FAST ALGORITHMS FOR VECTOR QUANTIZATION PICTURE CODING

by

WILLIAM HOWARD EQUITZ

B.S. Mathematics, Massachusetts Institute of Technology
(1983)

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1984

Signature of Author: _____
Department of Electrical Engineering and Computer Science, May 4, 1984

Certified by: _____
Dr. Jae S. Lim, MIT

Certified by: _____
Dr. Scott C. Knauer, AT&T Bell Laboratories

Accepted by: _____
Dr. Arthur C. Smith

FAST ALGORITHMS FOR VECTOR QUANTIZATION PICTURE CODING

by

WILLIAM HOWARD EQUITZ

Submitted to the Department of Electrical Engineering and Computer Science
on May 4, 1984 in partial fulfillment of the
requirements for the Degree of Master of Science in
Electrical Engineering

ABSTRACT

This research involves improving the method by which pictures can be coded using vector quantization. Section 2 will detail the background of vector quantization, and describe previous work in which vector quantization was used for speech and picture coding; section 3 will describe theoretical advances made during the investigation of this topic; section 4 will describe experimental results performed in the context of vector quantization picture coding; and section 5, conclusions.

Theoretical advances made include the following. A multi-dimensional data structure (k-d trees, developed by Bentley) was determined to be an appropriate way to allow searching in logarithmic time. Various methods of reducing vector dimensionality to improve search speed were investigated. Several implementations of a novel "clustering" algorithm were developed to derive a codebook from a set of training data in a fraction of the time previously required, without sacrificing performance.

Thesis Supervisor (Academic): Dr. Jae S. Lim

Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor (AT&T Bell Laboratories): Dr. Scott C. Knauer

Title: Head, Digital Architectures Research Department

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Picture coding is the process of reducing the amount of information needed to transmit and reproduce a picture with acceptable quality. Traditionally, attempts to reduce picture redundancy have centered around the elimination of interpixel repetition, either within a still picture, or between corresponding elements of the neighboring frames of a moving picture. The only form of picture coding to date to deal with groups of picture elements (pixels or pels) has been transform coding, and this method ends up dealing with scalars when doing quantization. Recently, however, it has proven attractive to look at groups of pixels (say, 4x4 blocks) as a discrete entity. Treating these blocks, or pixel vectors, as a unit, and trying to optimally describe these blocks of pixels in an efficient manner is the basis for vector quantization. In vector quantization, the pixel vectors are quantized as *vectors* as opposed to more traditional scalar quantization schemes.

Although the algorithmic improvements described in this thesis are relevant to any vector quantization application, this thesis will be written in the context of improving the method by which pictures can be coded using vector quantization. The basic idea is to develop a set (or codebook) of pixel blocks which are somehow representative of the blocks likely to occur in the picture to be transmitted. The vector quantization coder then examines the picture to be transmitted and tries to match up the blocks of pixels occurring in the picture with the closest match in a set of "quantized" pixel blocks (the "codebook") developed ahead of time. Picture transmission is then simply a process of transmitting the identity of the codebook block which was deemed to most closely match the pixel block from the original picture. Decoding is a simple matter of reconstructing the quantized picture by piecing together the blocks whose identity has been transmitted.

This thesis is organized as follows. Section 2 will detail the background of vector quantization, and describe previous work in which vector quantization was used for speech and picture coding; section 3 will describe theoretical advances made during the investigation of this topic; section 4

will describe experimental results; and section 5, conclusions.

Theoretical advances made include the following. A multi-dimensional data structure (k-d trees, developed by Bentley) was determined to be an appropriate way to allow searching in logarithmic time. Various methods of reducing vector dimensionality to improve search speed were investigated. Several implementations of a novel "clustering" algorithm were developed to derive a codebook from a set of training data in a fraction of the time previously required, without sacrificing performance.

2. *VECTOR QUANTIZATION*

2.1 Theory

2.1.1 General

2.1.1.1 Description

Vector quantization (VQ) is a process by which blocks of information are coded as a group in such a way as to minimize the expected error resulting from the coding. The idea is to identify a set of possible blocks of data (vectors) which might in some way be representative of the information to be transmitted. The blocks to be quantized are then matched against the code vectors to find the closest matching block among the code vectors. The vector is thus "quantized" to match a particular preset code vector, and the information is coded by giving the identity of the code vector the block was found to be the most similar to. The "trick" to this method, of course, is to have a good set of representative codes, typical of the data to be sent.

A form of two-dimensional vector quantization would occur when one tried to describe in what part of the country one's friends lived. For this application, quantization by cities might be enough. Rather than say that the exact coordinates of where Paul lived were 33.7°N 97.1°W, and that Alfred lived at 29.9°N 95.3°W, one might feel that it was enough to say they lived in Dallas and Houston, respectively. Naturally, this loses some precision, but it requires less information to be processed while giving all the essential data. Similarly, if even more coarse quantization was acceptable, one might say they both lived in Texas, or even "somewhere in the south". One would try to tailor the quantization to the data needing representation.

In more formal mathematical language, one could describe the vector quantization process as follows. If x was an arbitrary block (vector) of data to be quantized by encoder Ξ, then $\Xi(x) = C_j$ would represent quantization of x to the j^{th} value of the quantization alphabet C. For coding purposes, one could break up the quantization of x to C_j into two mappings. First, the

encoding of $\vec{x}$ to j, and second, the decoding which would map j to C_j. If j were allowed to take on values from 1 to J, then one could say the coding rate was $\log_2 J$ bits per vector.

With respect to coding, vector quantization could be described as follows. The first step is to somehow determine the allowable levels for quantization. This is equivalent to finding a codebook against which all blocks of data to be coded are matched. The next step is to sequentially break up the data to be coded into blocks, or "data vectors". Hopefully, the data within a block will represent some sort of natural unit in which there is likely to be a certain amount of order. Arbitrarily organized data will work, but best (lowest error) performance is obtained when individual elements within a block are somewhat correlated. Third, each block of data to be coded is matched up with an entry in the codebook in such a way that the codebook entry chosen most nearly approximates the original data. Finally, the identity of the codebook entry is transmitted in lieu of the original block of data. (See figure for block diagram of a vector quantization coder)

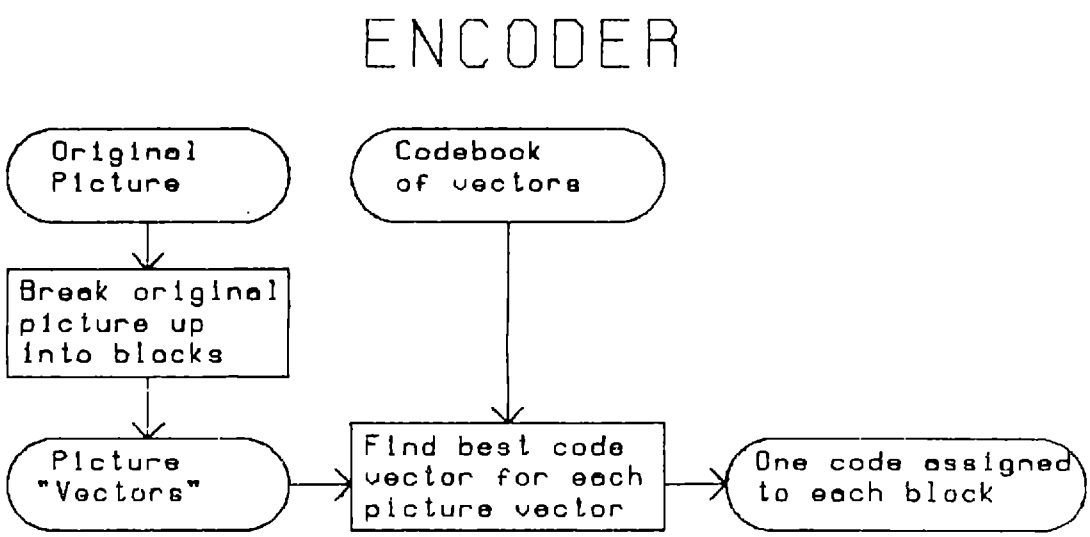

Figure 1. Vector Quantization Picture Coder

The beauty and attractiveness of vector quantization coding is that while the coder is relatively complicated, the decoder is as simple as table lookup. Assuming the receiver has an identical copy

of the codebook used while encoding the data (either built into the system or transmitted independently), decoding is simply a matter of replacing the code identity with what it represents. Since data was originally transmitted in a predictable, sequential order, reconstruction of the original signal is trivial. A computer terminal might be considered to use vector quantization decoding when it converts an ascii character code to a 5x7 matrix of dots making up a visually recognizable character. (See figure for block diagram of a vector quantization decoder)

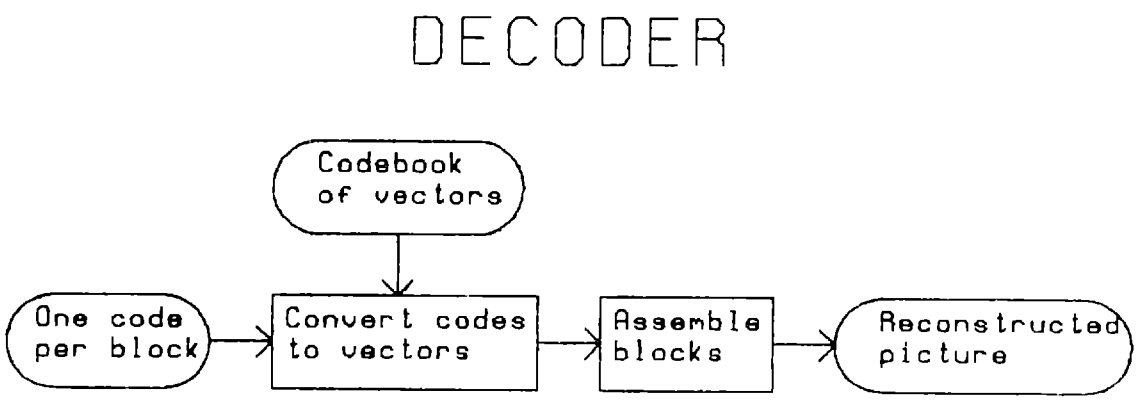

Figure 2. Vector Quantization Picture Decoder

2.1.1.2 General References

Perhaps the best general reference for basic vector quantization was written by Gray [Gray 1984]. In this paper he not only outlines the basics of vector quantization, but also describes various applications and details of more advanced vector quantization systems. Another important work on vector quantization includes a theoretical paper by Gersho [Gersho 1982] in which he discusses various mathematical properties of vector quantizers.

2.1.2 History

2.1.2.1 Motivations

As described by Gray [Gray 1984], one of the first justifications given for vector quantization appeared in Shannon's rate-distortion theory [Shannon 1948], the branch of information theory devoted to data compression. Shannon stated that better performance could always be achieved by coding vectors instead of scalars. As Gray said,

> While some traditional compression schemes such as transform coding operate on vectors and achieve significant improvement over PCM, the quantization is still accomplished on scalars and hence these systems are, in a Shannon sense, inherently suboptimal: better performance is always achievable *in theory* by coding vectors instead of scalars, even if the scalars have been produced by preprocessing the original input data so as to make them uncorrelated or independent!

This theory was not exploited in actual systems for two reasons. First, there existed no simple technique for determining the quantization levels to be used, and second, existing coding techniques were both adequate for existing tasks and showed promise of further improvement. Near the end of the 1970s the first problem was greatly alleviated by the discovery that work done by Lloyd [Lloyd 1957] on scalar quantization algorithms could be extended to the vector case.

2.1.2.2 LBG Algorithm

The recognition that Lloyd's algorithm was generalizable to the vector case was published in 1980 by Linde, Buzo, and Gray [Linde, Buzo, and Gray 1980]. This was a crucial step in the development of vector quantization because until that time there had existed no algorithm by which vector quantizers could be designed. This algorithm (often referred to as the "LBG Algorithm") provides a simple way in which vector quantization levels might be derived in an appropriate way, so as to in some sense optimally partition a training sequence of vectors into quantized levels.

Their algorithm for deriving a codebook based on a set of training vectors is iterative, and can be described as follows. The initialization step involves choosing the starting codebook of vectors.

This could be a codebook used previously, or something arbitrary, such as evenly spaced points in the vector space. The iteration begins by assigning each training vector to its "best fit" code, based on some error metric and an exhaustive search. Next, given a set of vectors assigned to a particular code, that code is modified to optimize its error relative to the training vectors currently assigned to it. This two step process continues iteratively (see figure).

