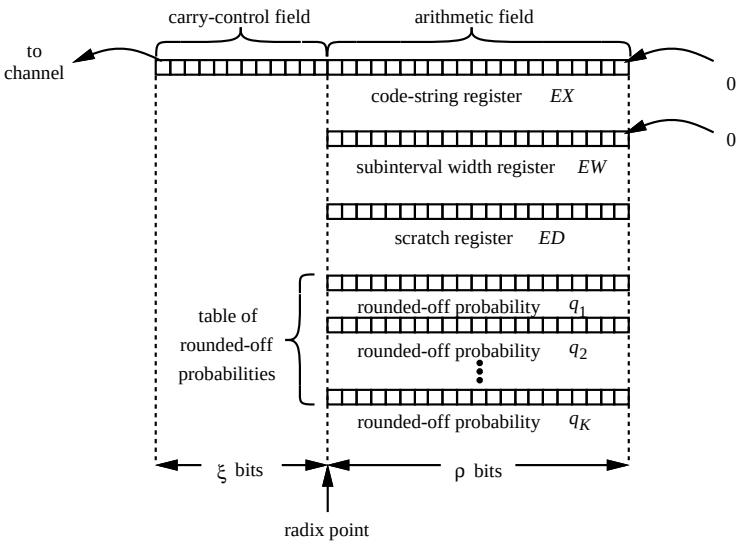
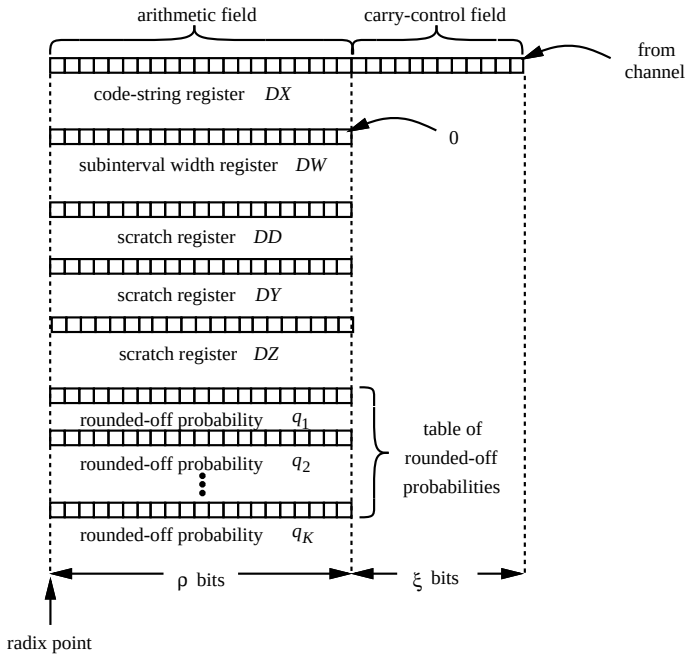


(a) Encoder registers.



(b) Decoder registers.

Figure 3.18: Register arrangement for a microprocessor implementation of (a) a K-ary arithmetic encoder, and (b) the corresponding decoder. The curved arrows indicate the direction in which bits are shifted during the renormalization process. The radix point indicates the separation between the integer and fractional parts of the registers — the first bit to the right of the radix point corresponds to a value of 1/2; the second bit, 1/4; and so forth. See Appendix C for a software implementation of the encoder/decoder pair.

The decoder has a few more registers, since more computations are interdependent, and consequently more intermediate results require storage.

Both the encoder and decoder are straightforward implementations of the recursion formulas presented earlier in this chapter. The interested reader is referred to the software presented in Appendix C, which explains the encoding and decoding algorithms in great detail.

It is worthwhile to elaborate on the manner in which carry-overs are handled. The remainder of this section discusses that.

Note that EX and DX in Figure 3.18 consist of $\rho + \xi$ bits, while other registers have only ρ bits. In EX, the extra ξ bits — called the *carry-control field* — between the radix point and the channel in EX provides a means of blocking carry-overs from propagating into the channel, where they cannot be handled. The general strategy is to check the carry-control field at every recursion to see if all ξ of its bits are ones. If so, then a zero bit is effectively inserted or "stuffed" between the radix point and the least-significant bit in the carry-control field, by shifting the carry-control field left by one bit position (transmitting its most significant bit in the process), and setting its least significant bit to zero. The inserted zero, here called the *carry-trap bit*, prevents propagation of any carry-overs that might occur during subsequent computations of X_i.

The decoder, upon detecting a string of ξ consecutive ones in the least-significant part of DX, interprets the bit that immediately follows as the carry-trap bit. If that bit is a one, then it can be concluded that a carry-over had occurred during encoding. The decoder can, in effect, complete the addition that the encoder had initiated by adding the carry-trap bit to the least significant bit in the string of ξ consecutive ones.

A slight complication arises because it is possible for the code-bit stream to contain ξ consecutive ones even if those ones never passed contiguously through register EX. In particular, this can happen if a string of $\xi - 1$ consecutive ones has been

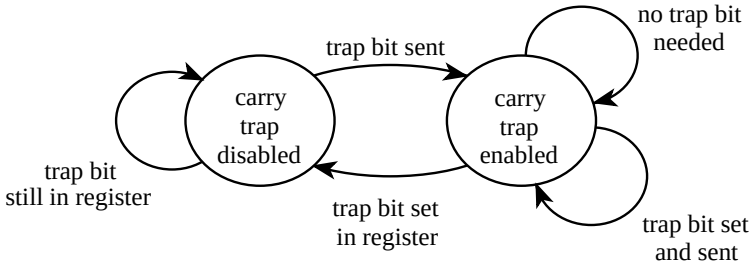


Figure 3.19: State transition diagram illustrating the carry trapping mechanism at the encoder. The state on the right is the initial state and corresponds to normal operation, where carry-trapping is enabled. After a carry-trap bit has been inserted, additional carry-trapping is disabled until the carry-trap bit has been shifted out of the carry-control field of the code register and sent to the channel.

partially shifted into the channel, and a carry-over changes the zero that delimits the string into a one. In this case, that carry-over is not *into* the string, so there is no chance of a carry-over problem. However, there is no way for the decoder to determine that the string of ξ consecutive was caused in this way; therefore, it must always treat the bit that follows the string as a carry-trap bit, and accordingly the encoder must always insert such a bit after every string of ρ consecutive ones, even if the string does not lie completely within the register.

A final complication arises in the following case. Suppose a carry-trap bit has been set, and is eventually converted to a one by a carry-over. Clearly, this one must not be counted towards the ξ consecutive ones required to trigger another carry-trap. Thus, carry-trapping must be disabled as long as the carry-control field contains a carry-trap bit. This consideration leads to the two-state carry-trapping mechanism as indicated in Figure 3.19.

