  num_used(lfu) = zeros(1, ReplacementNum);

  [distortion, partition] = vq(X, codebook);
  for j = 1:size(partition, 2)
    num_used(partition(j)) = num_used(partition(j)) + 1;
  end

  Distortion(((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = distortion;
  Index(((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = partition;
  OutputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = codebook(:, partition);
end

FinalCodebook = codebook;

index_length = log(codebook_size)/log(2);

% Variable-rate VQ
prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
end
Rate_VQ = entropy(prob);

% Side information requires only sending vector components as decoder knows
% which ones are replaced due to past least-frequently-used statistics
Rate_Side = ReplacementNum*(ComponentBits*vector_dimension)/Interval;
Rate = (Rate_VQ + Rate_Side)/vector_dimension;
```

B.7 The Lightstone-Mitra Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, FinalProb, OutputProcess] = ...
    lm96(InputProcess, Interval,  Lambda, InitialCodebook, ...
    InitialProb, ComponentBits, Iterations);
%
```

```
% [Distortion, Rate, FinalCodebook, FinalProb, OutputProcess] =
%         lm96(InputProcess, Interval, Lambda, InitialCodebook,
%              InitialProb, ComponentBits)
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% Interval: (scalar), number of vectors per adaption interval
% Lambda: (scalar), algorithm parameter, specifies weighting of rate
%         into adaption cost function
% InitialCodebook: (k x M)
% InitialProbabilties: (1 x M) initial codeword probabilties
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
% Iterations: (scalar) (optional) default = 1, number of iterations of
%   algorithm to run over each adaption interval
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% FinalProb: (1 x M) final codeword probabilties
% OutputProcess: (1 x N), quantized output process
%
% Implements Lightstone and Mitra's AVQ algorithm.  Uses entropy-
% constrained VQ (ECVQ) to update the codebook for each adaption interval.
% InitialProb determine the initial variable-length code for the
% InitialCodebook.  New VQ centroids are training using one iteration
% of ECVQ (Chou, Lookabaugh, Gray 1989).  New centroids are added to
% codebook based on tradeoff between improvement in distortion and
% cost of side-information rate.  Lambda controls the rate-distortion
% tradeoff in training and in codeword updating.  Iterations (default = 1)
% controls the number of ECVQ-training followed by codeword-updating
% iterations.
%
% M. Lightstone and S. K. Mitra, "Image-Adaptive Vector Quantization in an
%   Entropy-Constrained Framework," IEEE Trans. Image Proc., to appear 1996.
%

[vector_dimension, num_input_vectors] = size(InputProcess);
num_intervals = floor(num_input_vectors/Interval);
codebook_size = size(InitialCodebook, 2);
Distortion = zeros(1, num_input_vectors);
Index = zeros(1, num_input_vectors);
```

```
OutputProcess = zeros(vector_dimension, num_input_vectors);

if (~exist('Iterations'))
  Iterations = 1;
end

codebook = InitialCodebook;
prob = InitialProb;
num_replaced = 0;

for adaption_interval = 1:num_intervals

  X = InputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval));

  codewords_updated = zeros(1, codebook_size);

  for iteration = 1:Iterations

    [centroids, prob2, partition] = ecvq_train(X, codebook, prob, Lambda, 1);
    cnt = zeros(1, codebook_size);
    centroid_index = 0;

    for codeword = 1:codebook_size
      part = partition == codeword;
      cnt(codeword) = sum(part);
      if (cnt(codeword) > 0)
        centroid_index = centroid_index + 1;
        prob(codeword) = prob2(centroid_index);
        if (cnt(codeword) == 1)
          d1 = 0;
        else
          [d1, dummy] = vq(X(:,part), centroids(:,centroid_index));
          d1 = mean(d1);
        end
        [d2, dummy] = vq(X(:,part), codebook(:,codeword));
        d2 = mean(d2);
        delta_d = d1 - d2;
        delta_r = ComponentBits*vector_dimension/cnt(codeword);
        delta_j = delta_d + Lambda*delta_r;
        if (delta_j < 0)
```

```
          codewords_updated(codeword) = 1;
          codebook(:,codeword) = centroids(:,centroid_index);
        end
      else
        prob(codeword) = 0;
      end
    end

  end

  num_replaced = num_replaced + sum(codewords_updated);
  [distortion, partition] = ecvq(X, codebook, prob, Lambda);
  Distortion(((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = distortion;
  Index (((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = partition;
  OutputProcess(:, ((adaption_interval - 1)*Interval + ...
      1):(adaption_interval*Interval)) = codebook(:, partition);
end

FinalCodebook = codebook;
FinalProb = prob;

index_length = log(codebook_size)/log(2);

% Variable-rate VQ
prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
end
Rate_VQ = entropy(prob);

Rate_Side = num_replaced*(vector_dimension*ComponentBits + ...
    index_length)/num_input_vectors;
Rate = (Rate_VQ + Rate_Side)/vector_dimension;
```