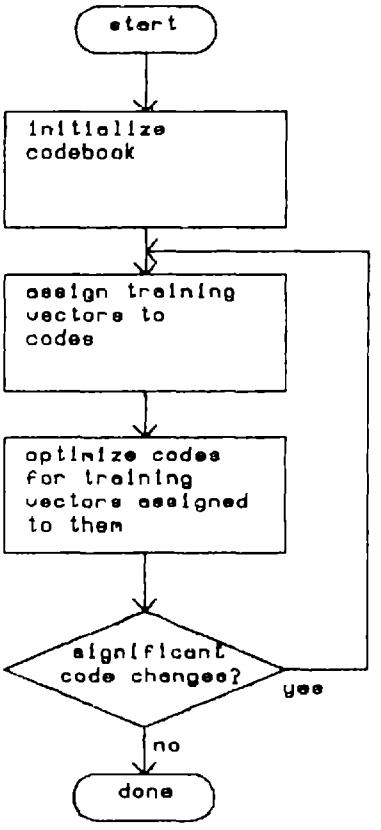

Figure 3. LBG Algorithm

Step one is where the training vectors are reassigned to the newly modified codebook, and step two is where the codebook gets altered to minimize the error between the codes and the training vectors currently assigned to it. The process is terminated when the overall error, between all the training vectors and the codes they are assigned, reaches a low enough percent change between one iteration and the next. The codebook is then considered complete, and, given arbitrary error

tolerances, provides at least a local minimum of error with respect to the training data used to generate it.

In this algorithm, by far the most difficult task is to come up with an acceptable initial codebook. As stated previously, one option is to initialize with an old codebook, or to evenly space the initial codes in the vector space. Perhaps the best simple initialization, however, is to randomly choose from the training set a sampling for use as the initial codes. Quantizers based simply on random samples as their codebook have been shown to be near-optimal [Roucos, Schwartz, and Makhoul 1983], so using a random quantizer as an initial codebook isn't as much a shot in the dark as it may appear at first.

A more complicated method of determining an initial codebook is by the use of what is called a "product code" [Gray 1984]. Using this method, one would quantize the individual pixel elements individually and then consider all the combinations of individually quantized pixel levels. Naturally, this generates far too many codes, but one might consider a subset of these codes as an initial codebook.

Another technique is generating the initial codebook by "splitting" [Linde, Buzo, Gray 1980]. In this method, one begins by developing a codebook with only one entry in it. The user then develops two codes using the knowledge that the optimal one code codebook was developed in the previous step. There are a number of ways to develop a codebook of size 2K from a codebook of size K, but perhaps the simplest is to leave the K previous codes, and add K more which are the original K slightly perturbed. This could be used as the initialization for a LBG algorithm which would converge on a codebook of size 2K. The advantage of retaining the original K codes is to ensure that the error does not increase.

As a final comment on the LBG algorithm, one might object to the notion of developing the codes for a training sequence and then "assuming" that they will be appropriate for other images is a purely heuristic idea, and not one particularly justifiable mathematically. In fact, however, as

Gray points out [Gray 1984], there are only relatively weak statistical requirements on the source generating both the training sequence and the data to be coded. Specifically, all that is required for assuring that the expected values of error for transmitted data be the same as the expected value of error in a training sequence is that the source for both the data to be coded and the training sequence be asymptotically mean stationary.

2.1.3 Variations

The vector quantization schemes described above are the basic algorithms, with no features added to aid in the implementation of the systems, and with nothing added to enhance the performance. Because the decoder is so simple, virtually all refinements of the basic algorithm are in the form of improving the way the codebook is derived. Several improvements are described below.

2.1.3.1 Tree Codebooks

The greatest liability to vector quantization is the fact that there is a tremendous amount of computation involved in the generation of codebooks. The LBG algorithm calls for exhaustive search of every code in the codebook for each vector in the training sequence. This set of searches must be repeated for each iteration. This means that computation is proportional to the product of codebook size and training sequence size. This is in addition to the fact that more iterations are required the more codes and training data is used.

One method to get around this problem is to use so-called "tree codebooks" in an effort to create admittedly sub-optimal codebooks in exchange for requiring a fraction of the computation. The original presentation of this method was done by Buzo, et al [Buzo, Gray, Gray, and Markel 1980] in the context of speech processing. The idea is to first determine a two code codebook based on the original training set. Next, one partitions the training set into two sections, based on which code the members are similar to. Then, the algorithm treats each half of the training set independently, and begins to again split each "half" into half again, and so forth. Consequently, at each node of the binary tree, there are only two codes, which direct the code looking for a closest

fit to one branch or another, until the tree terminates. While most easily described in terms of a binary tree, any number of branches could occur at the nodes.

Tree codebooks provide much faster determination of the closest fit codebook entry ($O(log(n))$ vs. $O(n)$), but have two drawbacks. First, this method requires storage for twice the number of codebook entries, and, second, the codebook entry one arrives at using tree codebooks is not necessarily the optimal entry. Buzo, et al [Buzo, Gray, Gray, and Markel 1980], report that for a speech application operating at 8 bits per vector an extra two bits per vector were required in the tree codebook case to achieve comparable performance with the full search case.

2.1.3.2 Vornoi Cell Searches

Another way to reduce search time is to do extensive codebook preparation to make finding the appropriate code easier. The "Vornoi Cell" method is based on the fact that the every possible picture vector is mapped to only one "best fit" code vector such that the picture vector space is partitioned into distinct "vornoi cells" in such a way that only those possible picture vectors within a vornoi cell would be coded with a particular entry from the codebook. This is similar to partitioning a geographical region into districts for school attendance purposes (see figure).

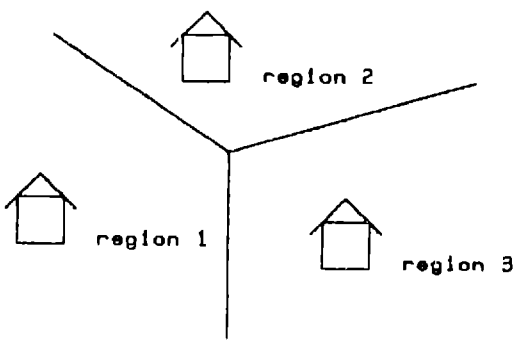

Figure 4. Vornoi Partitions

The locations (and *only* those locations) within a district are served by a common school, and by comparing one' cross streets with a table (the axes on a school district map, for instance) one can

quickly identify to which district one belongs. It is not necessary to measure the distance to every school to determine which is closest.

Similarly, vornoi cell techniques such as the one described by Cheng, et al [Cheng, Gersho, Ramamurthi, and Shoham 1984] set up a table for each dimension listing which vornoi cell could possibly overlap this dimension, allowing for progressive elimination of possible codes one dimension at a time. Determining the exact vornoi cell partitions is extremely costly, however, and invests significant effort in reducing coding time on the assumption that a given codebook will be used for coding a number of pictures.

2.1.3.3 Product Codes

A "product code" is a code that consists of a number of separately determined "factors" which combine to make up the entire code. A good example of a product code in vector quantization in the breaking apart of an LPC voice code vector into "gain" and "shape" factors as done by Buzo, et al [Buzo, Gray, Gray, and Markel 1980]. Here, the original LPC vector was considered naturally "separable" into two terms, both of which were necessary to code the intended vector accurately.

Another form of product code is to treat the full codebook as the product of a statistic and a separate codebook normalized for this statistic. In an effort to make a codebook more generally applicable, work has been done to normalize various statistics, so that more entries in the codebook would be "in the running" for best fit, thus giving more accurate coding. Baker and Gray [Baker and Gray 1983b] have done work with extracting the mean for separate transmission, and Murakami, et al [Murakami, Asai, and Yamazaki 1983] have normalized both the mean and variance of code blocks in their work. Naturally, along with normalizing the codes, the coder must also send along statistical information to allow for accurate reconstruction of the data.

Because of this additional information which need be sent, coders using algorithms which normalize and extract statistics have not yet demonstrated better performance than more straightforward vector quantizer systems. This is because the flexibility of separating statistical

information (such as the mean pixel value) from the vectors results in the capability to reproduce more possible vectors than might actually occur, thus introducing an inherent suboptimality. On the other hand, using product codes allows for more simple (although possibly suboptimal) code assignment because one decomposes a large search into two smaller ones.

2.1.3.4 Multistage Codebooks

Multistage codebooks are codebooks in which a number of codes are assigned to a given vector to be reproduced in such a way as to get progressively better approximations to a vector. The original vector is first matched to a code, then the error vector between the original and the first level code vector is matched at the second stage, etc. This method, first demonstrated by Juang and Gray [Juang and Gray 1982], has the advantage that it requires less computation in that the comparisons done at any given stage are typically much fewer than that which would be performed in a single-stage method. It also allows for greater freedom in the reproduction quality, as one might include as many stages as one needs accuracy. The drawbacks, of course, are that a number of codes are needed to represent a single vector, and that the decoder requires greater complexity to reconstruct the original vector.

2.1.3.5 Segmented Codebooks

A method analogous to that of tree coding is the method of segmented codebooks. In this type of system, the vector to be coded is only compared for best fit with a subset of the number of codes available. The novel aspect of segmented codebooks is that the user often chooses to artificially overrepresent a type of code (serviced by a particular segment) in an effort to improve coder performance on a particular type of vector. An example of using segmented codebooks to reduce the occurrence of particularly offending errors is the partitioning of a codebook of picture vectors into codes representing edge vectors and codes representing non-edge vectors. In this way, the user can in effect weight one type of error more highly than another. Gersho and Ramamurthi [Gersho and Ramamurthi 1982] have presented two segment codebook for picture coding, and

Ramamurthi, et al, [Ramamurthi, Gersho, and Sekey 1983] have presented three segment codebooks for image coding. Ramamurthi and Gersho [Ramamurthi and Gersho 1984b] have also presented a codebook partitioning scheme with thirty different categories corresponding to different edge types.

2.1.3.6 Vector Quantization in the Transform Domain

While vector quantization seems most appropriately suited to reducing redundancy in the spatial domain, some experiments have been done with applying vector quantization to 1-d and 2-d transforms [King and Nasrabadi 1983] [Nasrabadi and King 1983]. In these experiments, pictures were coded by applying either 1-d or 2-d Hadamard transforms and truncating the high frequency components followed by vector quantization.

As one might expect, results were no better than those achieved using vector quantization purely in the spatial domain. This is because the same vectors grouped together in the transform domain would also be grouped together in the original pixel luminance domain, since nearby vectors in one domain are also nearby in the other (see appendix for "proof" of distance constancy after transformations). Even these results must be tempered by the fact that Huffman coding of the vectors was also used, an obvious improvement which was left out of the results given by others. In addition, it is important to note that in this implementation one of the big advantages of vector quantization, the simple decoder, is sacrificed.

The possibility of using transformed vectors for *searching only* will be discussed in section three.

2.1.3.7 Vector Quantizers with Memory

Just as traditional data compression includes many systems involving some sort of predictor or adaptivity, so too can vector quantization be altered to use "memory". These coders are in most cases the direct analogs to the one-dimensional case, so they will not be discussed here. For an overview of existing vector quantizers with memory, the reader is referred to Gray's article [Gray 1984].

2.2 Applications

As mentioned a number of times above, vector quantization has been successfully applied in a number of practical applications as a means of data compression. The following is a brief description of some of the ways VQ has been used.

2.2.1 Speech Applications

2.2.1.1 Coding

Vector quantization was originally developed in the context of speech processing. Typically, a VQ speech coding system, such as described by Buzo, et al [Buzo, Gray, Gray, and Markel 1980] will code speech LPC coefficients as a group to make up a vector with dimensionality of about ten. Error measurements used in the vector quantization process need not be squared euclidean distance and for speech processing alternative metrics are often used. A number of the improvements listed above have been used in speech systems with good results.

2.2.1.2 Speech Recognition

It is also interesting to note that the speech recognition problem, when attacked by use of such methods as template matching, is in fact another application of vector quantization, although the dimensionality is of course greater than the simple use of LPC vectors listed above. The recognition system tries to match segments of a given speech waveform to a code (vector) representing a particular phoneme or word. Thus appropriately identified (coded), the speech recognition system need only deal with prototype "words", rather than with highly varying and speaker dependent utterances.

2.2.2 Picture Applications

Picture coding using vector quantization was developed and published by at least four separate groups since 1980 [Baker and Gray, 1983a] [Murakami, Asai, and Yamazaki 1982] [Gersho and Ramamurthi 1982] [Yamada, Fujita, and Tazaki 1980]. In all these cases, the "vector" being

quantized was a group of adjacent pixel intensity values. Typically, a rectangular block of size four to sixteen was used. Blocks of greater dimension were avoided because of the computation involved. In all cases, the algorithm used for developing the codebook was some variation of the LBG algorithm.