This method of controlling the propagation of carry-overs was devised by Langdon and Rissanen [47], and may have been based on a similar technique devised by Rubin [60].

3.6 Performance of Arithmetic Coding Applied to Quantization

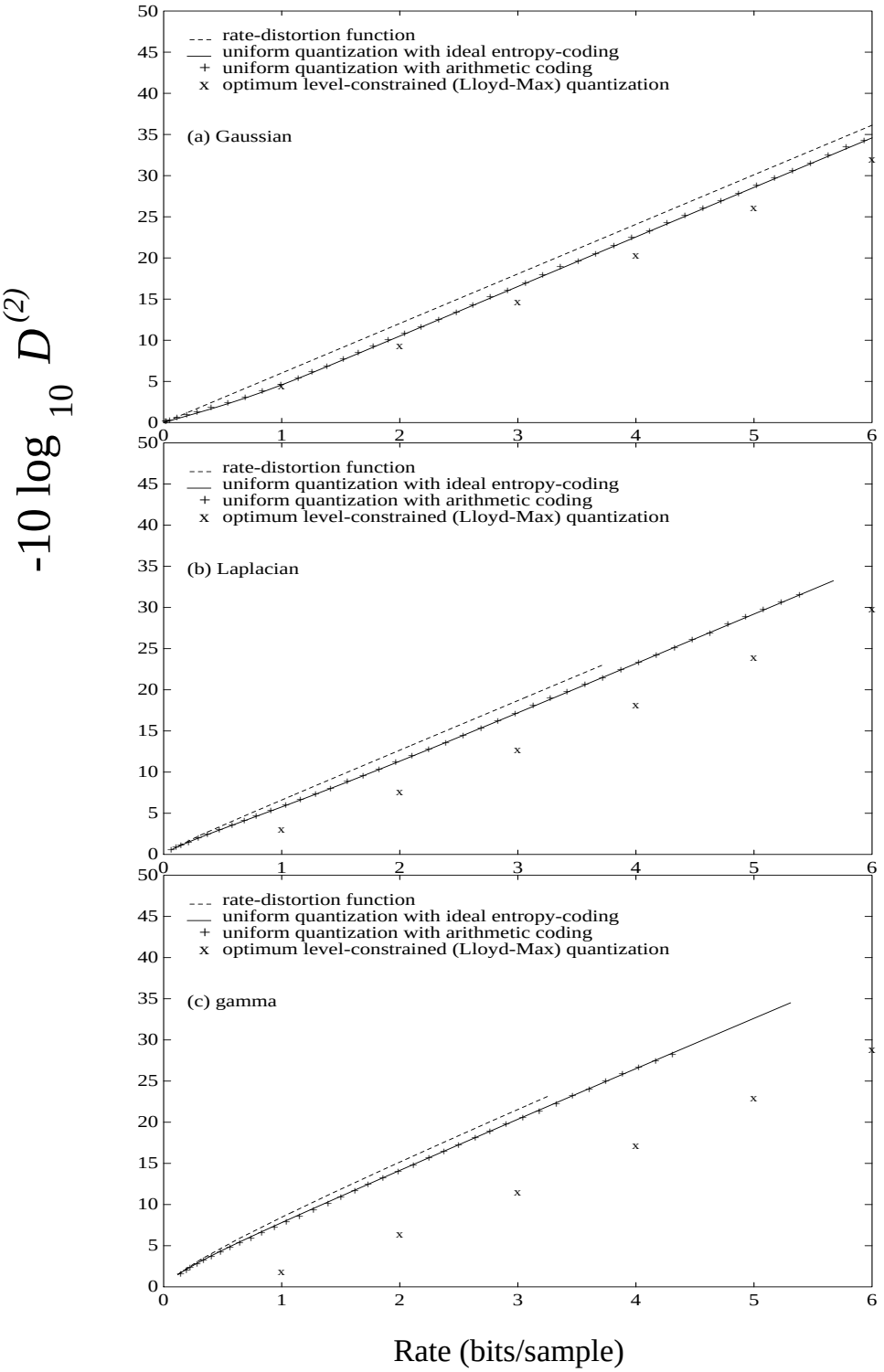


Figure 3.20: MSE-performance (rate versus mean-square error) of arithmetic-coded uniform quantization with 127 quantization regions, relative to the rate-distortion bound, uniform quantization with ideal entropy-coding and an unlimited number of quantization regions, and Lloyd-Max quantization, for (a) a Gaussian, (b) a Laplacian, and (c) a gamma memoryless source. To obtain each '+' point, the performance of arithmetic-coded quantization was measured by applying the software of Appendix C to a single block of 5,000 simulated quantized source samples. Here, the precision ρ of the arithmetic code is 16 bits, and the width ξ of the carry-control field is 12 bits.

Figure 3.20 shows the measured performance (in dB signal to MSE ratio) of a uniform quantizer with arithmetic coding, for several sources of interest. Note that the efficiency is nearly perfect over the range of rates considered.