B.8 The GTR Algorithm Source Code

```
function [Distortion, Rate, FinalCodebook, FinalProb, OutputProcess] = ...
    gtr(InputProcess, Lambda, ProbWindow, InitialCodebook, InitialProb, ...
        ComponentBits);
%
% [Distortion, Rate, FinalCodebook, FinalProb, OutputProcess] =
%   gtr(InputProcess, Lambda, ProbWindow, InitialCodebook,
%       InitialProb, ComponentBits)
%
% InputProcess: (k x N), input vector process of N k-dimensional vectors
% Lambda: (scalar), rate-distortion parameter
% ProbWindow: (scalar), window for time-averaging of probability estimates
% InitialCodebook: (k x M)
% InitialProb: (1 x M), initial probabilities for each partition
%   region
% ComponentBits: (scalar), number of bits for each codeword vector
%   component used in calculating the side information rate
%
% Distortion: (1 x N), squared error for each input vector
% Rate: (scalar), average rate for encoding of process (bits per sample)
% FinalCodebook: (k x M), M k-dimensional vectors
% FinalProb: (1 x M), final codeword probabilities
% OutputProcess: (k x N), quantized output
%
% Implements the generalized threshold replenishment algorithm (GTR)
% for AVQ.  Uses a move-to-front codebook reordering strategy.

[vector_dimension, num_input_vectors] = size(InputProcess);
codebook_size = size(InitialCodebook, 2);

codebook = InitialCodebook;
codeprob = InitialProb;

FinalDistortion = zeros(1, num_input_vectors);
Index = [];
total_replaced = 0;
const = -1/log(2);
OutputProcess = zeros(vector_dimension, num_input_vectors);

for current_vector = 1:num_input_vectors
```

```
[distortion, winner] = ecvq(InputProcess(:, current_vector), ...
    codebook, codeprob, Lambda);

% Estimate number of vectors in window that map to each partition
cnt = ProbWindow*codeprob;
% Increment winning partition
newcnt = cnt(winner) + 1;

delta_d = -distortion;
delta_r = vector_dimension*ComponentBits;

% Change in cost function to to update
delta_J = delta_d + Lambda*delta_r;

if (delta_J < 0)
  % Replace LRU codeword (use move-to-front) with current
  % source vector
  codebook = [InputProcess(:, current_vector) codebook(:, ...
        1:(codebook_size - 1))];
  % Split count between new codeword and winning codeword
  if (winner ~= codebook_size)
    cnt(winner) = newcnt/2;
  else
    cnt(winner) = newcnt;
  end
  cnt = [cnt(winner) cnt(1:(codebook_size - 1))];
  codeprob = cnt/sum(cnt);
  total_replaced = total_replaced + 1;
  FinalDistortion(current_vector) = 0;
else
  % No codeword update, just move winner to front of codebook
  reorder_indices = [winner 1:(winner - 1) (winner + 1):codebook_size];
  codebook = codebook(:, reorder_indices);
  cnt(winner) = newcnt;
  codeprob = cnt(reorder_indices)/sum(cnt);
  FinalDistortion(current_vector) = distortion;
  Index = [Index winner];
end

OutputProcess(:, current_vector) = codebook(:, 1);
```

```
end

Distortion = FinalDistortion;
FinalProb = codeprob;

prob = zeros(1, codebook_size);
if (size(Index, 2) > 0)
  for codeword = 1:codebook_size
    prob(codeword) = sum(Index == codeword);
  end
  prob = prob / size(Index, 2);
end

total_bits = total_replaced*(1 + vector_dimension*ComponentBits) + ...
    (num_input_vectors - total_replaced)*(1 + entropy(prob));

Rate = total_bits/num_input_vectors/vector_dimension;
FinalCodebook = codebook;
```

BIBLIOGRAPHY

[1] P. F. Swaszek, “Quantization for Signal Detection,” in *Advances in Statistical Signal Processing*, vol. 2, pp. 239–263, Greenwich, CT: JAI Press Inc., 1993.