3. THEORETICAL IMPROVEMENTS

In this section improvements to the existing LBG algorithm will be described. These improvements include the use of a multi-dimensional tree structure to allow for fast error-free nearest neighbor searching and the development of several variations of a completely novel algorithm for developing codebooks, an algorithm which significantly reduces the computation necessary to generate codebooks without sacrificing performance.

3.1 Multidimensional Searching with K-d Trees

This section describes the use of a multi-dimensional tree structure to facilitate fast codebook searching. Several issues involved with this application are also discussed. While the abstract concept of this tree structure was developed by Bentley [Bentley 1975], this presentation represents the first recognition of the appropriateness of this structure to the vector quantization search problem and the first discussion of the issues involved in doing so.

3.1.1 The Search Problem

The greatest drawback of vector quantization is that is very computationally demanding. To do full search vector codebook development on conservatively sized codebooks (on the order of 256 4x4 codes) takes a fair number of hours on a large computer to accomplish (see section 4 for details and actual performance tests). The vast majority of the time spent both in developing VQ codebooks and in the encoding process is spent trying to find the closest fit in the codebook to a vector. In codebook development, the vector being matched is from the training set of vectors, and in the encoding process, picture vectors are being matched with codebook entries. In an effort to alleviate this problem, several computation saving variations on the basic VQ algorithm have been developed, as outlined in the previous section. Unfortunately, all these methods achieve their computational savings at the cost of performance. As Gray says [Gray 1984],

> Ideally, one would like to take a full search, unconstrained VQ and find some fast means of encoding having complexity more like the [log(N)] techniques than that of the full search.

... Unfortunately, however, no design methods accomplishing this goal have yet been found.

3.1.1.1 Pattern Matching

One way to approach this problem is to attempt to solve it in the same way as would a person trying to find the best match to a given vector. To use the example of vector quantization of images, a person would look at the vector to be matched and classify using terminology such as "brighter on the left" or "has a dot in the middle" and would proceed from there. This is basically a pattern matching approach, and was investigated by the author as a means of reducing the number of vectors in the codebook which needed to be examined. Unfortunately, finding a set of mutually exclusive as well as meaningful "patterns" for classifying vectors is basically a heuristic problem. The edge classification schemes such as Ramamurthi and Gersho [Ramamurthi and Gersho 1984a] can be considered to be a pattern matching approach.

3.1.1.2 Multidimensional Nearest Neighbor Search

Another approach is to treat each vector as a point in multidimensional space. Using this approach, one could say that the goal was to group "nearby" vectors together. Of course, this leaves the problem of defining "nearby" in terms of which vectors are in actuality similar, but this problem is nothing new. When treated as a multidimensional search problem, it becomes clear that what is needed is a way of organizing data so that all nearby points (vectors) are easily determined. The most obvious way to do this is to have an M-dimensional array (where M is the dimensionality of the vector) with entries where vectors in a codebook occur. This way, a vector to be matched could first look in the array entry it would occupy, to see if an exact match occurred. If an exact match was not found, then an outwards search could be made through surrounding array entries until a vector was found. Unfortunately, this method of storage requires space exponential in the dimensionality of the vectors being used. What is needed is some sort of flexible structure which allows for fast access to a "region" in multidimensional space.

3.1.2 K-d Trees

K-d trees (short for K-dimensional trees) were developed by J. L. Bentley [Bentley 1975] and provide a data structure which allows for log(N) multidimensional searches to be accomplished. This was the structure which was used to allow for fast searching in connection with vector quantization.

3.1.2.1 Structure

K-d trees are binary trees designed to organize multidimensional data (see figure).

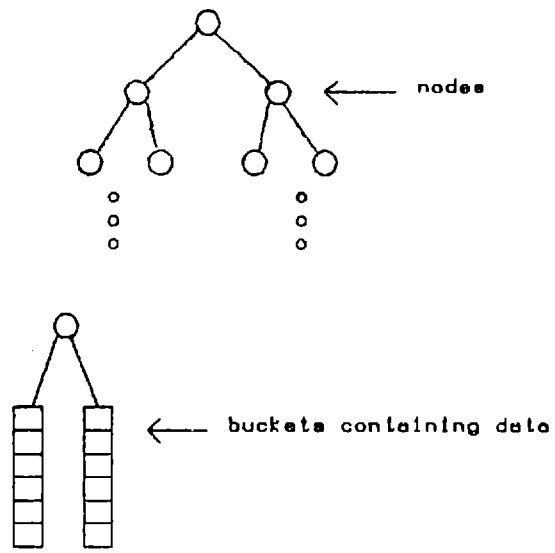

Figure 5. A K-d Tree

These trees consist of a set of interconnected nodes and a set of "buckets" located at the lowest level of the tree. The nodes serve to organize the data, and the buckets hold the data. Similar to binary search trees [Knuth 1973], each node in the tree partitions the data "below it" on the basis of whether the data is greater or less than a certain value. The novelty of the tree is based on the fact that each node partitions the data based on only one dimension.

A node in a k-d tree consists of either four or five elements. The first element tells on the basis of which dimension the data is split, and the second element gives some value which is the threshold which determines in which child of the node data could be found. If the appropriate dimension of the data was less than the threshold it would reside in the left child, and if it were equal, it would go in the right child. Data with dimensions equal to the "split value" could go to either (but not both) child, depending on convention. The next two entries are pointers to the left and right children. The children could be either another node, or could be a bucket, if the bottom of the tree had been reached. The optional fifth entry is a pointer to the node's parent. This element is optional, because it is not needed in all applications.

Buckets are really a special kind of "terminal" node which points to data instead of other nodes. In a completely organized tree, each bucket would point to only one "record", or "piece of multidimensional data". Usually, however, it is more efficient to have a bucket point to a number of records which would all belong at this spot. This way fewer nodes are needed. If a particular record is required, it can be searched for among the few records sharing a given bucket. The records in a given bucket could be stored in an array if it was guaranteed that there would never be more than a certain number of records per bucket, or could be stored as a linked list. As with a node, buckets and even records could contain a pointer to their "parent".

What this amounts to is a progressive chopping up of the K-dimensional space which the K-d tree is trying to organize. Each node represents a hyperplane parallel to all but one dimension, and separates the data on the basis of that dimension. For example, consider the two-dimensional case which is the simplest non-trivial case (see figures). Here, the top node represents the first partition, dividing the two-dimensional region into two half planes. The next level of the tree divides these half planes once again, and so on. The lines (hyper-planes) dividing up the "space" correspond to nodes, and the regions defined by the nodes correspond to buckets. Each partition could, in theory, be done with respect to any dimension, although to get the best partitions one would

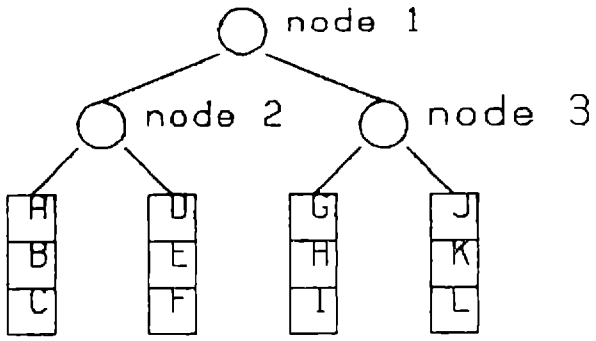

Figure 6. Example of a K-d Tree

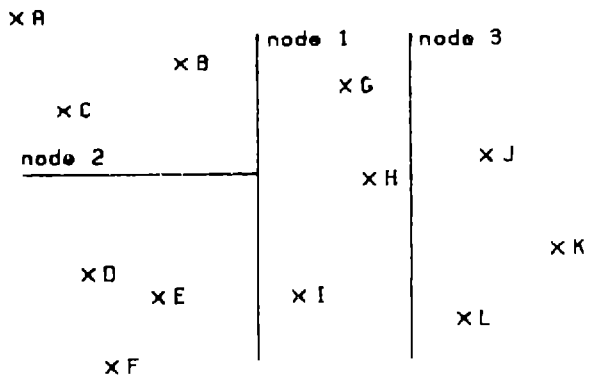

Figure 7. Partition Corresponding to Example of K-d Tree

normally use a variety of different dimensions for the partitioning.

3.1.2.2 Building a K-d Tree

K-d trees, like binary trees, are typically built from the top down to conform to a particular set of data. The basic "algorithm" for building a k-d tree is to build it recursively. If the set of records needed to build a tree is smaller than the maximum size of a bucket, all the records are stuffed in the bucket and the tree is done. If there are too many records, a single dimension is chosen to be the "splitting key" and the actual "splitting value" is chosen. The records are then divided up into two halves based on whether the value of the dimension being used as the splitting value is greater or less than the "splitting value". These two lists of records are then used to make a k-d tree for

each, and these trees serve as the left and right children of this node. Splitting will continue until the number of records in the right and left children falls below a certain threshold. In this way it is insured that the number of records in a bucket will never exceed a certain maximum size.

The parameters available during building are: which dimension to use as the key for splitting at a given node, the value of this dimension at which the partitioning should take place, and the maximum size of buckets which are to be allowed. These values are typically chosen as follows. The most tricky question is which key (dimension) should be used for splitting at a particular node. The simplest thing is to simply choose cyclically among the various choices. That is to say, on the top level of the tree use the first key, on the second level, use the second, etc.. When you've used all the keys once, start over with the first one again.

Another preferred method is to choose the dimension which best spreads out the data. That is to say that one could choose the dimension which has the largest variance associated with it. The reason for choosing this for the split key is that if the data is very spread out along a particular dimension then presumably differences in that dimension are more "significant" in some sense than differences in another, more densely grouped dimension. This has the effect of choosing uncorrelated dimensions and also has the effect of making the partitions more "cubical" as opposed to making them long and skinny, which would lead to more lengthy searches, as will be pointed out later.

The other two decisions are pretty easy. For as balanced a tree as possible, one could use the median key value from among the records involved as the split value. As far as bucket size goes, Friedman, et al, [Friedman, Bentley, and Finkel 1977] report that a bucketsize averaging around eight entries seems to be optimal for a wide range of problems.

3.1.3 Solution: Full Search Algorithm

3.1.3.1 Algorithmic Details

Using this data structure, Friedman, et al [Friedman, Bentley, and Finkel 1977] describe a search algorithm for finding best matches in logarithmic expected time. The algorithm can be described geometrically in the following way (see figure).

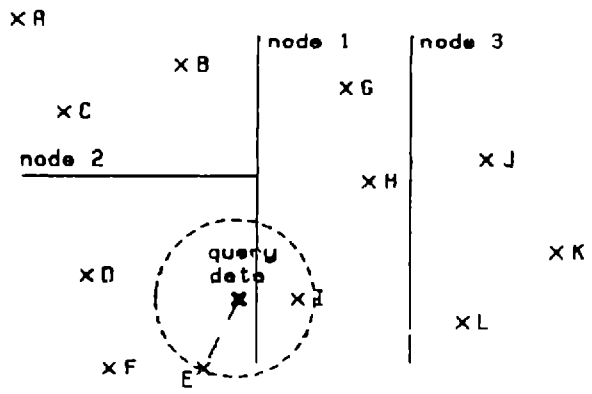

Figure 8. Searching a K-d Tree

Initially, a K-dimensional point, or "query record", is presented with the intention of finding which entry in the k-d tree is most similar to it. The first step is to identify which bucket the query record would have been assigned to had it been a member of the K-d tree. Next, every entry in that bucket is compared to the query record to determine which is most similar based on some distance-preserving error criterion. Then a "sphere" is constructed centered around the query record with a radius equal to the distance to the nearest neighbor found so far. The purpose of this is to see whether any other buckets (partitions in K-dimensional space) are worth checking to see if they could possibly contain an entry more similar (closer) to the query record. As many partitions are checked as are necessary, while updating the radius of the sphere as new "closest matches" are found.

If the number of dimensions is not too great, Bentley shows that search time is logarithmic with the size of data set being checked. At *worst*, the distance ("closeness") between the query record and all the tree entries must be calculated, and this is just the old search method plus the overhead

of maintaining a tree structure for searching.

3.1.3.2 The Problem of Too Many Dimensions

The problem with too many dimensions is that the K-d tree structure only allows a single dimension to be checked at each node, and the number of levels of the tree only grows logarithmically with data set size. This, of course, is also the basis for the computational advantages of K-d trees, but if all the dimensions are uncorrelated, then being able to only check a small fraction of them during the search procedure can result in greater computation during searches.