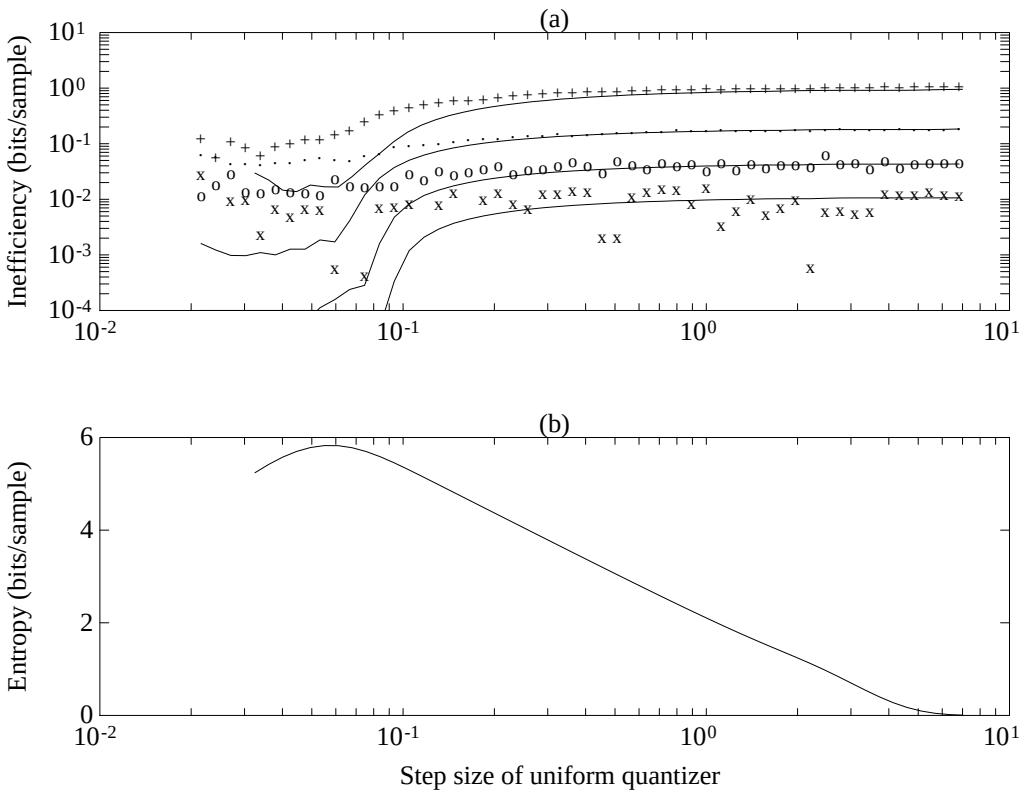


Figure 3.21: Inefficiency versus quantizer step size for arithmetic-coded uniform quantization of a Gaussian memoryless source. In (a) the measured inefficiency γ_N is shown, where $N = 50,000$, for many different step sizes, and for four different values of ρ: 8 (+), 10 ($\cdot$), 12 (o), and 14 (x). In all cases, the number of quantization regions is 127. To obtain these points the software presented in Appendix C was applied to a simulated quantized source. The solid lines in (a) indicate the component of inefficiency due to the rounding-off of probabilities to admissible values. For reference, the entropy is graphed against step size in (b). It can be seen that over the useful range of the quantizer, the rounding-off of probabilities accounts for nearly all of the inefficiency; the truncation in (3.29) appears to be significant only in the saturation region.

Performance can be characterized independently of distortion measure by expressing the inefficiency purely in terms of rate. This is accomplished in Figure 3.21(a), where the measured inefficiency (in bits-per-sample) is shown as a function of the

stepsize of a uniform quantizer for a Gaussian source. Note that when ρ is larger than roughly 14, the inefficiency is negligible (independent of which distortion exponent is used).

3.7 Alternative Coding Procedures

For many, the technique that comes to mind when the term *entropy coding* is mentioned is Huffman coding [14]. In Huffman coding, each source letter is represented by a distinct binary codeword. The codewords are of variable length; more probable source letters are assigned shorter codewords, while less probable letters are assigned longer codewords. It is easy to see that direct Huffman coding can never result in an average bit-rate of less than one bit per letter, since the shortest possible codeword has one bit. The present application requires that the average bit-rate occasionally be less than one bit per letter; therefore, direct Huffman coding is inappropriate.

Farvardin and Modestino [11] suggest an entropy-coding scheme that involves blocking the source letters into L-tuples, then applying a Huffman code to the resulting sequence of L-tuples. This blocking results in a new source alphabet that consists of K^L letters. To strike a compromise between performance at low bit-rates and codebook size, Farvardin and Modestino restrict L to be always equal to 2. This approach, although simple from both a conceptual and algorithmic point of view, suffers from two obvious limitations. First, the minimum bit-rate that can be achieved is $1/2$ (or more generally, $1/L$) bits per sample, and in the present application it is occasionally desired that the bit-rate be very close to zero. Second, as K becomes large (as is required at higher rates), the codebook becomes very large (perhaps prohibitively large) even when $L = 2$. These two considerations indicate that the scheme described in [11] is less appropriate in the present application than arithmetic coding.

Gallager [55] has suggested that the simplest and most natural form of entropy coding to use in the quantization application is a variant of run-length coding. It is

appropriate to question, then, why in this thesis arithmetic coding was chosen over this more intuitive run-length approach. Gallager argues that at very low bit-rates, run-length coding has nearly optimum performance in the sense that the average number of bits used per letter is nearly equal to the per-letter entropy. However, in his analysis, the number of quantization regions is limited to three ($K = 3$). To impose this limitation is appropriate in the case of very low bit-rates (the case on which Gallager focuses), but leads to an unacceptable amount of overload distortion at higher bit-rates (above about 1.6 bits per sample in the case of a Gaussian source and mean-square error). Thus, at higher bit-rates, a greater number of quantization regions is required. However, when the number of quantization regions is large, run-length coding becomes complicated, since the codebook must include runs of the most probable letter terminated by every other letter in the source alphabet. This can lead to a very large codebook in the case of a large number of quantization regions (e.g., $K = 100$), making pure run-length coding less attractive than arithmetic coding in the present application.

The simplicity and efficiency of run-length coding at low bit-rates and the difficulties associated with it at higher bit-rates suggest that a hybrid scheme be considered, in which run-length coding is used at lower bit-rates, and another technique, possibly K-ary arithmetic coding, is used at higher bit-rates. Such a hybrid scheme might offer some implementation advantages over pure K-ary arithmetic coding, like reduced computational complexity. This could be, in the author's opinion, a very workable approach.

As was mentioned at the beginning of the present chapter, binary arithmetic codes (arithmetic codes designed specifically for a source having a two-letter alphabet) are more widely used, and perhaps more widely understood, than K-ary arithmetic codes. Typical of such codes is the Q-coder recently described by Pennebaker and Mitchell [31][32]. The popularity of this type of arithmetic code may be explained in part by an implementation-related advantage offered by binary arithmetic codes over K-ary

arithmetic codes: in the binary case, it is a simple matter to replace multiplications in the recursion formulas with other (possibly less computationally expensive) operations without a incurring a significant reduction in efficiency [47][48]. Note that it is always possible to convert a sequence of letters selected from an alphabet of K letters (with $K > 2$) into an intermediate sequence of binary digits using some very simple, perhaps inefficient, fixed-length code. The resulting sequence of binary digits can then be encoded using a binary arithmetic code. This was, in fact, the approach to entropy coding first considered for this application by the author. The main problem with it is that the letter probabilities for the binary sequence must either be computed "on-the-fly" (via Bayes' rule) from the probabilities for the original K-ary source, or else they must be computed once and stored in a table. In the former case, the additional burden of having to compute the probabilities would far outweigh any computational simplification that might result from employing a binary rather than K-ary arithmetic code. In the latter case, the number of probabilities (or, if the approach described in [47] is used, the number of "skew numbers") that would have to be stored is increased by a factor of $\lceil \log_2 K \rceil$. Another problem with the approach is that if a general-purpose microprocessor or DSP chip is to be used instead of special-purpose hardware, then the bit manipulations involved in the translation of the source sequence to and from a binary sequence would significantly increase the number of instruction cycles required for encoding and decoding. For these reasons, direct K-ary arithmetic coding of the source sequence as described in this chapter is more appropriate than binary arithmetic coding in the present application.