[2] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Kluwer international series in engineering and computer science, Norwell, MA: Kluwer Academic Publishers, 1992.

[3] R. M. Gray, *Source Coding Theory.* Kluwer International Series in Engineering and Computer Science, Norwell, MA: Kluwer Academic Publishers, 1990.

[4] A. Papoulis, *Probability, Random Variables, and Stochastic Processes.* New York: McGraw-Hill, third ed., 1991.

[5] P. R. Halmos, *Measure Theory.* Princeton, NJ: Van Nostrand, 1950.

[6] T. Berger, *Rate Distortion Theory.* Englewood Cliffs, NJ: Prentice-Hall, 1971.

[7] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication.* Urbana, IL: University of Illinois Press, 1949.

[8] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley Series in Telecommunications, New York: John Wiley & Sons, Inc., 1991.

[9] R. M. Gray, *Entropy and Information Theory.* New York: Springer-Verlag, 1990.

[10] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the IRE*, vol. 40, pp. 1098–1101, September 1952.

[11] R. N. Williams, *Adaptive Data Compression.* Kluwer International Series in Engineering and Computer Science, Boston: Kluwer Academic Publishers, 1991.

[12] G. G. Langdon, Jr., “An Introduction to Arithmetic Coding,” *IBM Journal of Research and Development*, vol. 28, pp. 135–149, March 1984.

[13] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[14] J. Ziv and A. Lempel, "Compression of Individual Sequences Via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[15] C. E. Shannon, "Coding Theorems for a Discrete Source with a Fidelity Criterion," in *Information and Decision Processes* (R. E. Machol, ed.), pp. 93–126, New York: McGraw Hill, Inc., 1960.

[16] R. M. Gray, "Information Rates of Autoregressive Processes," *IEEE Transactions on Information Theory*, vol. 16, pp. 412–421, July 1970.

[17] J. Makhoul, S. Roucos, and H. Gish, "Vector Quantization in Speech Coding," *Proceedings of the IEEE*, vol. 73, pp. 1551–1588, November 1985.

[18] T. Berger, "Information Rates of Wiener Processes," *IEEE Transactions on Information Theory*, vol. 16, pp. 134–139, March 1970.

[19] R. M. Gray and L. D. Davisson, "Source Coding Theorems Without the Ergodic Assumption," *IEEE Transactions on Information Theory*, vol. 20, pp. 502–516, July 1974.

[20] P. C. Shields, D. L. Neuhoff, L. D. Davisson, and F. Ledrappier, "The Distortion-Rate Function for Nonergodic Sources," *The Annals of Probability*, vol. 6, no. 1, pp. 138–143, 1978.

[21] A. Leon-Garcia, L. D. Davisson, and D. L. Neuhoff, "New Results on Coding of Stationary Nonergodic Sources," *IEEE Transactions on Information Theory*, vol. 25, pp. 137–144, March 1979.

[22] J. C. Kieffer, "A Survey of the Theory of Source Coding with a Fidelity Criterion," *IEEE Transactions on Information Theory*, vol. 39, pp. 1473–1490, September 1993.

[23] R. M. Gray and F. Saadat, "Block Source Coding Theory for Asymptotically Mean Stationary Sources," *IEEE Transactions on Information Theory*, vol. 30, pp. 54–68, January 1984.

[24] T. Hashimoto and S. Arimoto, "On the Rate-Distortion Function for the Nonstationary Gaussian Autoregressive Process," *IEEE Transactions on Information Theory*, vol. 26, pp. 478–480, July 1980.

[25] J. Ziv, "Distortion-Rate Theory for Individual Sequences," *IEEE Transactions on Information Theory*, vol. 26, pp. 137–143, March 1980.

[26] J. C. Kieffer, “Fixed-Rate Encoding of Nonstationary Information Sources,” *IEEE Transactions on Information Theory*, vol. 33, pp. 651–655, September 1987.