This technique gets part of its computational advantage from finding the nearest neighbor to the query record close to where the algorithm first looks, and if all the dimensions are uncorrelated, the "first shot" gotten by following only a few dimensions down the tree won't necessarily be very close to the eventual nearest neighbor. Another way to look at it is to realize that this method presumes that all buckets will contain records which are pretty similar to one another, and if only a fraction of the possible dimensions are used as hyper-planes to define a bucket, then the buckets will be long and skinny, and the closest match in a bucket may generate a "sphere" which overlaps many other long, skinny buckets (see figure).

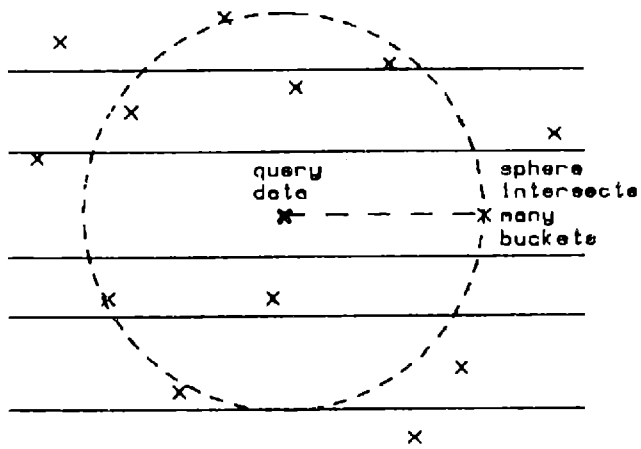

Figure 9. Searching a Tree with Skinny Buckets

Note that this becomes a serious problem only if the dimensions are very uncorrelated. If they are

correlated, then the members of a bucket will be grouped together (and the sphere will be small) despite the fact that they occupy a long, skinny bucket. However, it is reassuring to remember that if all the dimensions *are* uncorrelated, then there is no way to achieve adequate vector quantization in the first place.

3.1.3.3 Computed Keys

Although the above analysis indicates that it is *uncorrelated* dimensions, rather than simply a large number of dimensions that causes K-d tree searching to be inefficient, there is still a reason to want to work with reduced dimensionality for its own sake. Specifically, the larger the dimensionality of the records (vectors), the more computationally costly the distance calculations are. One would prefer to represent each vector by a set of "computed keys" which represent the vector while having reduced dimensionality. These keys are to be used *only* for searching purposes; at all times the actual pixel values are to be used for code representation.

The most obvious "computed keys" to use are those which concentrate the "information" into a reduced set of uncorrelated coefficients. By definition the Karhunen-Loeve Transform would optimally concentrate vector energy into uncorrelated coefficients, but because of computational considerations the Discrete Cosine Transform (DCT) was used as a method of reducing vector dimensionality while still retaining near optimal energy compaction properties [Chen and Smith 1977].

Because an orthogonal transform such as the DCT preserves distances (see appendix) one can directly perform all distance comparisons in transform space without affecting the algorithm, and since only the low frequency transform coefficients significantly affect the picture [Chen and Smith 1977], one can perform distance calculations on a truncated set of coefficients at reduced cost in the spirit of transform coding. It is not clear however, whether the reduced cost of distance calculations will outweigh the added cost of computing the transform coefficients (see section four for performance tests).

Other computed keys derived in more heuristic ways could also be used as a method of reducing dimensionality while preserving picture information. Such computed keys include using just pixels on the edge of pixel blocks and using every other pixel (pixel subsampling), under the assumption that nearby pixels were very correlated. Both these methods were tried and yielded satisfactory, although somewhat degraded performance. Other heuristic computed keys specific to other applications could also be used.

3.1.3.4 Computational Tricks

In addition to the major search speed improvement of K-d tree searching with or without computed keys, other, more mundane "tricks" were used to speed calculations. First, when performing distance calculations, one wants only the closest pair of points and does not care whether a particular pair of points was close to being the best or not, as long as it is not the best. To this end, distance calculations were always performed with respect to a threshold which was the distance to the closest neighbor discovered so far. If the threshold was exceeded on the basis of only a few dimensions (assuming a distance metric in which the triangle inequality holds), the calculation was terminated on the theory that "a miss is as good as a mile".

Another "trick" used was the pre-calculation of squares in a look-up table for use when doing euclidean distance calculations or when calculating variances for choosing which dimension to use at a K-d tree node.

3.2 Nearest Neighbor Searching

This section deals with the development of a completely new algorithm to determine the codebook used for vector quantization coding. This algorithm, the Nearest-Neighbor or "NN" algorithm, results in a significant computational savings as compared with the previously used algorithm without sacrificing performance.

3.2.1 Clustering

3.2.1.1 Codebook generation is Clustering

A crucial realization in understanding vector quantization codebook development is that these algorithms are in fact what are known as "clustering" algorithms. While other authors have occasionally referred to this fact in passing, to the best of the author's knowledge, this presentation represents the first successful attempt to use this insight to fundamentally change the way codebooks are developed for vector quantization coding.

Clustering algorithms have "traditionally" been used to find correlations between many seemingly unrelated variables. A typical biological application might be an attempt to isolate genetically distinct classifications among a wide range of samples. A marketing application might be to distinguish various types of "typical" consumers from among the population at large. And more recently, pattern recognition algorithms in artificial intelligence will call for grouping an image into various distinct objects or patterns.

All these problems, as well as the vector quantization problem, have a great similarity. A "clustering" algorithm is one which takes a large number of data points and attempts to group them into meaningful groups, and that is *exactly* the point of vector quantization. The goal is to exploit the redundancy that occurs because there are many similar patterns occurring locally in a picture. The codebook generation problem is simply one of finding out what these locally similar occurrences are.

3.2.1.2 LBG and the K-means clustering algorithm

A very good argument for vector quantization codebook development being just an application of clustering is the fact that the only existing algorithm for determining a VQ codebook is algorithmically equivalent to one of the most famous clustering methods, the "K-means" algorithm. The K-means algorithm [Hartigan 1975] is a method which defines "a priori" how many clusters one wants. It involves choosing K initial "means", or "original clusters" and iteratively improving

upon these clusters. Exactly as in the LBG algorithm, the K-means algorithm alternates between assigning all members of a training set to the most appropriate cluster and redefining these clusters by replacing them with the mean of those training vectors assigned to each "cluster".

Linde, Buzo, and Gray argue that their algorithm is *not* equivalent to clustering algorithms because the motivations behind it are different. They claim that the motivation for their algorithm, and for the Lloyd quantization algorithm [Lloyd 1957] which was its predecessor, is that at each iteration one is trying to minimize error not only for the training set but for a class of data represented by the training set. With all due respects to Linde, Buzo and Gray, this difference seems academic.

3.2.2 Nearest Neighbor (NN) Clustering

This section describes the development of the NN algorithm. Motivations, theoretical justification, and various implementations of this algorithm are discussed. Finally, the advantages of using this completely novel algorithm for the development of vector quantization codebooks are discussed.

3.2.2.1 Motivation

The LBG (k-means) clustering algorithm succeeds as a quantizer because it tends to generate spherical clusters. Another algorithm which may generate "better" clusters in a pattern recognition sense, such as one developed by Gowda and Krishna [Gowda and Krishna 1978], would in fact provide for quantization with greater error because they often result in non-spherical clusters (see figures). Of course this is not a problem in many applications, but is unacceptable for a quantizer intended to minimize error. What one wants are clusters which contain only those members which are similar to one another in the sense that they can be represented by a common vector similar to all members.

Further considerations affecting algorithm design were the two biggest computational drawbacks of the LBG algorithm. First, the LBG algorithm is iterative and has an unknown execution time. The algorithm may converge rapidly or may take many iterations. Unfortunately, this problem gets worse when dealing with points with large dimensionality, when dealing with a large number

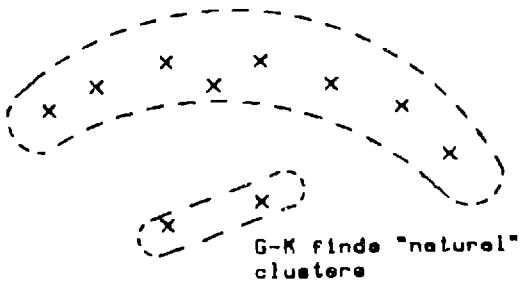

Figure 10. Clusters Found by Gowda-Krishna Algorithm

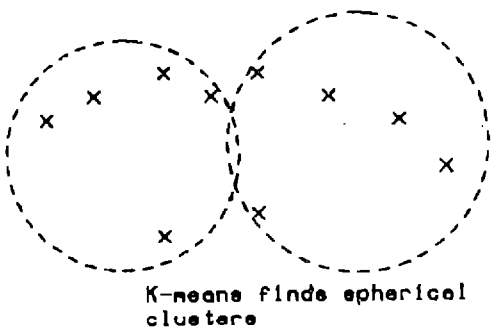

Figure 11. Clusters Found by K-means Algorithm

of clusters, and when dealing with a large training set. Also, the LBG algorithm depends on the entire training set at each step. This means that there is a computational bottleneck at each iteration which gets worse as the size of the training set increases, whereas a more parallel algorithm, dealing with less than the entire training set, would be much more attractive computationally, particularly in terms of eventual hardware implementations. With this all in mind, an appropriate clustering algorithm was designed.

3.2.2.2 Mathematical Justification

The algorithm that was developed is based on the premise that all members of a cluster should be similar to the vector representing that cluster, and is "agglomerative" in that it begins by having

each point represent its own cluster, followed by merging of pairs of similar clusters while trying to minimize error. If N clusters each contain one point, and one wants to have (N - 1) clusters with minimal error, the optimal partitioning is to merge the two closest points and replace them with the point which minimizes their error with respect to the new point (replace them with their mean).

Unfortunately, once clusters have more than one member, things become more complicated. However, if one assumes that relative to the entire set of points, the members of a cluster can be approximated by their mean, then it becomes possible to optimally derive (N - 1) clusters given an optimal partition of N clusters. One does this by merging the two clusters which when merged minimize error. Consequently, if one begins with an optimal partition (each point with its own cluster), by induction it is true that by merging similar clusters until the desired number of clusters (equivalent to the number of codes wanted for the codebook) is reached, one will arrive at the optimal clusters in the sense that they minimize the error between the members of the cluster and their mean. The codes for the codebook will be the means of each cluster.

With the following notation:

$\bar{x}_i$ = *mean of the ith cluster*
n_i = *number of elements in ith cluster*
$\bar{x}_{ij}$ = *mean of new cluster formed by merging clusters i and j*
n_{ij} = *number of elements in merge of ith and jth cluster*
$E(\bar{x}_i, \bar{x}_j)$ = *error between point i and point j*

the optimal merging of two clusters (using squared euclidean distance as the error measure) is accomplished by replacing the two clusters with the weighted mean

$$\bar{x}_{ij} = \frac{n_i\bar{x}_i + n_j\bar{x}_j}{n_i + n_j}$$

and by letting

$$n_{ij} = n_i + n_j$$

Again, using the approximation that members can be represented by their mean, then the

appropriate pair of clusters to merge can be calculated by observing the error generated by merging two clusters.

$$error = n_i * E(\mathcal{X}_{ij}, \mathcal{X}_i) + n_j * E(\mathcal{X}_{ij}, \mathcal{X}_j)$$

which, with squared euclidean distance as the error measure reduces to

$$error = \frac{n_i n_j}{n_i + n_j} |\mathcal{X}_i - \mathcal{X}_j|^2$$

One need only then find the minimal value of this term to optimally cluster the data in the training set.

The codebook development algorithm is then simply a matter of progressively merging together pairs of clusters with minimal weighted distance. The key to quick execution of this algorithm is to be able to quickly find the closest pairs of points among an essentially randomly distributed set. The obvious (slow) way is to find each point's nearest neighbor (a log(N) search using k-d trees) and choose the minimum distance between nearest neighbors. This leads to closest pair computation cost on the order of Nlog(N). It is possible, however, to find the closest pair of points in linear time using other, more efficient methods, as described below.

3.2.3 Variations of NN Clustering

The basic approach taken in all these variations is based on the fact that one can afford to be a little inexact in choosing the closest pair to merge. With a minimum training set of a single image and typical parameters, there are thousands of merges performed in determining the codebook, so it really isn't too critical if one merges the absolute closest pair at each try. One can easily afford to merge the tenth closest pair, or even the hundredth, as long as the close pairs get merged eventually. If a set of points get merged, it does not matter which pairs were merged in what order.