The reader who familiar with the prior work in arithmetic coding may have observed that the K-ary arithmetic code presented in this chapter is just one of many possible arithmetic codes. Although the code described in this chapter is both efficient and easily realizable, it may not be the best choice in situations where multiplication is much more computationally expensive than other operations. Recall from Section 3.4 that encoding requires two fixed-precision multiplications per source letter, and that

decoding requires on the order of $\log_2 K$ fixed-precision multiplications per source letter (under the assumption that a binary search is used). Rissanen and Mohiuddin [61] have recently shown that it is possible to substitute shifting operations for multiplications in a K-ary arithmetic code while retaining good efficiency, at least for some sources.[20] This multiplication-free approach would be appropriate when extremely low-cost hardware is to be used (e.g., when general-purpose microprocessors are to be used for encoding and decoding). However, if a modern DSP chip (like ATT's DSP16A or Motorola's 56000) is available, then multiplications are no more expensive than other operations, and the code described in this chapter would be more appropriate because of its near optimality in terms of data compression, and because it appears to require two fewer comparison operations per encoding and decoding cycle than the multiplication-free approach described in [61].

The foregoing list of alternative coding techniques is far from exhaustive, but is thought to be representative. Consideration of the alternatives leads the author to believe that the K-ary arithmetic code described in the present chapter is exceptionally well-suited to the quantization problem.

3.8 Summary and Conclusions

The material presented in this chapter demonstrates that in practice, it is not difficult to encode quantizer output levels with nearly perfect efficiency over a relatively broad range of bit-rates, for the Gaussian, Laplacian, and gamma sources of interest. This fact had not been previously established to the present degree of generality.[21] The finding is important for two reasons. First, it essentially solves the source coding problem for entropy-coded quantization (at least in the case of reliable channels and moderately fast hardware). Second, it allows researchers involved in designing a new

[20] Their approach is analogous to that used in by Langdon and Rissanen in [47] and by Pennebaker and Mitchell in [32], but the extension to the K-ary case as described in [61] is, in the present author's opinion, not at all obvious. Rissanen and Mohiuddin are to be congratulated for their insight.

[21] As was observed in Section 3.7, certain special cases have been successfully addressed in prior work.

system to reasonably assume the availability of ideal entropy coding in their initial calculations and simulations, simplifying and speeding their work.

A significant part of this chapter was concerned with establishing the principles of arithmetic coding from a very basic level. The author's motivation behind this came from two observations. First, although arithmetic coding and its advantages are well understood by information theorists, arithmetic coding has not yet gained widespread acceptance among engineers, perhaps because previous detailed treatments of the subject have been targeted towards information theorists. Second, to the present author's knowledge, a detailed description of a K-ary source appropriate to the quantization problem is not available in any of the existing literature; the material presented in this chapter fills that void.

CHAPTER 4:

Conclusions

The main result of the present study is that scalar quantization with arithmetic coding is a very efficient and practical means of encoding the memoryless sources considered.

4.1 Summary of Work

The problem of scalar quantization with entropy-coding has been considered in detail for three commonly used source model pdf's — Gaussian, Laplacian, and gamma — in the low rate region. The two constituent subproblems — quantization and entropy-coding — have been considered separately, and the separation has been justified.

Chapter 2 considered the problem of scalar quantization, under the assumption that ideal entropy-coding is available. For Gaussian, Laplacian, and gamma memoryless sources and for difference distortion measures, it has been shown that the performance of entropy-constrained quantization is extremely good relative to alphabet-constrained quantization at all rates. Moreover, it has been shown that as the distortion measure exponent increases, the performance of entropy-coded quantization appears to approach the best possible performance of any source coding scheme, even at low rates of encoding (the result was known previously for high rates).

Further, it has been shown that contrary to a widely-held belief, uniform entropy-constrained quantization is not exactly optimal in the case of a Laplacian source and mean-square error criterion.

One of the more surprising findings of the present work is the asymmetry of optimum entropy-constrained quantizers in the case of a Gaussian source and mean-absolute distortion measures, for rates less than one bit per sample. Asymmetry in the alphabet-constrained case has been reported previously by Sharma [37] and Kabal [42], and in the entropy-constrained case, but with a severe restriction on the number of reconstruction levels, by Farvardin [26]. In the latter work, asymmetry

is also reported in some cases when the number of reconstruction levels is large, but it is asserted there that in every such case an optimal symmetric quantizer can also be found. The asymmetry found in the present study is remarkable for two reasons: first, it does not derive from a restriction on the number of levels, and second, the performance of the asymmetric quantizers is *significantly* (e.g., 0.3 dB) better than the corresponding locally-optimal symmetric quantizers for at least some rates.

In Chapter 3, the assumption of the availability of ideal entropy-coding was justified by the description of an efficient arithmetic code. The code was derived from basic principles, and was shown to perform with near-perfect efficiency in the scalar quantization problem over a useful range of entropy for the three sources considered, independent of the value of the distortion exponent.

4.2 Suggestions for Further Research

This work can be extended in several ways. The most straightforward would be to investigate some of the important practical problems which were neglected here. These include buffering to allow fixed-rate transmission, making the system perform reasonably well in the presence of channel errors, and implementing the arithmetic code in hardware efficiently.

While the random search technique described in Chapter 2 provides evidence about the character of optimal quantizers, the evidence is not, by its nature, conclusive. An alternative method of estimating the best performance attainable by entropy-constrained quantization is suggested by Pearlman [62]. The basic idea is to compute the rate-distortion performance indirectly, by calculating the rate-distortion function of the source with an appropriate constraint on reconstruction levels (or, in Pearlman's terms, *reproduction alphabet*). This approach requires an efficient means of finding the optimal reproducing letters at each step of the algorithm. It is claimed in [62] that by modifying the distortion measure at each iteration, the performance of the optimal entropy-constrained quantizer can be found for a given number of re-

construction levels. By allowing the number of levels to become large, the computed performance will closely approximate that of the optimal entropy-constrained quantizer (without restriction on the number of levels). Though this technique will not reveal the decision levels of an optimal quantizer, it will still reveal the performance and therefore ought to be investigated. Knowledge of the best attainable performance would be valuable in researching the problem of entropy-coded quantization, as it would be a tighter bound than the rate-distortion function.

