A comparison of codebook generation techniques for vector quantization

Robert F. Sproull[1]
Ivan E. Sutherland[1]
Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043

Successful vector quantization of images depends on constructing suitable codebooks: the quality of the final image depends critically on the quality of the codebook, and codebook construction techniques can be very slow. This paper examines tradeoffs between speed and quality of codebook-generation algorithms and offers new ways to produce excellent codebooks with only modest computation cost.

This paper compares the performance of four algorithms for constructing codebooks. The LBG method [7] produces the best codebooks but requires the most computation. The method by Equitz [2,3] produces codebooks nearly as good and requires somewhat less computation. This paper describes a new method based on eigenvector subdivision that produces useable codebooks in a fraction of the computational effort of either of the other methods. A fourth hybrid method yields very good codebooks with modest computation by using the eigenvector subdivision method to obtain a first approximation that is refined with LBG optimization.

Our eigenvector subdivision method divides the training set vectors into successively smaller sets based on the direction of the principal eigenvector of each subset. After each subdivision, a new principal eigenvector is computed for each subset to select a direction for the next subdivision. Because the principal eigenvector points in the direction of greatest variance in the subset, subdivision normal to the principal eigenvector breaks the subset into more nearly hyper-spherical pieces. When enough subsets have been formed, the centroid of each subset is used as a codeword.

This paper also offers some insight into the content of codebooks. We show a method for visualizing an n-dimensional codebook by drawing n^2 two-dimensional projections. These projections show clearly how little variance there is in the codebook in some directions, reflecting the small size of several eigenvalues of the training set data. Ignoring the thickness of the training set data in directions with small eigenvalues and assuming a uniform distribution in other directions leads to a simple estimate of RMS encoding errors for a vector quantizer. This estimate agrees quite well with measured RMS encoding errors.

1 Codebook geometry

Before turning to the algorithms, we first offer a pictorial view a codebook by plotting all possible two-dimensional views. Figures 1 and 2 show plots of a 16-dimensional

[1]Performed this work as consultants to Apple Computer.

0-8186-2717-4/92 $3.00 © 1992 IEEE

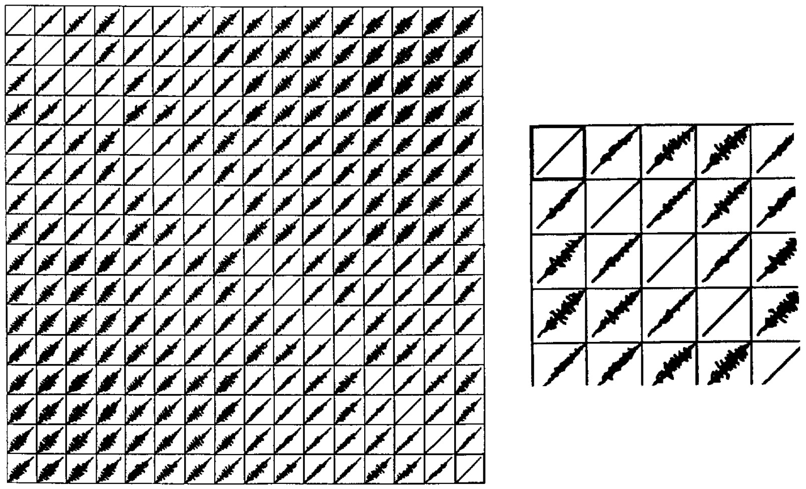

Figure 1: Visualization of a codebook for a 16-dimensional vector space. The plot in row i and column j plots values in the ith dimension against those in the jth dimension. The figure at the right is an enlarged view of the upper left corner of the figure at the left.