3.2.3.1 Spherical Search

The simplest compromise is to merge all pairs which fall below a certain threshold distance at the same time. Of course, the merging of two points affects which points might be closest, but the supposition is that no single merge will dramatically affect the weighted distance calculations. This is not a bad assumption, either, since the merges were designed to minimize the change in weighted distances between points.

In the "spherical search" method, a threshold is determined and all pairs having a weighted distance below this threshold are merged. The members of the training set can first be organized as a K-d tree, and then all points within the threshold distance can be determined by checking all buckets which intersect with a sphere of the threshold radius.

The threshold distance can be determined in two ways. Most simply, the sphere can be started out at radius zero and expanded slowly to catch more and more pairs. Second, the threshold could be determined by checking the distances between a random sample of records and considering this an estimate of what the closest distances between pairs were. This is the basis for an algorithm by Rabin [Rabin 1976] for determining the closest pair of neighbors given a set of points. Rabin's algorithm was presented in the context of demonstrating a particular type of algorithm (what he calls "probabilistic algorithms"), and is very efficient for points of just a few dimensions. In his algorithm, he sampled points and then partitioned the vector space into cubes in which he did local searches which arrived at the closest pair of points in time linear with the data set size. The time was linear because the distance comparisons were done only locally (within a partition), and the way K-d trees are constructed there is a fixed maximum size of a bucket.

While this algorithm as Rabin presents it cannot be efficiently implemented in practical vector quantization applications because it requires storage space exponential in vector dimension (vectors typically have over ten dimensions), a modified version is implementable by using the partitions generated by the K-d tree buckets as the partitions. These partitions grow in number only linearly

with data set size and independent of dimension. In addition, the minimum dimension of these bucket partitions could be considered an estimate of threshold radius size.

3.2.3.2 Yuval Search

Another method utilizing local nearest neighbor searches could be used to find close pairs quickly. Yuval [Yuval 1975, 1976] has shown that given a regular K-dimensional lattice of minimum dimension D, one can find all pairs of points of maximum distance $\frac{D}{(K+1)}$. First one constructs K additional lattices by shifting each hyperplane making up the lattice $\frac{D}{(K+1)}$ in each dimension. Next one searches locally within the partitions in each of the (K + 1) lattices for pairs below the minimum distance (see figure).

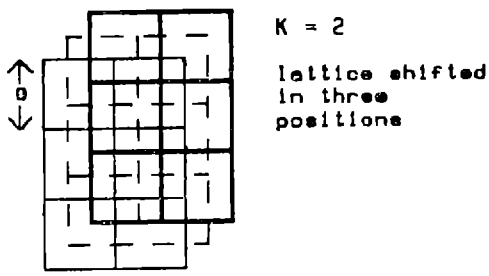

Figure 12. Yuval Shifting of Partitions

Once again, the K-d tree bucket partitions can be used as the lattices.

This algorithm works terrifically for low dimensionalities, but for larger values of K, the fact that one can only be guaranteed of catching pairs of distance $\frac{D}{(K+1)}$ makes it necessary to increase D to such a large value as to negate the advantages of needing only to search for closest pairs within a lattice partition. A compromise is to allow for a certain amount of error and to not use (K + 1) as the factor for shifting and calculating the search threshold, but to use a factor F with $F < (K + 1)$ and do $(F-1)$ shifts of distance $\frac{D}{F}$ in each dimension and search for pairs of

distance apart of less than $\frac{D}{F}$. The limit of this method is to let F equal one (see section four for performance tests).

See appendix for listing of algorithm in generic computer language notation.

3.2.3.3 Simple Search

The most simple method using local searches is to search for closest pairs only within a partition. K-d partitions are applicable once again, and in this method, no absolute threshold of pair distance is used, but rather, the best pair in each partition is considered as a candidate for merging. Clearly some partitions will have larger "smallest pairs" than others, but if only a fraction of the partitions' best pairs get merged (say, the best ¼ of the pairs), extremely acceptable results can be obtained very quickly.

See appendix for listing of algorithm in generic computer language notation.

3.2.4 NN as LBG startup

It is important to remember that although these algorithms may come up with acceptable codebooks (often better than those generated by the LBG algorithm with random initialization), the codebooks can be improved by using the result of this algorithm as the initialization step of the LBG algorithm, so at the very worst, the nearest neighbor clustering algorithms result in a very good initializer for the LBG algorithm.

The important point is that although the result in the NN clustering algorithm can virtually always be improved upon by later LBG processing, the idea was to come up with a fast algorithm to give performance comparable with that obtained by common implementations of the LBG algorithm. The fact that the NN algorithm can *ever* come up with better codebooks than the LBG algorithm (and it does, see section four for performance tests) just shows how important the fact is that the LBG algorithm only guarantees "local" minimums for coding error, a fact often glossed over.

3.2.5 Theoretical Advantages of NN over LBG

In summary, the theoretical advantages gained by use of the NN algorithm for developing codebooks are as follows. First, and most important, the NN algorithm is virtually always faster than the LBG algorithm by a significant factor (see next section for performance tests) and computational complexity grows only linearly with the size of the training set, independent of codebook size. Second, since all practical implementations of the NN algorithm utilize local nearest neighbor searching, processing can be done in parallel to take advantage of the added speed that extra hardware can bring. Third, this algorithm begins with many clusters and works its way down to the final codebook size, thus allowing codebooks to be developed either with a pre-specified number of entries or with a flexible codebook size constrained only by the maximum allowable error in reproduction. Fourth, this algorithm is non-iterative and is therefore guaranteed to terminate in a finite amount of time and not be sensitive to various "initializations". Last, this algorithm lends itself much more readily to flexible tradeoffs between speed and accuracy, as demonstrated by the variety of implementations described above.

4. PERFORMANCE TESTS

All tests were performed on either a Vax 11/780 or a Vax 11/750 with programs written in the C programming language using the Berkeley 4.2BSD Unix operating system. Execution time is measured in Berkeley 4.2BSD Unix internal accounting units. These "units" are measured in seconds and are similar to CPU seconds, except that they are adjusted for system load. All performance tests shown in a single table were run on the same computer. All tests involving the LBG algorithm were run until there was less than a .1% change in the error between the training vectors and the best fit of the codebook being developed. All pictures were 512x512 pixels, and, except where noted, all codebooks were developed using all blocks of the picture to be coded as the training set. All codebooks developed contained 256 entries and blocks were 4x4 pixels in size for a coding rate of .5 bits / pel.

During the course of research, non-linear filters were developed to "smooth out" the abrupt transitions which occurred between blocks. These filters greatly enhanced the visual acceptability of the pictures, but for the sake of concentrating on the performance of other aspects of the coding system, these filters were not used in the following tests.

4.1 K-d Tree Search in LBG Algorithm

The programs used to compare differing search variations were identical with the exception of the exact parameters being varied.

4.1.1 K-d Tree Search Speed vs. Exhaustive Search Speed

In the LBG algorithm, of the two steps in the algorithm (assigning training vectors to codes and adjusting codes to fit the data assigned) the searching out of the best vector is by far the most costly step. If we let

I = *number of iterations the algorithm uses*
T = *number of vectors in the training set*
C = *number of codes the algorithm is generating*

and consider the search to be the dominant time consuming operation, then we can say that execution time of the standard LBG algorithm is proportional to $(I * T * C)$. If the LBG algorithm using K-d trees uses K-d trees efficiently, then one can say that execution time would be proportional to $(I * T * log(C))$. I (the number of iterations) is a complicated function of both T and C, and is data dependent. Of course, there are unknown constants involved, and as pointed out in section 3, it is not at all clear that it is possible to exploit the potential "log search" of K-d trees. At any rate, though, if K-d tree searching is found to be competitive with exhaustive searching than one can safely say that its performance advantage can only increase as the size of the codebook being developed increases. Actual results (see table below) show favorable performance.

There is a certain unpredictability in the execution time of the LBG algorithm because both the algorithm and the computational "tricks" (see section 3.1.3.4) are very sensitive to the data being processed. Differences in the number of iterations needed for convergence (and in the codes generated) with and without K-d trees are because occasionally two separate entries in the codebook are equally good fits to a training vector, and the two implementations don't necessarily choose the same code in this case.

Execution time (in Berkeley 4.2BSD Unix accounting units)				
Picture	LBG with random initialization			
	Without k-d tree searching		With K-d trees	
	iterations	time (secs)	iterations	time (secs)
baboon	18	21954	17	15234
lake	27	33825	30	15059
airport	17	20696	18	15926
lena	25	31123	32	12977
peppers	27	32931	21	7181
plane	24	29904	25	8211

TABLE 1. Speed without K-d Trees vs. Speed with K-d Trees

4.1.2 Computed Keys for Searching

Computed keys were tested and found to degrade performance (because of the approximations they entail) with very slight (if any) savings in execution times, thus supporting the claim that using

computed keys will not result in any savings beyond that achieved by computing distances between points of reduced dimensions. However, this computation time includes the cost of computing the keys (a relatively high cost for sophisticated keys such as transform coefficients). If the cost of computing these keys could be reduced, a reduction in search costs could be expected.

4.2 Nearest Neighbor Search

The nearest neighbor search technique (NN search) presented in section 3 was tested and compared to the LBG algorithm (executed using K-d tree searching). Initially, the Yuval search variation (see section 3.2.3.2) was tested but performed poorly in terms of execution time because of the problem of having too large a dimension for searching as described in section 3. Next, a version of the NN algorithm run using the simple pair merging variation (see section 3.2.3.3) was tried and performed so well that the NN spherical search variation was not pursued.

4.2.1 NN Speed vs. LBG Speed

In the version of the simple NN algorithm tested, each bucket found its pair which when merged would create the least error, and exactly the half of the buckets with the pairs with the lowest merge error had their pairs merged at any given pass. This one half figure was arbitrary and done (as opposed to merging all the best pairs) to prevent buckets with no good pairs from getting a merge. When a bucket got too small (contained three or fewer points) it was eliminated and its remaining points redistributed. Any time a merge of cost zero (two identical points) was discovered, the merge was performed, independent of the other merges.

Since buckets are kept at a small constant size (from 3 to 12 entries), searching within a bucket is a constant computational cost. Using the notation from above, during the algorithm initially there are T clusters (one for each training vector), and at the end there are C clusters, and since each merge reduces the number of clusters by one, there must be $(T - C)$ merges, which is approximately equal to T merges, since the whole point of the clustering is to make T much greater than C.

Now, since on each pass each bucket gets searched for its closest pairs, and since half the buckets get their pair merged on each pass, the number of buckets searched is approximately $2 * T$, or of order T. This means that the NN clustering algorithm is linear in the size of the training set and independent of the size of codebook generated (even though larger codebooks take slightly less time than small ones). Of course, one still has to worry about unfavorable constants and the overhead of using a K-d tree, etc., but actual results (see table below) show extremely good performance. Recall that for these examples, the training set is the picture to be coded itself (T = 16,384 for 512x512 pictures and 4x4 codes), although in practice (although not necessarily) the training set would be composed of several pictures which belong to a certain "class of pictures".

Execution time (in Berkeley 4.2BSD Unix accounting units)			
Picture	Codebook Development Algorithm		
	LBG with random initialization		simple NN
	iterations	time (secs)	time (secs)
baboon	17	9273.0	312.0
lake	30	9462.6	320.8
airport	18	10148.9	321.9
lena	32	8344.0	321.1
peppers	21	4628.5	317.7
plane	25	5419.8	341.7

TABLE 2. Execution time of LBG vs. NN

4.2.2 NN Quality vs. LBG Quality

4.2.2.1 Numerical Pixel Error

A very important question at this point is how good is the performance of the NN algorithm in terms of the pictures it results in. After all, the algorithm is useless if it produces bad pictures, even though it may be extremely fast. It should be noted that the codebooks generated by the NN algorithm are necessarily sub-optimal, since the LBG algorithm can always improve on them by running a few iterations. On the other hand, the LBG algorithm nevers guarantees optimality either. What is desired is a fast alternative giving performance equivalent to LBG performance, and as the results (see table) show, quality is comparable, and even better in many cases.