The apparent asymmetry of the optimal quantizer for a Gaussian source with mean-absolute distortion measure at low rates discovered in the present study is contrary to intuition, and more work seems to be required in this area before the phenomenon can be properly understood. Also, the apparent "symmetry breakpoint" at approximately 1.3 bits/sample for a Gaussian source with higher-order distortion measures is unexpected. In this context, the present work can be extended in at least two ways. The first is by simply performing the same random search experiments for the source-exponent combinations not tried in the present study, perhaps using a greater number of samples and a greater number of entropy bins. The second is by demonstrating, analytically, that globally optimal quantizers indeed have the asymmetric decision levels obtained in the present study, and are in fact unique (allowing for sign change and multiple decision levels). A study of the latter type will be hampered by two obstacles: first, the existence of symmetric locally-optimal quantizers for all three sources, for all rates, and for all error exponents considered (as demonstrated herein); and second, the insensitivity of both entropy and error to infinitesimal-probability regions (i.e., regions bounded by infinitesimally spaced or multiple decision levels).

In conclusion, it should be emphasized that the concern of this thesis has been with the efficient quantization of a probabilistic model of a physical source of information, not of the source itself. However disconcerting this abstraction might be, some deliberation will reveal that there is no reasonable alternative. The quantizer

— or, more generally, the encoder — *must* be designed according to what the designer believes the source will be, and this requires modeling (whether that modeling be implicit or explicit). Ultimately, the degree to which the results of this thesis pertain to real-world applications is the degree to which the assumed models successfully characterize the important aspects of the physical sources of information in those applications.

Appendix A: A Simple Optimization Procedure

This appendix presents a simple *ad hoc* procedure for finding a local minimum of a smooth, non-linear, scalar-valued function of a vector.

Let $\langle \mathbf{u}, \mathbf{v} \rangle$ denote the Euclidean inner (dot) product of the n-dimensional vectors $\mathbf{u}$ and $\mathbf{v}$. Let $\|\mathbf{u}\|$ denote the Euclidean norm of an n-dimensional vector $\mathbf{u}$. Let $\{\mathbf{x}_k\}$, $k = 0, 1, \ldots$, be a sequence of n-dimensional vectors, let x_{ki} denote the i^{th} component of $\mathbf{x}_k$, and let $F(\mathbf{x}_k)$ be a scalar-valued function of $\mathbf{x}_k$ for which the partial derivatives $\partial F(\mathbf{x}_k)/\partial x_{ki}, i = 1, \ldots, n$, are continuous and available in analytic form. The gradient of $F(\mathbf{x}_k)$, denoted $\nabla F(\mathbf{x}_k)$, is an n-dimensional vector function of $\mathbf{x}_k$ having as its i^{th} component $\partial F(\mathbf{x}_k)/\partial x_{ki}$. Let $\mathbf{g}_k$ be an abbreviation for $\nabla F(\mathbf{x}_k)/\|\nabla F(\mathbf{x}_k)\|$, the normalized gradient evaluated at $\mathbf{x}_k$. Let $\mathbf{x}^*$ be the local minimum of $F(\mathbf{x})$ which is sought, and let $\mathbf{x}_0$ be a starting point known to be in the vicinity of $\mathbf{x}^*$.

An ad-hoc recursive procedure to generate a sequence $\mathbf{x}_k, k = 1, \ldots$, that converges to $\mathbf{x}^*$, is given by

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha_k \, \mathbf{g}_{k-1}, \tag{A.1}$$

where the adaptive step size α_k is computed as

$$\alpha_k = \begin{cases} 0.1 \, \|\mathbf{x}_0\| & \text{if } k = 1; \\ \alpha_{k-1} \left[0.8 + 0.5 \langle \mathbf{g}_k, \mathbf{g}_{k-1} \rangle \right] & \text{otherwise.} \end{cases} \tag{A.2}$$

This procedure is similar to the *method of steepest descent* [40, pp. 214-225], but differs in one important respect: instead of performing a line minimization at every step, the new point is obtained by simply stepping in the direction of the negative gradient by a small amount α_k, where the amount is determined by the recent consistency of the direction of the search (as measured by the dot product of the normalized gradient with the previous normalized gradient). The philosophy behind this strategy is simple. If the direction of the search has been consistent, as would be the case far from a local optimum, then the search is accelerated. On the other hand, if it has been inconsistent, as would be the case in the vicinity of a local minimum, then the search is decelerated.

The constants 0.8 and 0.5 in $(A.2)$ were empirically determined to be suitable in a large number of test cases. These constants determine the maximum and minimium possible step-size multipliers, which here are 1.3 and 0.3, respectively.

While the convergence properties of this optimization procedure have not been explored in detail, it is conjectured that the properties are similar to the method of steepest descent in that the efficiency of the search is markedly reduced in the vicinity of a local minimum. Still, the procedure is easy to implement, and has proven itself to be extremely reliable in a variety of problems, including the design of level-constrained quantizers (see Chapter 2) and the design of filters and filter banks [63][64]. In each of these applications, the procedure presented here was found to converge more reliably and occasionally more quickly than the *TOMS* algorithm #500 [41] and the Fletcher-Reeves-Polak-Ribiere algorithm [65, pp. 317-323], although this might be attributable to an inappropriate choice of parameters in the latter algorithms.

Appendix B: Software to calculate moments of densities

This appendix presents C-language software to compute the definite integral

$$\int_a^b X^i f_x(X) dX,$$

where $i = 1, \ldots, 6$, and $f_x(X)$ is a zero-mean unit-variance Gaussian, Laplacian, or gamma density. These routines were developed to facilitate computation of some of the numerical results presented in Chapter 2.