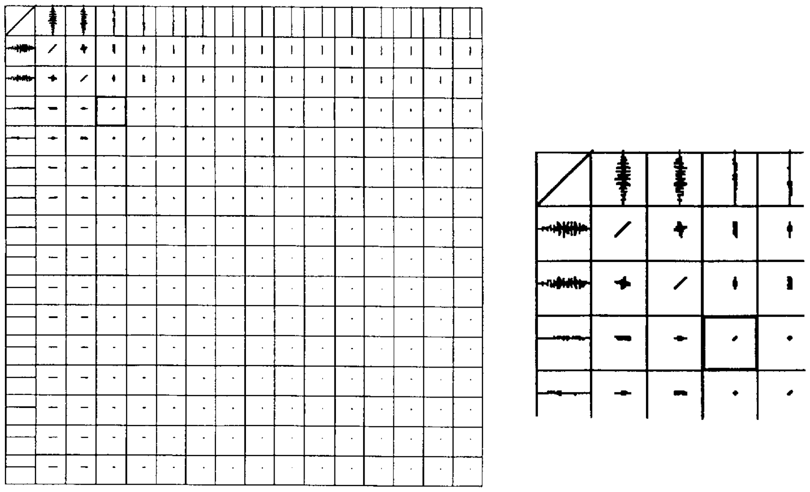

Figure 2: Visualization of a codebook for a 16-dimensional vector space after rotation by the Karhunen-Loève transformation.

vector quantizer codebook taken from a monochromatic image. Each 16-element vector contains the pixel values from a 4×4 square patch: the first four elements in the vector correspond to the top row of the patch, the next four to the next row, and so on. Since there are $16^2 = 256$ ways to choose two dimensions from 16 dimensions, there are 256 two-dimensional views.

Figure 1 shows a codebook plotted with axes that correspond to the intensity of particular pixels in the 4×4 patch. It is hard to make much out of this figure because the coordinates correspond to individual pixels and thus the shape of the distribution is concealed.

A better view of a codebook is obtained by first rotating the vector space so as to emphasize the directions of greatest variation of the codebook vectors. The rotation transformation, called the Karhunen-Loève transformation, is obtained by measuring the distribution of codewords by the eigenvectors and eigenvalues of its covariance matrix. The principal eigenvector, i.e., the one with the largest eigenvalue, points in the direction in which the distribution has greatest variance, and eigenvectors with successively smaller values point in mutually perpendicular directions in which the distribution has less and less variance. Thus the 16 eigenvectors represent the 16 orthogonal directions of a coordinate system that reveals the variance of the distribution. The Karhunen-Loève transformation rotates the original space so that these directions are aligned with coordinate axes.

The shape of a codebook rotated by the Karhunen-Loève transformation is clearly evident in Figure 2. The axes are shown in order of decreasing eigenvalue: the first dimension plotted is the direction of greatest variance in the data, the next dimension is a direction orthogonal to the first with next greatest variance, and so on. The fact that the distribution is long in one dimension and thin in others is quite evident. We think of such distributions as n-dimensional almonds, because like an almond they have a long axis, one or more medium length axes, and several short axes, and they are rounded or pointed at their ends.

2 Four algorithms for making codebooks

2.1 LBG algorithm

The LBG algorithm involves iterative refinement of a trial codebook and splitting of its codewords. Each refinement iteration has two steps, an *encoding* step and an *adjustment* step. The encoding step involves assigning each training set vector to the trial code vector closest to it, thus forming clusters, one associated with each code vector in the trial codebook. The adjustment step moves the trial codeword to the centroid of its cluster. To increase the number of codewords, one or more clusters with large distortion are split in two parts, usually by introducing a new codeword slightly displaced from the existing codeword.

A complete LBG algorithm applies the two-step refinement iteration and splitting a great many times. The algorithm must start with an initial trial codebook, and terminates when a refinement iteration fails to produce at least some minimal improvement in RMS encoding error of the training set. Many choices of initial codebook are possible. For example, purely random vectors or vectors chosen randomly from the training set may be used. Alternately, the output from another algorithm may be a good initial trial codebook [2].

The trouble with the LBG method is that it converges slowly. Because each refinement can only reduce the total distortion, the algorithm provably converges to a local optimum, but not necessarily the global optimum. Each refinement iteration produces a small improvement in the codebook but is slow because it must examine all of the training set vectors. Despite its large computational cost, our experiments indicate that the LBG algorithm routinely produces the best codebooks.