Squared Pixel Error for Reconstructed Pictures (x 10^7)			
Picture	Codebook Development Algorithm		
	LBG initialized with random training vectors	simple NN	LBG initialized with simple NN codes
baboon	6.565	7.215	6.542
lake	2.674	2.655	2.371
airport	1.718	1.539	1.379
lena	1.165	1.253	1.101
peppers	1.561	1.343	1.195
plane	1.549	1.436	1.250

TABLE 3. Coded Picture Error of LBG vs. NN

Admittedly, squared error is a poor error criterion, but since that is what is used in the codebook generation algorithm to "optimize" the codes, it seems appropriate. In any case, various versions of a typical coded picture are shown below along with blowups to show detail (see figures), and additional pictures are reproduced in an appendix. These pictures support the results reported with mean square criterion.

4.2.2.2 User Perceived Error

In the pictures that were coded, various subjective quality issues were observed which were not necessarily captured in the numerical quality ratings. Specifically, pictures coded with the LBG codebooks tended to have more objectionable contouring in the "flat" smoothly varying regions. In addition, when the picture was "noisy" (such as the fur on the 'Baboon' picture) both methods produced excellent reproductions, while (predictably) vector quantization coded pictures performed more poorly in general on pictures with lots of sharp edges and detail. These comments are, of course, far from a rigorous subjective evaluation, but are intended solely as an aid to evaluating the effectiveness of NN vs. LBG codebook development, and as an aid to evaluate the effectiveness of vector quantization picture coding in general.

Figure 13. 'Peppers' Original

Figure 14. 'Peppers' Coded with LBG Codebook

Figure 15. 'Peppers' Coded with Simple NN Codebook

Figure 16. Blowup (256x256 pixels) of 'Peppers' Coded with LBG Codebook

Figure 17. Blowup (256x256 pixels) of 'Peppers' Coded with Simple NN Codebook

4.2.3 NN Codebooks vs. LBG Codebooks

A feature of the NN algorithm is its tendency to come up with codebooks containing more edges, since it is less likely to cluster together blocks containing "edges" and blocks with more even gray level values. This is because with random initialization, there are less likely to be many "edge" codes (since edges typically compose only a small fraction of the picture) and the edges are then likely to get compromised by being grouped with other blocks from the training set. One might argue that it is the initialization of the LBG algorithm that is at fault here, rather the the algorithm itself, but the fact is that the initialization is such an important part of the algorithm that the algorithm *as a whole* deserves the "criticism". The inclusion of more edge code blocks is an advantage beyond simple squared error reduction, because poorly reconstructed edges are more offending to the observer than overall DC level errors.

4.2.4 NN as LBG Startup

As can be seen by the table above, when the output from the nearest neighbor (NN) clustering

algorithm was used as the initializer for the LBG algorithm, total coding error was lower than LBG with random initialization in all cases. This is an example of how very dependent the LBG algorithm is on its initialization step. In addition, with the NN initializer the LBG algorithm always converged in fewer iterations (often half as many), so the computation time overall was lower also, as one or two iterations alone take more time than the entire execution of the NN algorithm.

Consequently, it is clear that even if the NN algorithm were to be considered unacceptable because it usually does not generate the "optimal" codebook, it is at worst an excellent initializer for the LBG algorithm, and used this way results in excellent performance as well as computational savings over the LBG algorithm with random initialization.

4.2.5 Performance Outside Training Set

While every attempt will presumably be made to code pictures using a codebook developed with a training set representative of the pictures being coded, one might worry that all the above results might somehow be anomalous and not valid for pictures coded by codebooks developed on different training sets. Tests were performed to check on this possibility and it was determined that NN codebooks continue to perform comparably with LBG codebooks for pictures different from the sort the codebooks were designed for. In these cases, picture reconstruction was understandably worse, but suprisingly, coding with codebooks developed with a training set of vectors markedly different from the picture to be coded left the pictures still quite recognizable.

5. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

This thesis presented two fundamental improvements in the amount of computation needed to do vector quantization picture coding. Although these results were presented in the context of picture coding, the results are generalizable to any vector quantization system. A data structure (K-d trees by Bentley) was presented as a means of drastically reducing time spent in searching without sacrificing any performance.

A completely novel code generation algorithm was presented which generates codebooks comparable to those generated by the LBG algorithm in a significantly smaller fraction of the time needed for the LBG algorithm, even when the LBG algorithm is enhanced with K-d tree searching. It was shown that the new algorithm is both faster and increases in complexity more slowly than the existing codebook generation algorithm. Pictures generated with this new algorithm were shown to be at least as good as those generated with the old algorithm. The algorithm presented is also much more amenable to parallel implementations.

By using the codebooks generated by the new algorithm as a starting codebook for the LBG algorithm it was discovered that the LBG algorithm using random training vectors as an initializer typically converges to sub-optimal codebooks. It was determined that alternate representations for pixel intensity values in a block (computed keys) could be used as a basis for searching, but that with current implementations the actual time savings by such an approach were minimal, because of the costs involved in generating the computed keys.

Obvious enhancements of vector quantization picture coding include using entropy coding to exploit the wide variation in individual codebook entry usage and using some sort of block predictor analogous to single pixel predictive schemes to reduce the coding rate.

Coding of color images could also be done with no alteration of the existing coding algorithms. This could be done in two different ways. First, one might vector quantize luminance information

and chrominance information separately, allowing for more coarse (possibly subsampled) resolution for the chrominance data. Alternately, one might simply including the chrominance information within a block (subsampled as needed) with the luminance information already being processed. In this way, a 4x4 pixel block might actually consist of twenty to thirty numbers, some of which would represent color information. All facets of vector quantization could then proceed as before, with this increased dimensionality.

Another improvement would be to weight the blocks in the training set prior to developing a codebook on the basis of how well one wants them to be reproduced. In the NN algorithm this would mean initializing each cluster (when each entry in the training set is a cluster) with a weight perhaps greater than one. In picture applications this might correspond to assigning a weight to a block based on some sort of gradient or other edge detecting measure that the block exhibits, on the theory that it is important to correctly reproduce edges.

Finally, a very important issue to be researched is the possibility of using a single codebook, or set of general codebooks, to code a wide variety of images. Although with the developments of this paper it is more possible than ever to develop a special codebook for each image (a codebook that, with given parameters, would cost $\frac{1}{8}$ bit per pixel to transmit), it would be much more convenient if one could provide the decoder with a small set of codebooks ahead of time and then, after perhaps performing a few statistical tests to determine the "type" of picture being coded (faces vs. aerial photos for example), the coder would need only inform the decoder which codebook to use.

6. *REFERENCES*

Baker and Gray (1983a)
R. L. Baker and R. M. Gray,
"Image Compression Using Non-Adaptive Spatial Vector Quantization",
in Proc. 16th Asilomar Conf. on Circuits, Systems & Computers,
pp. 55-61, 1982.

Baker and Gray (1983b)
R. L. Baker and R. M. Gray,
"Differential Vector Quantization of Achromatic Imagery",
Proc. International Picture Coding Symposium,
UC-Davis, pp. 105-106, March 1983.

Bentley (1975)
J. L. Bentley,
"Multidimensional Binary Search Trees Used for Associative Searching",
Communications of the ACM,
Vol. 18, No. 9, pp. 509-517, Sept. 1975.

Buzo, Gray, Gray, and Markel (1980)
A. Buzo, A. H. Gray Jr., R. M. Gray, J. D. Markel,
"Speech Coding Based Upon Vector Quantization",
IEEE Trans. on ASSP,
Vol. ASSP-28, No. 5, pp. 562-574, Oct. 1980.

Chen and Smith (1977)
W.-H. Chen and C. H. Smith,
"Adaptive Coding of Monochrome and Color Images",
IEEE Trans. on Communications,
Vol. COM-25, No. 11, pp. 1285-1292, Nov. 1977.

Cheng, Gersho, Ramamurthi, and Shoham (1984)
D.-Y. Cheng, A. Gersho, B. Ramamurthi, and Y. Shoham,
"Fast Search Algorithms for Vector Quantization and Pattern Matching",
Proc. ICASSP,
pp. 9.11.1-9.11.4, San Diego, March 1984.

Friedman, Bentley, and Finkel (1977)
J. H. Friedman, J. L. Bentley, and R. A. Finkel,
"An Algorithm for Finding Best Matches in Logarithmic Expected Time",
ACM Trans. on Mathematical Software,
Vol. 3, No. 3, pp. 209-226, Sept. 1977.

Gersho (1982)
A. Gersho,
"On the Structure of Vector Quantizers",
IEEE Trans. on Information Theory,
Vol. IT-28, No. 2, pp. 157-166, March 1982.

Gersho and Ramamurthi (1982)

A. Gersho and B. Ramamurthi,
"Image Coding Using Vector Quantization",
in Proc. of IEEE Conf. on Acoustics, Speech and Signal Processing, Paris,
pp. 428-431, Paris, France, 1982.

Gowda and Krishna (1978)
K. C. Gowda and G. Krishna,
"Agglomerative Clustering Using the Concept of Mutual Nearest
Neighborhood",
Pattern Recognition,
Vol. 10, pp. 105-112, 1978

Gray (1984)
R. M. Gray,
"Vector Quantization",
to appear in IEEE Trans. on ASSP,
1984.

Hartigan (1975)
J. A. Hartigan,
"Clustering Algorithms",
Wiley, 1975.

Juang and Gray (1982)
B.-H. Juang and A. H. Gray Jr.,
"Multiple Stage Vector Quantization for Speech Coding",
Proc. IEEE Conf. on Acoustics, Speech & Signal Processing,
Paris, Vol. 1, pp. 597-600, May 1982.

King and Nasrabadi (1983)
R. A. King and N. M. Nasrabadi,
"Image Coding Using Vector Quantization in the Transform Domain",
Pattern Recognition Letters,
Vol. 1, Numbers 5-6, pp. 323-329, July 1983.

Knuth (1973)
D. E. Knuth,
"The Art of Computer Programming - Vol. 3 Sorting and Searching",
Addison, 1973.

Linde, Buzo, and Gray (1980)
Y. Linde, A. Buzo, and R. M. Gray,
"An Algorithm for Vector Quantizer Design",
IEEE Trans. on Communications,
Vol. COM-28, No. 1, pp. 84-95, January 1980.

Lloyd (1957)
S. P. Lloyd,
"Least Squares Quantization in PCM",
IEEE Transactions on Information Theory,
Vol. IT-28, No. 2, pp. 129-137, March 1982 (reprint).

Murakami, Asai, and Yamazaki (1982)
T. Murakami, K. Asai, and E. Yamazaki,
"Vector Quantiser of Video Signals",
Electronics Letters,
Vol. 18, No. 23, pp. 1005-1006, Nov. 1982.

Murakami, Asai, and Yamazaki (1983)
T. Murakami, K. Asai, and E. Yamazaki,
"A Design of Vector Quantizer for Video Signals",
Proc. International Picture Coding Symposium,
UC-Davis, pp. 25-26, March 1983.

Nasrabadi and King (1983)
N. M. Nasrabadi and R. A. King,
"Transform Coding Using Vector Quantization",
in Proc. Conference on Information Science and Systems,
March 1983.

Rabin (1976)
M. O. Rabin,
"Probabilistic Algorithms",
in Algorithms and Complexity, J. F. Traub, ed.,
pp. 21-39, New York: Academic Press, 1976.

Ramamurthi and Gersho (1984a)
B. Ramamurthi and A. Gersho,
"Edge-Oriented Spatial Filtering of Images With Application to
Post-Processing of Vector Quantized Images",
Proc. ICASSP 84,
San Diego, pp. 48.10.1-48.10.4, March 1984.

Ramamurthi and Gersho (1984b)
B. Ramamurthi and A. Gersho,
"Image Vector Quantization With A perceptually-Based Cell Classifier",
Proc. ICASSP 84,
San Diego, pp. 32.10.1-32.10.4, March 1984.

Ramamurthi, Gersho, and Sekey (1983)
B. Ramamurthi, A. Gersho, and A. Sekey,
"Low-rate Image Coding Using Vector Quantization",
Proc. IEEE Global Telecommunications Conference,
pp5.6.1-5.6.4, San Diego, 1983.

Roucos, Schwartz, and Makhoul (1983)
S. Roucos, R. M. Schwartz, J. Makhoul,
"A Segment Vocoder at 150 B/S"
in Proc. ICASSP 83,
pp. 61-64, Boston, 1983.

Shannon (1948)
C. E. Shannon,

"Mathematical Theory of Communication",
Bell System Technical Journal,
July and October, 1948.

Yamada, Fujita, and Tazaki (1980)
Y. Yamada, K. Fujita, and S. Tazaki,
"Vector Quantizer of Video Signals",
Proc. of Annual Conference of IECE,
p. 1031, 1980.

Yuval (1975)
G. Yuval,
"Finding Near Neighbours in K-Dimensional Space",
Information Processing Letters,
Vol. 3, No. 4, pp. 113-114, March 1975.