The routines presented in this appendix have been tested on a Digital Equipment Corporation VAX 785 computer running the Berkeley Unix (BSD 4.3) operating system, and also on a Cray Research model Cray-2 supercomputer running the UNICOS operating system. The routines should be linked with the standard math library as well as with functions that evaluate the probability densities and perform Romberg integration (see the comments in the software listing below for a more detailed description of what is required).

```
/* MOMENT.C:  Computes the definite integral of p(x)*x^i, from a to b,
*  where p(x) is a Gaussian, Laplacian, gamma, or Rayleigh pdf.
*  Written by Ashok Popat, 12/88.
*
*  usage:
*
*      extern double moment();
*      double val;
*
*      val = moment(pdfchar,numer,i,a,b);
*      int pdfchar, numer, i;
*      double a,b;
*
*  The pdf is determined by pdfchar --- 'l' for Laplacian, 'n' for Gaussian,
*  'g' for gamma, and 'r' for Rayleigh.  WARNING: On non-Vax machines,
*  pdfchar should be changed to be of type "char".
*  If numer = 1, the Gaussian integration is performed; if numer = 2, then
*  Romberg integration is performed; otherwise, the integration is performed
*  analytically.  The exponent in the moment is given by i, and the limits of
*  integration are given by a and b.  The moment value is returned in val.
*
*  requirements:
*
*  The following external routines are required:
*  qromb() --- Romberg integration routine --- should take same arguments
*          as qgauss() provided here.  The routine given by Press et al
*          does nicely.
*  p_lapla(), p_gauss(), p_gamma() p_rayle() --- routines that take single
*          argument and return the pdf value at that point for Laplacian,
*          Gaussian, gamma, and Rayleigh distributions, respectively.
*/

#include <math.h>
#include <stdio.h>
#include "formulae.h"    /* declarations of p_lapla(), p_gauss(), etc. */

/* some useful constants, obtained by using MACSYMA in numerical mode: */

#define S 1.526399745532848235622388155084
#define SS 2.329896183162743847239353634628534 81
#define THREEP75    2.27950705695478
#define THREEP25    1.31607401295249
#define S3    3.55635294109757
#define S4    5.42841622431632
#define S5    8.28593314344282
#define S6   12.64764624165331
#define INVSS 0.42920367320510338076867830836024855 79
#define HALFINVSS 0.21460183660255169038433915418012427895
#define PI 3.141592653589793238
#define INVPI 1.0/3.141592653589793238
#define SQRT3 1.73205080756887729352
#define THSQRT3 3*1.73205080756887729352
#define SQRT2 1.41421356237309504880
#define SQRTPI 1.7724538509055160
#define FROT 1.31607401295249246081
#define SQ3O2 0.86602540378443864676
#define TWOSQRTPI 3.54490770181103205459633496
#define SQRTTWOPI 2.50662827463100050241277209
#define TWOPISQRT3 10.88279618540530710351768279
#define POINT5 0.5
#define SQRT2S SQRT2*S
#define SQRT2SQRTPI SQRT2*SQRTPI

extern double qromb();
static double (*p_pdf)();

static double exponent;

double s_d(x)
    double x;
{
 double y,xp;
 xp = pow(x,exponent);
 y = xp*(*p_pdf)(x);
 return(y);
}

/* The following subroutine to perform Gaussian quadrature was taken from
*  Press, et al., Numerical Recipes in C, Cambridge Univ. Press, NY., 1988.
*/
double qgaus(func,a,b)
double a,b;
double (*func)();      /* ANSI: double (*func)(double); */
{
      int j;
      double xr,xm,dx,s;
      static double x[]={0.0,0.1488743389,0.4333953941,
            0.6794095682,0.8650633666,0.97390652};
      static double w[]={0.0,0.2955242247,0.2692667193,
            0.2190863625,0.1494513491,0.06667134};
```

```
        xm=0.5*(b+a);
        xr=0.5*(b-a);
        s=0;
        for (j=1;j<=5;j++) {
                dx=xr*x[j];
                s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
        }
        return s *= xr;
}

extern double gauss_moment(i,a,b)
     double a,b;
     int i;
{
  static double sa,sb,tmp,a2,a3,a4,a5,a6,b2,b3,b4,b5,b6;

  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  switch (i) {
  case 0:
    sa = erf(a/SQRT2)*POINT5;
    sb = erf(b/SQRT2)*POINT5;
    break;
  case 1:
    sa = -exp(-a*a*POINT5)/(SQRTTWOPI);
    sb = -exp(-b*b*POINT5)/(SQRTTWOPI);
    break;
  case 2:
    sa = erf(a/SQRT2)*POINT5-a*exp(-a*a*POINT5)/(SQRTTWOPI);
    sb = erf(b/SQRT2)*POINT5-b*exp(-b*b*POINT5)/(SQRTTWOPI);
    break;
  case 3:
    a2 = a*a;
    b2 = b*b;
    sa = -(SQRT2*a2+2.0*SQRT2)*exp(-a2*0.5)/SQRTPI*0.5;
    sb = -(SQRT2*b2+2.0*SQRT2)*exp(-b2*0.5)/SQRTPI*0.5;
    break;
  case 4:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    sa = -exp(-a2*0.5)*(SQRTPI*(SQRT2*a3+3.0*SQRT2*a)-3.0*PI*exp(a2*0.5)
   *erf(a/SQRT2))*INVPI*0.5;
    sb = -exp(-b2*0.5)*(SQRTPI*(SQRT2*b3+3.0*SQRT2*b)-3.0*PI*exp(b2*0.5)
   *erf(b/SQRT2))*INVPI*0.5;
    break;
  case 5:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    sa = -(a4+4.0*a2+8.0)*exp(-1.0*a2*0.5)/(SQRT2SQRTPI);
    sb = -(b4+4.0*b2+8.0)*exp(-1.0*b2*0.5)/(SQRT2SQRTPI);