2.2 Equitz algorithm

Where the LBG algorithm builds up a codebook by adding codewords, the Equitz algorithm reduces the size of the training set gradually to produce the required codebook. The Equitz algorithm starts with the entire training set as a first approximation to the codebook, and reduces it by successive approximations. Each iteration finds the pair of vectors in the approximation that are closest together and combines them. The definition of close together weights previously combined clusters according to how many training-set vectors have been accumulated into them, so that "heavy" clusters will combine only if closer together than "light" ones. The effect is to perform the merge that will add the least distortion to the training set.

The algorithm hinges on a technique for finding the closest pair of vectors, which could be an expensive task. The obvious exhaustive search to find the closest pair of N vectors will require N^2 distance measurements. Equitz's method uses a faster approximation: he divides the training set into clusters of about 25 vectors each. Thereafter, the search for a vector's nearest neighbor is confined to the vectors in the same cluster as the vector, so that only slightly more than $25N$ distance measurements are required. The clusters are formed in a data structure called a k-d tree [1], which recursively splits the training set into clusters separated by cut planes aligned with a coordinate axis. We have developed a variant of Equitz's approach that uses a generalized k-d tree in which the cut planes are normal to the principal eigenvector of the distribution being cut [8].

2.3 Eigenvector subdivision algorithm

When we looked at pictures of codebooks such as Figure 2, we imagined finding clusters of training set data by recursively subdividing the data by cutting planes normal to the principal eigenvector. This notion led to devloping the generalized k-d tree [8] and a corresponding codebook-generation algorithm. A similar scheme, using cut planes aligned with axes, was used by Heckbert to build quantizers for color maps [6].

We think of the eigenvector subdivision method in geometric terms. The principal eigenvector of a distribution lies along its longest axis, so cutting the distribution near its center and normal to its principal eigenvector makes two more nearly hyper-spherical parts. If the principal eigenvalue is substantially larger than the next largest eigenvalue, the distribution is cigar-shaped, i.e., long in one direction, and we can expect more than one cut along nearly parallel planes, since cutting a long cigar in half leaves two pieces each of which is still cigar-shaped. In fact, the training set data we have examined using 4×4 squares of pixels as vectors is so highly correlated in the direction of average gray shade that the first eight or more cuts in the eigenvector method merely separate the data into groups of generally lighter and generally darker vectors, i.e., they cut normal to the "gray" axis.

If the two largest eigenvalues are substantially equal but larger than the third, the distribution is pancake-shaped. In this case it may be sensible to make two cuts at once, one normal to each of the two largest eigenvectors, but little harm is done by making them successively, as our programs do. The idea is always to cut in such a direction as to make the subdivided pieces more nearly hyper-spherical.

The eigenvector subdivision method places the training set data into a subdivision tree much like a k-d tree. Each subdivision, however, is made with a plane normal to the principal eigenvector of the data being split. Each node in the tree indicates the orientation and location of the cut plane and points to other tree nodes that further separate the data. The result is much like a k-d tree except that the orientation of the cut planes is data-dependent. In fact, our programs use the same code for eigenvector trees as for classical k-d trees, with a parameter to say how the orientation of the planes is to be selected. The algorithms for building and searching this kind of tree are given in detail in [8].

The algorithm builds the subdivision tree of the training-set vectors until there are as many leaf nodes as there are codewords in the required codebook. The centroid of the training set vectors in each leaf node of this tree then becomes the corresponding codeword.

The eigenvector method gives a rationale for selecting the orientation of each cut plane, but its location along the eigenvector can be chosen in many ways. We have experimented with two alternatives: pass the cut plane through the mean of the distribution being cut, or cut at the median value so as to place the same number of vectors on each side of the cut.

The eigenvector subdivision method for computing codebooks is very fast. It is fast because its only iterative processes operate on a very small amount of data, namely a covariance matrix. It is well adapted to run on vector processors because its major calculations are large inner products across the training set data. It makes only $O(logN)$ passes on the training set data to compute a codebook of size N. Finally, its output provides not only a codebook but also a set of subdivision planes suitable for use in a fast cut-plane encoding method.