Yuval (1976)
G. Yuval,
"Finding Nearest Neighbours",
Information Processing Letters,
Vol. 5, No. 3, pp.63-65, August 1976.

7. APPENDICES

7.1 Algorithms In General Language Notation

7.1.1 Building K-d Trees

```
/* Function which returns a node created out of the given list of records
 */

make_tree(records[], numrecords)

{
   if (numrecords <= BUCKETSIZE)
   return(make_bucket(records[], numrecords));

   node_key = next_key(records[], numrecords);

   splitval = median(node_key, records[], numrecords);

   new_tree = make_node(node_key, splitval,
         make_tree(l_list[], l_count),
         make_tree(r_list[], r_count));
   return (new_tree);
}
```

7.1.2 Searching K-d Trees

```
/* Routine to initialize search parameters and start recursive searching
 */

tree_search(root)

{
   best_distsq = INFINITY;

   for (i = 0; i < num_keys; i++)
   {
         keymax[i] = KEYMAX;
         keymin[i] = KEYMIN;
   }

   done_flag = FALSE;

   node_search(root);
}

/* Recursive routine to search to a best batch to a query record
 */

node_search(node)
```

```
{
   if  (node is really a bucket)
   {
         bucket_search(node);
         if (ball_within_bounds())
         done_flag = TRUE;
         return(0);
   }

   if (query_rec.keys[node_keynum] < node_keyval)
   {
         tempkeylim = keymax[node_keynum];
         keymax[node_keynum] = node_keyval;
         node_search(node -> l_child);
         if (done_flag) return(0);
         keymax[node_keynum] = tempkeylim;
   }
   else
   {
         tempkeylim = keymin[node_keynum];
         keymin[node_keynum] = node_keyval;
         node_search(node -> r_child);
         if (done_flag) return(0);
         keymin[node_keynum] = tempkeylim;
   }

   if (query_rec.keys[node_keynum] < node_keyval)
   {
         tempkeylim = keymin[node_keynum];
         keymin[node_keynum] = node_keyval;
         if (bounds_overlap_ball())
         {
            node_search(node -> r_child);
            if (done_flag) return(0);
         }
         keymin[node_keynum] = tempkeylim;
   }
   else
   {
         tempkeylim = keymax[node_keynum];
         keymax[node_keynum] = node_keyval;
         if (bounds_overlap_ball())
         {
            node_search(node -> l_child);
            if (done_flag) return(0);
         }
         keymax[node_keynum] = tempkeylim;
   }

   if (ball_within_bounds())
   done_flag = TRUE;
```

```
    return(0);
}

/* Routine to determine if all necessary buckets have been checked
 */

ball_within_bounds()

{
    for (i = 0; i < num_keys; i++)
    {
        keyterm = query_rec.keys[i] - keymax[i];
        if (keyterm * keyterm <= best_distsq)
        return(FALSE);
        keyterm = query_rec.keys[i] - keymin[i];
        if (keyterm * keyterm <= best_distsq)
        return(FALSE);
    }

    return(TRUE);
}

/* Routine to determine if buckets on the opposite side of a node need
   to be searched
 */

bounds_overlap_ball()

{
    for (i = 0; i < num_keys; i++)
    {
        keydif = keymin[i] - query_rec.keys[i];
        if (keydif > 0)
        {
            sum += keydif * keydif;
            if (sum > best_distsq)
            return(FALSE);
        }
        keydif = query_rec.keys[i] - keymax[i];
        if (keydif > 0)
        {
            sum += keydif * keydif;
            if (sum > best_distsq)
            return(FALSE);
        }
    }

    return(TRUE);
}
```

7.1.3 LBG Algorithm

```
/* Routine to implement iterative lbg algorithm
 */

#define EPSILON .001 /* change in codes needed to demand another iteration */

lbg()
{
   iterations = 0;
   new_error = INFINITY;
   error_test = EPSILON + 1.;
   codeinit(partitions, dimension, trainsize);

/* basic loop to implement algorithm
 */

   while (error_test > EPSILON)
   {
         iterations += 1;
         old_error = new_error;
         new_error = 0.;
         for (i = 0; i < trainsize; i++)
         {
            minerror = L_INFINITY;
            for (j = 0; j < partitions; j++)
            {
         error = distance(query_rec, trainers[j]);
         if (error < minerror)
         {
            minerror = error;
            best_code = j;
         }
            }

            trainers[i].code = best_code;
            new_error += minerror;
         }

         if (new_error == 0.) error_test = 0.;
         else                 error_test = (old_error - new_error) / new_error;
         if (error_test > EPSILON) fix_codes(partitions, dimension, trainsize);
   }
}
```

7.1.4 Yuval NN Clustering

```
/* Routine to implement yuval shifting of partitions to detect all nearby pairs
 */
```

```
yuval_cluster(root, partitions)
{
   delta_factor = number_of_dimensions + 1;

   while(num_clusters > partitions)
   {
         if (old_clusters == num_clusters)
         "eliminate smallest bucket and redistribute points it contained";

         old_clusters = num_clusters;
         min_d = smallest_bucket_dimension(root);

         thresh = min_d / delta_factor;
         threshsq = (min_d * min_d) / (delta_factor * delta_factor);

         num_shifts = delta_factor;

         for (i = 0; i < num_shifts; i++)
         {
            find_low_pairs(bucket_list, num_buckets, threshsq);

            shift_tree(root, thresh);
         }

/* shift the partitions defined by the K-d tree over by a constant amount
  in each dimension
 */
         shift_tree(root, -thresh * num_shifts);

         if ((num_clusters - num_pairs) < partitions)
         sort(pair_list);

/* Merge together all pairs below threshold distance
 */
         root = mst_merge(partitions, root, pairs, num_pairs);
   }
   return(root);
}
```

7.1.5 Simple NN Clustering

```
/* Routine to implement simple version of NN clustering algorithm
 */

 /* fraction of buckets that get their "best pair" merged */
#define MERGEFRACTION .5

mst_cluster(root, num_clusters_wanted)

{
```

```
   while (num_clusters > num_clusters_wanted)
   {
/* get list of closest pair of points within each bucket
 */
        num_pairs = best_pairs(bucket_list, num_buckets);

/* merge all pairs of points with zero distance even if they are not the
   bucket's "best pair"
 */
        root = mst_merge(num_clusters_wanted, root, z_pairs, z_num_pairs);

/* sort pairs by distance
 */

        sort(pairs, num_pairs);

        num_merges = MERGEFRACTION * (float) num_pairs;

/* Merge fraction of best pairs
 */
        root = mst_merge(num_clusters_wanted, root, pairs, num_merges);
   }
   return(root);
}

/* Routine to merge two points
 */

mst_merge(num_clusters_wanted, root, pt_pairs, num_pairs)

{
   for (j = 0; (j < num_pairs) && (num_clusters > num_clusters_wanted); j++)
   {
        pt_1 = pt_pairs[j].point_one;
        pt_2 = pt_pairs[j].point_two;

        if(pt_1 and pt_2 in different clusters)
        {
           num_clusters -= 1;

           for (i = 0; i < dimension; i++)
           pt_1 -> data[i] =
        (pt_1 -> weight * pt_1 -> data[i] +
        pt_2 -> weight * pt_2 -> data[i]) /
        (pt_1 -> weight + pt_2 -> weight);

           pt_1 -> weight += pt_2 -> weight;
           pt_2 -> weight = 0;

           root = tree_delete(pt_1, root);
           root = tree_delete(pt_2, root);
```

```
            pt_2 -> next_rec = pt_1;

            root = tree_insert(root, &pt_1);
        }
    }
    return(root);
}
```

7.2 Distance Constancy with Orthogonal Transforms

Let $\vec{x}_1$ = *N-dimensional vector 1*
$\vec{x}_2$ = *N-dimensional vector 2*
$\mathbf{T}$ = *linear orthogonal transform (N x N matrix)*
$\vec{X}_1 = \mathbf{T}\vec{x}_1$ = *transformed vector 1*
$\vec{X}_2 = \mathbf{T}\vec{x}_2$ = *transformed vector 2*

d_{12}^2 = *distance between* $\vec{x}_1$ *and* $\vec{x}_2$
$= |\vec{x}_1 - \vec{x}_2|^2$
$= (\vec{x}_1 - \vec{x}_2)^T(\vec{x}_1 - \vec{x}_2)$
D_{12}^2 = *distance between* $\vec{X}_1$ *and* $\vec{X}_2$
$= |\vec{X}_1 - \vec{X}_2|^2$
$= (\vec{X}_1 - \vec{X}_2)^T(\vec{X}_1 - \vec{X}_2)$

Now, $D_{12}^2 = (\vec{X}_1 - \vec{X}_2)^T(\vec{X}_1 - \vec{X}_2)$
$= (\mathbf{T}\vec{x}_1 - \mathbf{T}\vec{x}_2)^T(\mathbf{T}\vec{x}_1 - \mathbf{T}\vec{x}_2)$
$= (\mathbf{T}(\vec{x}_1 - \vec{x}_2))^T(\mathbf{T}(\vec{x}_1 - \vec{x}_2))$ *by linearity*

define $\Delta\vec{x} = \vec{x}_1 - \vec{x}_2$

so $d_{12}^2 = \Delta\vec{x}^T\Delta\vec{x}$
and $D_{12}^2 = (\mathbf{T}\Delta\vec{x})^T(\mathbf{T}\Delta\vec{x})$
$= \Delta\vec{x}^T\mathbf{T}^T\mathbf{T}\Delta\vec{x}$
$= \Delta\vec{x}^T(\mathbf{T}^T\mathbf{T})\Delta\vec{x}$

now $\mathbf{T}$ = *an orthonormal matrix with orthonormal rows, since each row is a basis function*

This implies that
$\mathbf{T}\mathbf{T}^T = \mathbf{I}$

But since $\mathbf{T}$ *is square, its right inverse is the same as its left inverse, and*
$\mathbf{T}^T = \mathbf{T}^{-1}$

therefore $\mathbf{T}^T\mathbf{T} = \mathbf{T}\mathbf{T}^T = \mathbf{I}$

$$\begin{aligned} \textit{Consequently } D_{\mathbf{T}}^2 &= \Delta x^T(\mathbf{T}^T\mathbf{T})\Delta x \\ &= \Delta x^T(\mathbf{I})\Delta x \\ &= \Delta x^T\Delta x \\ &= d_{\mathbf{I}}^2 \end{aligned}$$

7.3 Photographs of Coded Pictures

This appendix contains reproductions of the pictures used to test the effectiveness of the various algorithmic improvements described in the body of this presentation. In addition, they serve to test the performance of vector quantization picture coding in general on a wide variety of pictures. Each of five originals is reproduced as are the reconstructed pictures after being vector quantization coded with an LBG-derived codebook and with an NN-derived codebook.

All pictures coded with 256 4x4 codes for a rate of .5 bits per pixel. Unless specified, pictures were of size 512x512 pixels.