    break;
  case 6:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    sa = exp(-a2*0.5)*(15.0*SQRT2SQRTPI*exp(a2*0.5)*erf(a/SQRT2)
        -2.0*a5-10.0*a3-30.0*a)/(SQRT2SQRTPI)*0.5;
    sb = exp(-b2*0.5)*(15.0*SQRT2SQRTPI*exp(b2*0.5)*erf(b/SQRT2)
        -2.0*b5-10.0*b3-30.0*b)/(SQRT2SQRTPI)*0.5;
    break;
  default:
    fprintf(stderr,"order out of range in gauss_moment\n");
    exit(-1);
  }
  return(sb-sa);
}

#define odd(n) (n%2)

static double p_l_m(i,a,b)
     double a,b;
     int i;
{
  static double sa,sb,tmp,a2,a3,a4,a5,a6,b2,b3,b4,b5,b6;

  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  switch (i) {
  case 0:
    sa = -exp(-SQRT2*a)*POINT5;
    sb = -exp(-SQRT2*b)*POINT5;
    break;
  case 1:
    sa = -(2.0*a+SQRT2)*exp(-SQRT2*a)/4.0;
    sb = -(2.0*b+SQRT2)*exp(-SQRT2*b)/4.0;
    break;
  case 2:
    sa = -(SQRT2*a*a+2.0*a+SQRT2)*exp(-SQRT2*a)/SQRT2*POINT5;
    sb =  -(SQRT2*b*b+2.0*b+SQRT2)*exp(-SQRT2*b)/SQRT2*POINT5;
    break;
  case 3:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    sa = -(2.0*a3+3.0*SQRT2*a2+6.0*a+3.0*SQRT2)*exp(-SQRT2*a)*0.25;
    sb = -(2.0*b3+3.0*SQRT2*b2+6.0*b+3.0*SQRT2)*exp(-SQRT2*b)*0.25;
    break;
  case 4:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    sa = -(a4+2.0*SQRT2*a3+6.0*a2+6.0*SQRT2*a+6.0)*exp(-SQRT2*a)*0.5;
    sb = -(b4+2.0*SQRT2*b3+6.0*b2+6.0*SQRT2*b+6.0)*exp(-SQRT2*b)*0.5;
    break;
  case 5:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    sa = -(2.0*a5+5.0*SQRT2*a4+20.0*a3+30.0*SQRT2*a2+60.0*a+30.0*SQRT2)
      *exp(-SQRT2*a)*0.25;
    sb = -(2.0*b5+5.0*SQRT2*b4+20.0*b3+30.0*SQRT2*b2+60.0*b+30.0*SQRT2)
      *exp(-SQRT2*b)*0.25;
    break;
  case 6:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    a6 = a5*a;
    b6 = b5*b;
    sa = -(SQRT2*a6+6.0*a5+15.0*SQRT2*a4+60.0*a3+90.0*SQRT2*a2+180.0*a+90.0
    *SQRT2)*exp(-SQRT2*a)/SQRT2*0.5;
    sb = -(SQRT2*b6+6.0*b5+15.0*SQRT2*b4+60.0*b3+90.0*SQRT2*b2+180.0*b+90.0
    *SQRT2)*exp(-SQRT2*b)/SQRT2*0.5;
    break;
  default:
    fprintf(stderr,"order out of range in lapla_moment\n");
    exit(-1);
  }
  return(sb-sa);
}

extern double lapla_moment(i,a,b)
     double a,b;
     int i;
{
  double tmp;
  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  if (odd(i)) {
    if (a>0.0) return(p_l_m(i,a,b));
    else if (b>0.0) return(p_l_m(i,0.0,b) - p_l_m(i,0.0,-a));
```

```
    else return(-1.0*p_l_m(i,-b, -a));
    }
  else {
    if (a>0.0) return(p_l_m(i,a,b));
    else if (b>0.0) return(p_l_m(i,0.0,b) + p_l_m(i,0.0,-a));
    else return(p_l_m(i,-b, -a));
  }
}

static double p_g_m(i,a,b)
     double a,b;
     int i;
{
  static double sa,sb,tmp,a2,a3,a4,a5,a6,b2,b3,b4,b5,b6,sqrta,sqrtb;

  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  switch (i) {
  case 0:
    sa = erf(sqrt(SQRT3*a)/SQRT2)*POINT5;
    sb = erf(sqrt(SQRT3*b)/SQRT2)*POINT5;
    break;
  case 1:
    sa = erf(sqrt(a*SQ3O2))/SQRT3*POINT5 -
      sqrt(a/(TWOPISQRT3))*exp(-SQ3O2*a);
    sb = erf(sqrt(b*SQ3O2))/SQRT3*POINT5 -
      sqrt(b/(TWOPISQRT3))*exp(-SQ3O2*b);
    break;
  case 2:
    sa = -1/(THREEP25)*exp(-SQRT3*0.5*a)*
      (sqrt(PI)*sqrt(a)*(SQRT2*a+SQRT2*SQRT3)-(THREEP25)*PI*
       erf((THREEP25)*sqrt(a)/SQRT2)*exp(SQRT3*0.5*a))*INVPI*0.5;
    sb = -1/(THREEP25)*exp(-SQRT3*0.5*b)*
      (sqrt(PI)*sqrt(b)*(SQRT2*b+SQRT2*SQRT3)-(THREEP25)*PI*
       erf((THREEP25)*sqrt(b)/SQRT2)*exp(SQRT3*0.5*b))*INVPI*0.5;
    break;
  case 3:
    sqrta = sqrt(a);
    sqrtb = sqrt(b);
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    sa = -exp(-SQRT3*a*0.5)*
      (sqrta*(6.0*THREEP25*a2+10.0*THREEP75*a+30.0
       *THREEP25)-15.0*SQRT2SQRTPI*
       erf(THREEP25*sqrta/SQRT2)*
       exp(SQRT3*a*0.5))/(SQRT2*SQRT3*SQRTPI)/6.0;
    sb = -exp(-SQRT3*b*0.5)*
      (sqrtb*(6.0*THREEP25*b2+10.0*THREEP75*b+30.0
       *THREEP25)-15.0*SQRT2SQRTPI*
       erf(THREEP25*sqrtb/SQRT2)*
       exp(SQRT3*b*0.5))/(SQRT2*SQRT3*SQRTPI)/6.0;
    break;
  case 4:
    sqrta = sqrt(a);
    sqrtb = sqrt(b);
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    sa = -exp(-SQRT3*a*0.5)*(SQRTPI*sqrta*
  (3*SQRT2*THREEP75*a3+21*SQRT2*THREEP25*a2+
  35*SQRT2*THREEP75*a+105*SQRT2*THREEP25)-
  105*PI*erf(THREEP25*sqrta/SQRT2)*exp(SQRT3*a*0.5))*INVPI/18.0;
    sb = -exp(-SQRT3*b*0.5)*(SQRTPI*sqrtb*
  (3*SQRT2*THREEP75*b3+21*SQRT2*THREEP25*b2+
  35*SQRT2*THREEP75*b+105*SQRT2*THREEP25)-
  105*PI*erf(THREEP25*sqrtb/SQRT2)*exp(SQRT3*b*0.5))*INVPI/18.0;
    break;
  case 5:
    sqrta = sqrt(a);
    sqrtb = sqrt(b);
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    sa = -(1.0/THREEP75)*exp(-SQRT3*a*0.5)*
  (sqrta*(2*SQRT3*a4+18*a3+42*SQRT3*a2+210*a+
  210*SQRT3)-105*SQRT2*THREEP25*SQRTPI*
  erf(THREEP25*sqrta/SQRT2)*exp(SQRT3*a*0.5))/
  (SQRT2SQRTPI)*0.5;
    sb = -(1.0/THREEP75)*exp(-SQRT3*b*0.5)*
  (sqrtb*(2*SQRT3*b4+18*b3+42*SQRT3*b2+210*b+
  210*SQRT3)-105*SQRT2*THREEP25*SQRTPI*
  erf(THREEP25*sqrtb/SQRT2)*exp(SQRT3*b*0.5))/
  (SQRT2SQRTPI)*0.5;