2.4 Hybrid method

As we began to study the performance of the three algorithms described above, we looked for a compromise between the quality of the codebook produced by the slow LBG method and the speed of the eigenvector method. The obvious hybrid algorithm uses the fast eigenvector technique to build an initial codebook that is refined by the more accurate LBG algorithm. We ran several experiments in which the size of the initial codebook varies, i.e., changing how much subdivision is done by the eigenvector algorithm and how much splitting by the LBG algorithm. The hybrid technique is a good compromise between speed and quality.

3 Encoding

The job of encoding, or quantizing, an image is to find, for each vector in the image, the nearest codeword in the codebook. We have used two methods: a *full search* and a faster but less accurate *cut plane search.*

A full search of a codebook with N codewords would at first seem to require N distance measurements—an exhaustive linear search of the codebook. But if the codebook is loaded into a k-d tree data structure—either the classical form with axis-aligned cut planes or the form with cut planes normal to eigenvectors—the search is much faster. The speed of searches using the two kinds of trees is explored in [8].

When the codebook is produced using the eigenvector method, the cut planes of the subdivision tree that produced the codebook can be used to encode vectors. An image vector to be encoded is compared to the cut plane at the root node of the tree and the search proceeds to one of two child nodes depending on the relationship of the vector to the plane, and so on down the tree. When a leaf node is reached, the codeword associated with the leaf node is reported. Unlike full search, this technique does not always locate the codeword nearest to the data vector, so larger distortion results. In this respect, cut-plane searching is similar to *tree searching* of codebooks [5].

4 Experiments and results

To obtain credible performance measurements, we have coded all algorithms using one programming environment and one coding style. The three programs are written in C and use a common set of routines for accessing files, building and searching k-d trees, doing vector computations, computing averages, and so forth. The programs were written and debugged on a MicroVAX II and run against real data on a Cray YMP. No vectorization was used on the Cray.

The training set data consisted of five images supplied to us by Tom Stockham of the University of Utah. These images are called *park, parkcity, station, train,* and *tree;* they have appeared in other experiments. Each image is 512 pixels by 512 pixels in size carefully digitized from photographs. Each pixel is represented by a single 8-bit value representing log intensity. We used 4×4 square patches of pixels as vectors.

Table 1 summarizes experimental results. In each experiment we recorded the number of Cray seconds required and the RMS distortion for full search encoding the entire training set. For the eigenvector subdivision experiments we recorded also the RMS distortion for the faster cut-plane encoding. The table shows results only for codebooks with 1024 vectors; more complete results and discussion of algorithm variants can be found in a companion report [9].

The eight eigenvector subdivision experiments include all combinations of three binary variants. These are: (1) choice of which cluster to divide next, i.e., depth first *versus* worst distortion first; (2) orientation of cut planes in the k-d tree, i.e., cut planes running normal to eigenvectors *versus* cut planes normal to the coordinate axis with maximum data variance; and (3) location of the cut plane, i.e., cut plane positioned at the mean location of the cluster *versus* located at the median location. The best combination expands nodes with most distortion, uses eigenvector cut planes, and places the cut plane at the median of the cluster.

The seven Equitz algorithm experiments explore various ways of building the k-d tree used in its search: the size of buckets in terminal nodes of the tree, and the kind of cut planes used in the tree. Variants can build a tree of the entire training set ("infinite tiles") or can limit the number of tiles in memory before merge steps are performed. Not surprisingly, the least distortion is obtained when the entire training set participates in the merge, with large buckets, and cut planes that match the orientation of the data.

The LBG experiments differ in the settings of various parameters that control how new codewords are formed. The *split* parameter is the fraction of existing codewords that are candidates for splitting. The most interesting LBG experiment concerns splitting: how should the two new code vectors be related to the location of the code vector being split? In experiment 23 the codewords are aligned parallel to the principal eigenvector of the cluster being split rather than aligned randomly, as in all the other LBG tests. This not only reduces computing time by about 25% from the corresponding random vector experiment, number 17, but also gives the best codebook of any produced.