[27] R. M. Gray and L. D. Davisson, *Random Processes.* Englewood Cliffs: Prentice-Hall, 1986.

[28] P. L. Zador, “Asymptotic Quantization Error of Continuous Signals and the Quantization Distortion,” *IEEE Transactions on Information Theory*, vol. 28, pp. 139–149, March 1982.

[29] A. Gersho, “Asymptotically Optimal Block Quantization,” *IEEE Transactions on Information Theory*, vol. IT-25, pp. 373–380, July 1979.

[30] P. A. Chou, T. Lookabaugh, and R. M. Gray, “Entropy-Constrained Vector Quantization,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 31–42, January 1989.

[31] M. R. Garey, D. S. Johnson, and H. S. Witsenhausen, “The Complexity of the Generalized Lloyd-Max Problem,” *IEEE Transactions on Information Theory*, vol. 28, pp. 255–256, March 1982.

[32] S. P. Lloyd, “Least-square quantization in pcm,” *IEEE Transactions on Information Theory*, vol. 28, pp. 129–137, March 1982. originally an unpublished Bell Laboratories Technical note (1957).

[33] Y. Linde, A. Buzo, and R. M. Gray, “An Algorithm for Vector Quantizer Design,” *IEEE Transactions on Communications*, vol. 28, pp. 84–95, January 1980.

[34] T. Kohonen, *Self–Organization and Associative Memory.* Berlin: Springer-Verlag, 1984.

[35] S. C. Ahalt, A. K. Krishnamurthy, P. Chen, and D. E. Melton, “Competitive Learning Algorithms for Vector Quantization,” *Neural Networks*, vol. 3, no. 3, pp. 277–290, 1990.

[36] S. Grossberg, “Adaptive Pattern Classification and Universal Recoding: I. Parallel Development and Coding of Neural Feature Detectors,” *Biological Cybernetics*, vol. 23, pp. 121–134, 1976.

[37] N. Ueda and R. Nakano, “A New Competitive Learning Approach Based on an Equidistortion Principle for Designing Optimal Vector Quantizers,” *Neural Networks*, vol. 7, no. 8, pp. 1211–1227, 1994.

[38] A. K. Krishnamurthy, S. C. Ahalt, D. Melton, and P. Chen, "Neural Networks for Vector Quantization of Speech and Images," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 1449–1457, October 1990.

[39] S. C. Ahalt, P. Chen, and A. K. Krishnamurthy, "Performance Analysis of Two Image Vector Quantization Techniques," in *Proceedings of the International Joint Conference on Neural Networks*, vol. I, (Washington, D.C.), pp. 169–175, June 18–22, 1989.

[40] D. DeSieno, "Adding a Conscience to Competitive Learning," in *Proceedings of the IEEE International Conference on Neural Networks*, vol. I, pp. 117–124, July 1988.

[41] R. Hecht-Nielsen, "Applications of Counterpropagation Networks," *Neural Networks*, vol. 1, no. 2, pp. 131–139, 1988.

[42] Z. Zhang and V. K. Wei, "An On-Line Universal Lossy Data Compression Algorithm via Continuous Codebook Refinement—Part I: Basic Results," *IEEE Transactions on Information Theory*, vol. 42, pp. 803–821, May 1996.

[43] D. L. Neuhoff, R. M. Gray, and L. D. Davisson, "Fixed Rate Universal Block Source Coding with a Fidelity Criterion," *IEEE Transactions on Information Theory*, vol. 21, pp. 511–523, September 1975.

[44] N. M. Nasrabadi and Y. Feng, "A Dynamic Finite-State Vector Quantization Scheme," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, (Albuquerque, New Mexico), pp. 2261–2264, April 1990.

[45] K. Zeger, A. Bist, and T. Linder, "Universal Source Coding with Codebook Transmission," *IEEE Transactions on Communications*, vol. 42, pp. 336–346, February/March/April 1994.

[46] M. Lightstone and S. K. Mitra, "Image-Adaptive Vector Quantization in an Entropy-Constrained Framework," *IEEE Transactions on Image Processing*, 1996. to appear.

[47] T. P. Bizon, M. J. Shalkhauser, and W. A. Whyte, Jr., "Real-Time Transmission of Digital Video Using Variable-Length Coding," Tech. Rep. 106092, NASA Lewis Research Center, Cleveland, OH, 1993.