7.3.1 Baboon

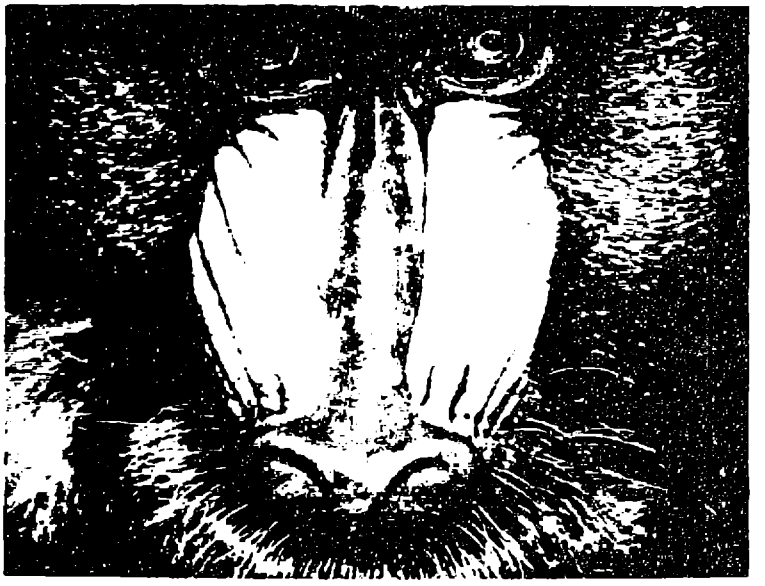

Figure 18. 'Baboon' Original

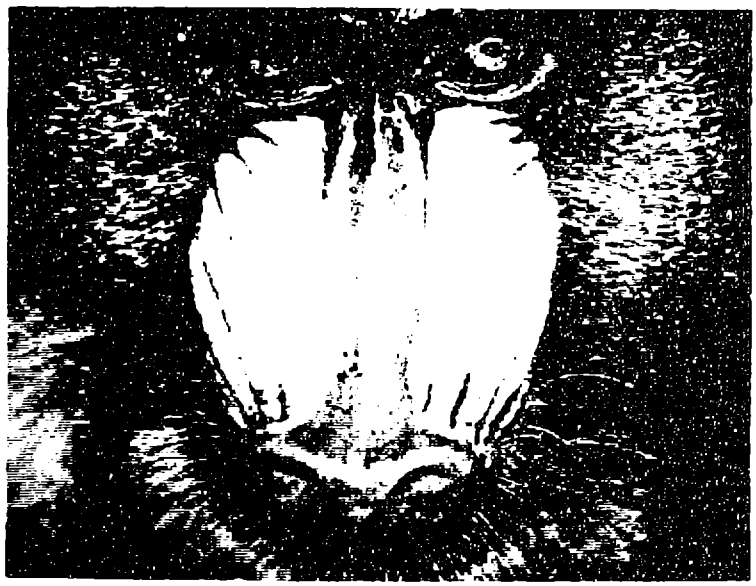

Figure 19. 'Baboon' Coded with LBG Codebook

Figure 20. 'Baboon' Coded with Simple NN Codebook

7.3.2 Lake

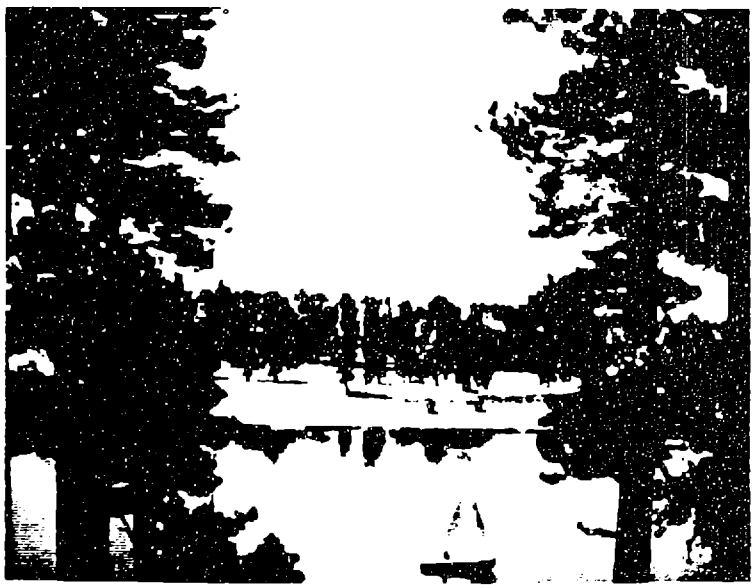

Figure 21. 'Lake' Original

Figure 22. 'Lake' Coded with LBG Codebook

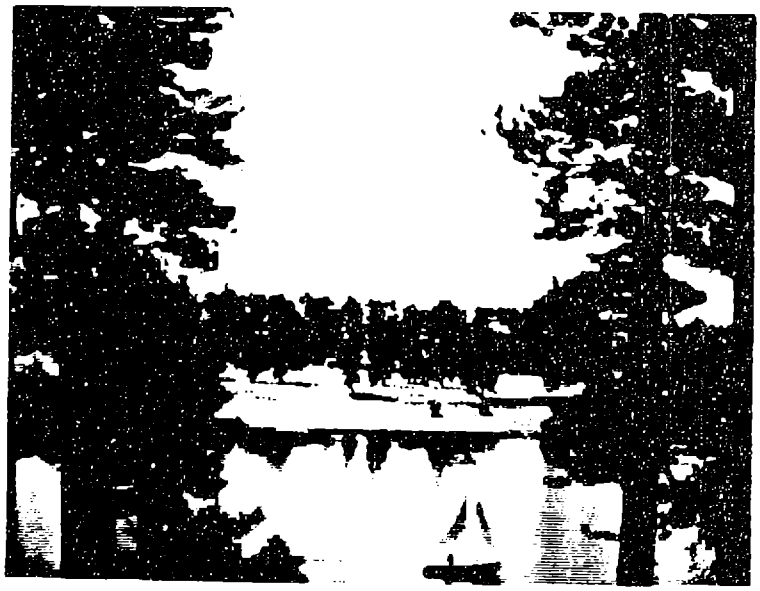

Figure 23. 'Lake' Coded with Simple NN Codebook

7.3.3 Airport

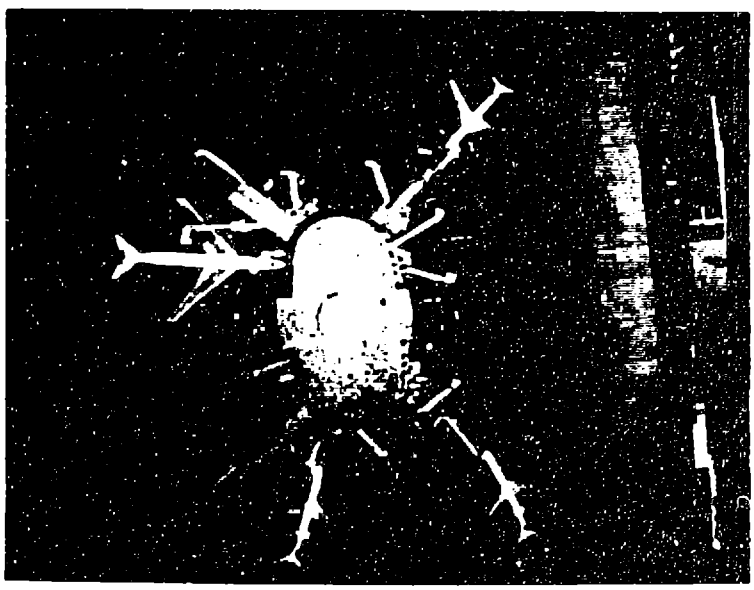

Figure 24. 'Airport' Original

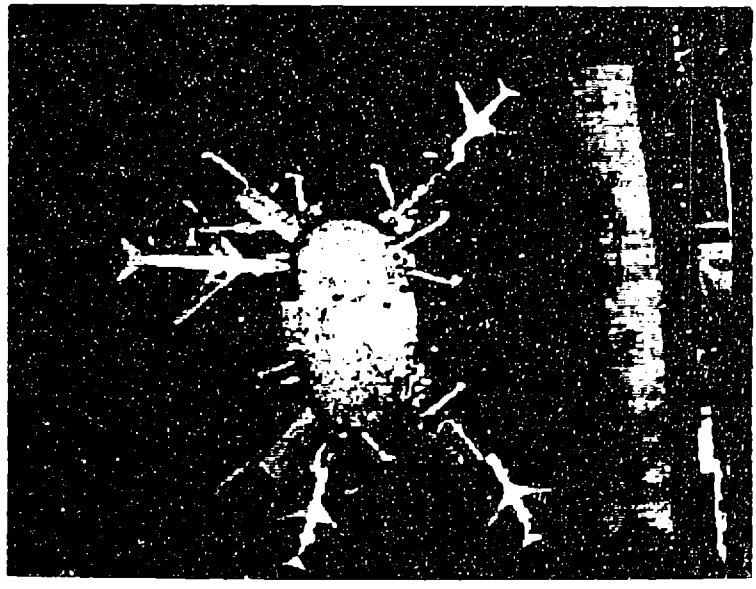

Figure 25. 'Airport' Coded with LBG Codebook

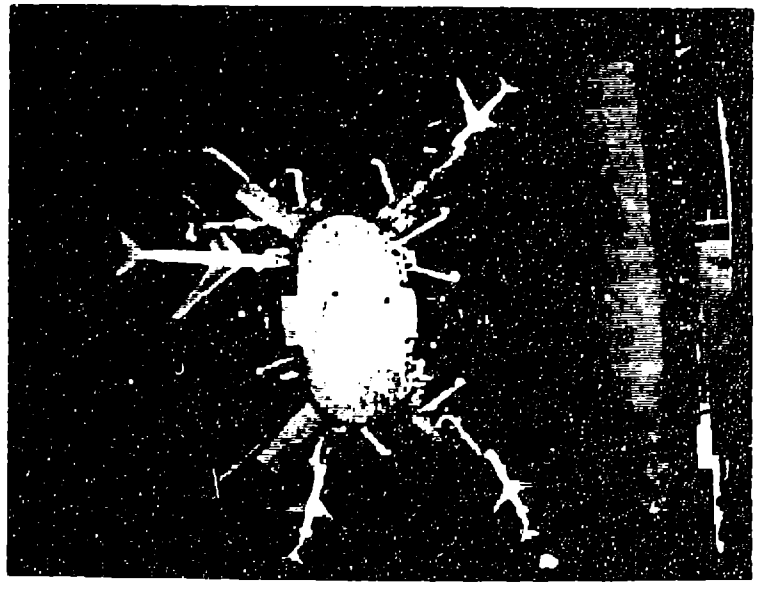

Figure 26. 'Airport' Coded with Simple NN Codebook

7.3.4 Lena

Figure 27. 'Lena' Original

Figure 28. 'Lena' Coded with LBG Codebook

Figure 29. 'Lena' Coded with Simple NN Codebook

7.3.5 Plane

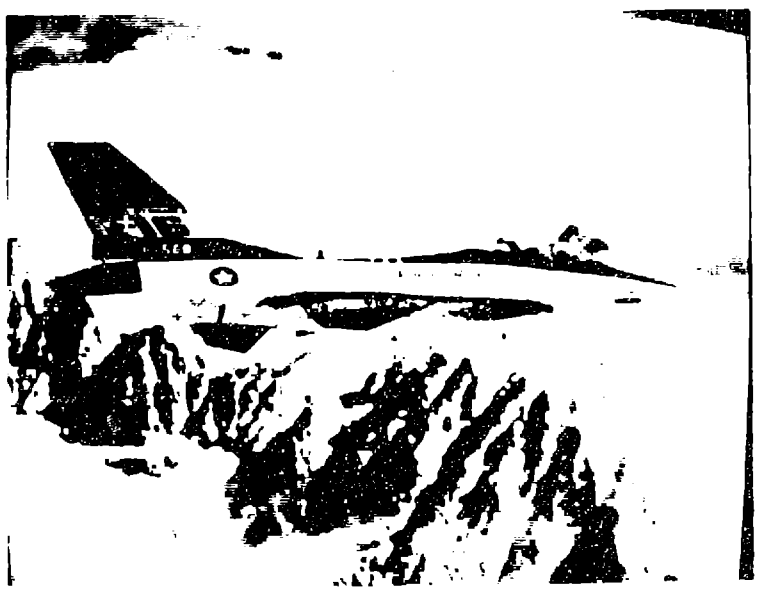

Figure 30. 'Plane' Original

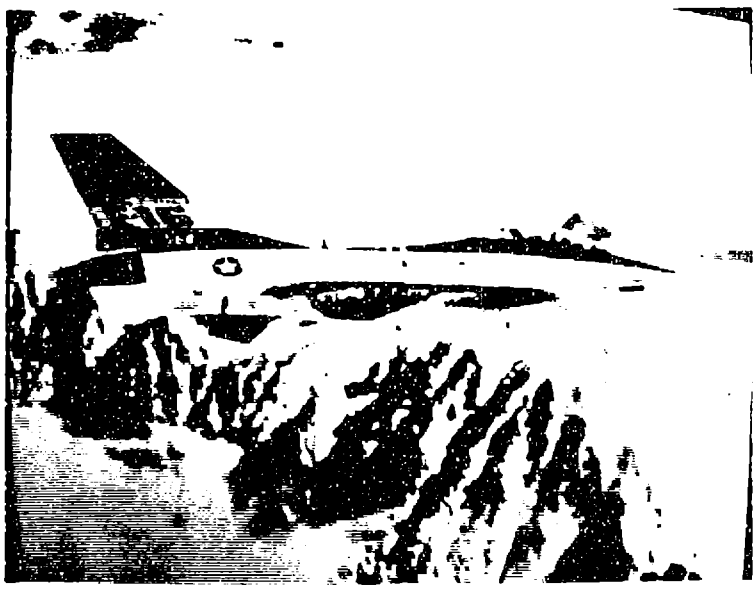

Figure 31. 'Plane' Coded with LBG Codebook

Figure 32. 'Plane' Coded with Simple NN Codebook