The final two experiments are hybrids. They both start with a codebook produced by the eigenvector method and then refine it using the LBG method. Experiment 24 uses an eigenvector codebook of the correct size, simply refining it by the LBG method, splitting codewords only when other codewords are deleted as too unpopular. It produces results in the same class as other LBG codebooks in about 1/4 the computing time. It is our best method in terms of its compromise between computing time and codebook quality. The final experiment primes the LBG method with a 256-word eigenvector codebook, but lets the LBG algorithm do further splitting. Again the RMS error is clearly like those of the other LBG algorithms, but so is the computing time. One should use the LBG algorithm to refine the location of codewords, not to split them.

Experiment	Description	Cray secs.	Distortion	Distortion (cut plane)
1	EV000: depth, eigenvector planes, median	189	7.52	8.10
2	EV001: depth, eigenvector planes, mean	150	7.09	7.47
3	EV010: depth, axis planes, median	150	7.81	9.34
4	EV011: depth, axis planes, mean	125	7.32	8.59
5	EV100: distortion, eigenvector planes, median	198	7.05	7.55
6	EV101: distortion, eigenvector planes, mean	156	<u>6.92</u>	7.31
7	EV110: distortion, axis planes, median	156	7.29	8.75
8	EV111: distortion, axis planes, mean	133	7.14	8.40
9	EQ000: infinite tiles, bucket (15,40), eigenvector planes	923	6.73	
10	EQ001: infinite tiles, bucket (15,40), axis planes	946	6.83	
11	EQ010: infinite tiles, bucket (30,80), eigenvector planes	2866	<u>6.68</u>	
12	EQ011: 8192 tiles, merge to 50%, bucket (15,40), eigenvector planes	718	6.80	
13	EQ100: 8192 tiles, merge to 90%, bucket (15,40), eigenvector planes	766	6.79	
14	EQ101: 4096 tiles, merge to 90%, bucket (15,40), eigenvector planes	762	6.79	
15	EQ110: 4096 tiles, merge to 50%, bucket (15,40), eigenvector planes	711	6.88	
16	LBG000: split 1.0, hit 0.1, delete 0.01	3304	6.56	
17	LBG001: split 0.5, hit 0.1, delete 0.01	3815	6.49	
18	LBG002: split 0.25, hit 0.1, delete 0.01	4755	6.50	
19	LBG003: split 0.125, hit 0.1, delete 0.01	5705	6.49	
20	LBG004: split 0.00625, hit 0.1, delete 0.01	43289	6.47	
21	LBG005: split 0.5, hit 0.0, delete 0.0	3794	6.49	
22	LBG006: split 0.5, hit 0.25, delete 0.125	3851	6.49	
23	LBG009: split 0.5, hit 0.1, delete 0.01, eigenvector	2961	<u>6.47</u>	
24	HY001: EV101+LBG001 refinement	1169	6.51	
25	HY002: EV101 256-word codebook+LBG001 split & refinement	2534	<u>6.50</u>	

Table 1: Experimental results for variants of four codebook-generation algorithms for 1024-vector codebooks: eigenvector (1–8), Equitz (9–15), LBG (16–23), and hybrid (24–25). The best results for each algorithm are underlined. The last column gives the distortion if the cut planes generated by the eigenvector algorithm are used for encoding rather than a "full search" of the codebook. Experiment 24 yields an excellent codebook with only modest computation.

Dim. n	Eigenvalue u	Square root $u^{1/2}$	Running product v	$(v/s)^{1/n}$ $s = 256$	$(v/s)^{1/n}$ $s = 1024$	$(v/s)^{1/n}$ $s = 4096$
1	31121.03	176.41	1.76×10^{2}	0.69	0.17	0.04
2	906.56	30.11	5.31×10^{3}	4.56	2.28	1.14
3	750.37	27.39	1.45×10^{5}	8.28	5.22	3.29
4	260.40	16.14	2.35×10^{6}	9.79	6.92	4.89
5	192.39	13.87	3.26×10^{7}	10.49	7.95	6.03
6	169.65	13.02	4.24×10^{8}	$\underline{10.88}$	8.63	6.85
7	83.88	9.16	3.88×10^{9}	10.61	$\underline{8.71}$	7.14
8	67.45	8.21	3.19×10^{10}	10.28	8.64	7.27
9	60.02	7.75	2.47×10^{11}	9.96	8.54	7.32
10	58.14	7.62	1.88×10^{12}	9.70	8.44	$\underline{7.35}$
11	28.23	5.31	1.00×10^{13}	9.18	8.09	7.14
12	26.88	5.18	5.19×10^{13}	8.76	7.80	6.95
13	21.76	4.66	2.42×10^{14}	8.34	7.50	6.74
14	12.30	3.51	8.49×10^{14}	7.84	7.10	6.43
15	11.36	3.37	2.86×10^{15}	7.41	6.76	6.16
16	5.97	2.44	6.99×10^{15}	6.91	6.34	5.81