[48] X. Wang, S. Shende, and K. Sayood, "Online Compression of Video Sequences Using Adaptive VQ Codebooks," in *Proceedings of the IEEE Data Compression Conference* (J. A. Storer and M. Cohn, eds.), (Snowbird, UT), pp. 185–194, IEEE Computer Society Press, 1994.

[49] R. Lancini, F. Perego, and S. Tubaro, "Neural Network Approach for Adaptive Vector Quantization of Images," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, (San Francisco, CA), pp. 389–392, March 1992.

[50] P. A. Chou and M. Effros, "Rate and Distortion Redundancies for Source Coding with Respect to a Fidelity Criterion," in *Proceedings of the IEEE International Symposium on Information Theory*, (San Antonio, TX), January 1993.

[51] M. Lightstone and S. K. Mitra, "Adaptive Vector Quantization for Image Coding in an Entropy-Constrained Framework," in *Proceedings of the International Conference on Image Processing*, vol. 1, (Austin, TX), pp. 618–622, November 1994.

[52] D. B. Paul, "A 500-800 bps Adaptive Vector Quantization Vocoder Using a Perceptually Motivated Distance Measure," in *Conference Record, IEEE Globecom*, pp. 1079–1082, 1982.

[53] O. T.-C. Chen, B. J. Sheu, and Z. Zhang, "An Adaptive Vector Quantizer Based on the Gold-Washing Method for Image Compression," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 4, pp. 143–157, April 1994.

[54] R. M. Goodman, B. Gupta, and M. Sayano, "Neural Network Implementation of Adaptive Vector Quantization for Image Compression," tech. rep., Department of Electrical Engineering, California Institute of Technology, Pasadena, CA, 1991.

[55] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM*, vol. 29, pp. 320–330, April 1986.

[56] J. H. Conway and N. J. A. Sloane, "Fast Quantizing and Decoding Algorithms for Lattice Quantizers and Codes," *IEEE Transactions on Information Theory*, vol. 28, pp. 227–232, March 1982.

[57] J. H. Conway and N. J. A. Sloane, "Voronoi Regions of Lattices, Second Moments of Polytopes, and Quantization," *IEEE Transactions on Information Theory*, vol. 28, pp. 211–226, March 1982.

[58] R.-F. Chang, W.-T. Chen, and J.-S. Wang, "Image Sequence Coding Using Adaptive Tree-Structured Vector Quantization with Multipath Searching," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, (Toronto, Canada), pp. 2281–2284, May 1991.

[59] M. Goldberg and H. Sun, "Frame Adaptive Vector Quantization for Image Sequence Coding," *IEEE Transactions on Communications*, vol. 36, pp. 629–635, May 1988.

[60] A. Gersho and M. Yano, “Adaptive Vector Quantization by Progressive Codevector Replacement,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 133–136, May 1985.

[61] M. Goldberg and H. Sun, “Image Sequence Coding Using Vector Quantization,” *IEEE Transactions on Communications*, vol. COM-34, pp. 703–710, July 1986.

[62] W.-C. Fang, B. J. Sheu, O. T.-C. Chen, and J. Choi, “A VLSI Neural Processor for Image Data Compression Using Self-Organization Networks,” *IEEE Transactions on Neural Networks*, vol. 3, pp. 506–518, May 1992.

[63] O. T.-C. Chen, B. J. Sheu, and W.-C. Fang, “Image Compression Using Self-Organization Networks,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 4, pp. 480–489, October 1994.

[64] T.-C. Lee and A. M. Peterson, “Adaptive Vector Quantization Using a Self-Development Neural Network,” *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 1458–1471, October 1990.

[65] A. K. Jain, *Fundamentals of Digital Image Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[66] C. W. Rutledge, “Vector DPCM: Vector Predictive Coding of Color Images,” in *Proceedings of the IEEE Global Telecommunications Conference*, pp. 1158–1164, September 1986.

[67] J. E. Fowler, M. R. Carbonara, and S. C. Ahalt, “Image Coding Using Differential Vector Quantization,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, pp. 350–367, October 1993.

[68] J. E. Fowler, K. C. Adkins, S. B. Bibyk, and S. C. Ahalt, “Real-Time Video Compression Using Differential Vector Quantization,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 14–24, February 1995.