Table 2: Estimates of distortion for different codebook sizes. The inputs to the computation are the sixteen eigenvalues in the second column. Underlined in the last three columns are estimates of distortion for three different codebook sizes. Larger codebooks populate more of the "dimensions" of the distribution and thus have lower distortion.

5 A method for estimating codebook quality

The eigenvalues computed from a training set can be used to produce an estimate of the distortion that a good codebook of size s will introduce when encoding the training set. This estimate is based on a geometric argument and the fact that the eigenvalues computed from a distribution tell quite a lot about its geometry. In particular, if the distribution is a glob of data, perhaps extended in some directions more than in others, the eigenvalues give us a way to approximate the volume of the glob. The values in the second column of Table 2 are the eigenvalues of the covariance matrix of the training set vectors used in these experiments, sorted in order of decreasing value. The third column gives their square roots so as to measure RMS distortion. The product of square roots of the largest n eigenvalues, as shown in the next column, is approximately the n-dimensional volume, v, of the glob in the n-dimensional space of the corresponding eigenvalues.

A well-formed codebook of size s should fill the n-dimensional volume v. If we assume that image vectors are evenly distributed throughout this volume (which they are not), the codebook divides the volume v into s volumes of equal size, each occupying volume v/s. Moreover, those volumes will be roughly hyper-spherical, and so the average encoding error should be on the order of $(v/s)^{1/n}$, as shown in the fourth column in Table 2. Notice that the numbers in this column increase to a maximum and then decrease again.

The only question that remains is how many dimensions to consider for making an estimate of RMS error. In directions for which the square root of the eigenvalue is less than the expected encoding error, the volume associated with a codeword will stretch completely across the training set distribution. In other words, in such dimensions the distribution is so thin that its thickness will no longer contribute to encoding errors. We can thus ignore the effect of this and other dimensions with smaller eigenvalues. Thus the maximum value in the column of predicted RMS errors in Table 2 is the one to choose. These estimates compare quite well with distortions of actual codebooks.

6 Acknowledgements

We wish to thank Professor Thomas Stockham for his instruction in vector quantization techniques and for providing us with the image data we used. We thought of ourselves as his non-resident post-doctoral students. We also wish to thank Apple Computer for its support of this work and for providing the computing facilities required. Particular thanks to Al Alcorn and Larry Tesler of Apple whose encouragement and support made the project possible.

References

[1] J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Comm. ACM*, 18(9):509–517, September 1975.

[2] W. Equitz. Fast Algorithms for Vector Quantization Picture Coding. *ICASSP*, 18.1.1, 1987.

[3] W. Equitz. A New Vector Quanitization Clustering Algorithm. *IEEE Trans. ASSP*, 37(10):1568–1575, October 1989.

[4] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Software*, 3(3):209–226, September 1977.

[5] R.M. Gray. Vector Quantization. *ASSP Magazine*, p. 4–29, April 1984.

[6] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics*, 16(3):297–307, July 1982.

[7] Y. Linde, A. Buzo, and R.M. Gray. An algorithm for vector quantizer design. *IEEE Trans. Comm.*, COM-28(1):84–95, January 1980.

[8] R.F. Sproull. Refinements to nearest-neighbor searching in k-d trees. *Algorithmica*, 6:579–589, 1991.

[9] R.F. Sproull and I.E. Sutherland. A comparison of codebook generation techniques for vector quantization. Technical report, Advanced Technology Group, Apple Computer, 20525 Mariani Ave., Cupertino, CA, 95014, 1992.