    break;
  case 6:
    sqrta = sqrt(a);
    sqrtb = sqrt(b);
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    a6 = a5*a;
    b6 = b5*b;
    sa = -(1.0/THREEP25)*exp(-SQRT3*a*0.5)*
  (SQRTPI*sqrta*(3*SQRT2*a5+11*SQRT2*SQRT3*a4+99*SQRT2*a3 +
  231*SQRT2*SQRT3*a2+1155*SQRT2*a+1155*SQRT2*SQRT3)-
  1155*THREEP25*PI*erf(THREEP25*sqrta/SQRT2)*
  exp(SQRT3*a*0.5))*INVPI/6.0;
    sb = -(1.0/THREEP25)*exp(-SQRT3*b*0.5)*
  (SQRTPI*sqrtb*(3*SQRT2*b5+11*SQRT2*SQRT3*b4+99*SQRT2*b3 +
  231*SQRT2*SQRT3*b2+1155*SQRT2*b+1155*SQRT2*SQRT3)-
  1155*THREEP25*PI*erf(THREEP25*sqrtb/SQRT2)*
  exp(SQRT3*b*0.5))*INVPI/6.0;
    break;
  default:
    fprintf(stderr,"order out of range in gamma_moment\n");
    exit(-1);
  }
  return(sb-sa);
}

extern double gamma_moment(i,a,b)
     double a,b;
     int i;
{
  double tmp;
  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  if (odd(i)) {
    if (a>0.0) return(p_g_m(i,a,b));
    else if (b>0.0) return(p_g_m(i,0.0,b) - p_g_m(i,0.0,-a));
    else return(-1.0*p_g_m(i,-b, -a));
    }
  else {
    if (a>0.0) return(p_g_m(i,a,b));
    else if (b>0.0) return(p_g_m(i,0.0,b) + p_g_m(i,0.0,-a));
    else return(p_g_m(i,-b, -a));
  }
}


extern double rayle_moment(i,a,b)
     double a,b;
     int i;
{
  static double sa,sb,tmp,a2,a3,a4,a5,a6,b2,b3,b4,b5,b6;

  if (a>b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  if (a < 0.0) a = 0.0;
  switch (i) {
  case 0:
    sa =  -exp(-(a*a)*HALFINVSS);
    sb =  -exp(-(b*b)*HALFINVSS);
    break;
  case 1:
    sa = SQRT2SQRTPI*S*erf(a/(SQRT2*S))*0.5-a*exp(-(a*a)*HALFINVSS);
    sb = SQRT2SQRTPI*S*erf(b/(SQRT2*S))*0.5-b*exp(-(b*b)*HALFINVSS);
    break;
  case 2:
    a2 = a*a;
    b2 = b*b;
    sa = -((a2)+2*(SS))*exp(-(a2)*HALFINVSS);
```

```
    sb = -((b2)+2*(SS))*exp(-(b2)*HALFINVSS);
    break;
  case 3:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    sa = exp(-a2/SS*0.5)*(3*SQRT2SQRTPI*S3*exp(a2/SS*0.5)*
    erf(a/(SQRT2*S))-2*a3-6*SS*a)*0.5;
    sb = exp(-b2/SS*0.5)*(3*SQRT2SQRTPI*S3*exp(b2/SS*0.5)*
    erf(b/(SQRT2*S))-2*b3-6*SS*b)*0.5;
    break;
  case 4:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    sa = -(a4+4*SS*a2+8*S4)*exp(-a2/SS*0.5);
    sb = -(b4+4*SS*b2+8*S4)*exp(-b2/SS*0.5);
    break;
  case 5:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    sa = exp(-a2/SS*0.5)*(15*SQRT2SQRTPI*S5*exp(a2/SS*0.5)*
 erf(a/(SQRT2*S))-2*a5-10*SS*a3-30*S4*a)*0.5;
    sb = exp(-b2/SS*0.5)*(15*SQRT2SQRTPI*S5*exp(b2/SS*0.5)*
 erf(b/(SQRT2*S))-2*b5-10*SS*b3-30*S4*b)*0.5;
    break;
  case 6:
    a2 = a*a;
    b2 = b*b;
    a3 = a2*a;
    b3 = b2*b;
    a4 = a3*a;
    b4 = b3*b;
    a5 = a4*a;
    b5 = b4*b;
    a6 = a5*a;
    b6 = b5*b;
    sa = -(a6+6*SS*a4+24*S4*a2+48*S6)*exp(-a2/SS*0.5);
    sb = -(b6+6*SS*b4+24*S4*b2+48*S6)*exp(-b2/SS*0.5);
    break;
  default:
    fprintf(stderr,"order out of range in rayle_moment\n");
    exit(-1);
  }
  return(sb-sa);
}

extern double moment(pdfchar,numer,i,a,b)
     int pdfchar,i,numer;
     double a,b;
{
  double val;
  exponent = (double) i;
  if (numer==1) {
    switch (pdfchar) {
    case 'n':
      p_pdf = p_gauss;
      break;
    case 'l':
      p_pdf = p_lapla;
      break;
    case 'g':
      p_pdf = p_gamma;
      break;
    case 'r':
      p_pdf = p_rayle;
      if (b < 0.0) return(0.0);
      if (a < 0.0) a = 0.0;
      break;
    default:
      fprintf(stderr,"unrecognized pdf\n");
      exit(-1);
    }
    val = qgaus(s_d,a,b);
  }
  else if (numer==2) {
    switch (pdfchar) {
    case 'n':
      p_pdf = p_gauss;
      break;
    case 'l':
      p_pdf = p_lapla;
      break;
    case 'g':
      p_pdf = p_gamma;
      break;
    case 'r':
      p_pdf = p_rayle;
      if (b < 0.0) return(0.0);
      if (a < 0.0) a = 0.0;
      break;
    default:
      fprintf(stderr,"unrecognized pdf\n");
      exit(-1);
    }
    val = qromb(s_d,a,b);
  }
  else {
    switch (pdfchar) {
    case 'n':
      val = gauss_moment(i,a,b);
      break;
    case 'l':
      val = lapla_moment(i,a,b);
      break;
    case 'g':
      val = gamma_moment(i,a,b);
      break;
    case 'r':
      if (b < 0.0) return(0.0);
      if (a < 0.0) a = 0.0;
      val = rayle_moment(i,a,b);
      break;
    default:
      fprintf(stderr,"unrecognized pdf\n");
      exit(-1);
    }
  }
  return(val);
}

#undef S
#undef SS
#undef THREEP75
#undef THREEP25
#undef S3
#undef S4
#undef S5
#undef S6
#undef INVSS
#undef HALFINVSS
#undef PI
#undef SQRT3
#undef THSQRT3
#undef SQRT2
#undef SQRTPI
#undef FROT
#undef SQ3O2
#undef TWOSQRTPI
#undef SQRTTWOPI
#undef TWOPISQRT3
#undef POINT5
```

Appendix C: Software to perform K-ary arithmetic coding

This appendix presents C-language procedures to perform arithmetic coding and decoding of a discrete stationary memoryless source. With some simple modifications, the procedures can accommodate arbitrary discrete sources. The listings presented here correspond to the contents of two files: (1) a file containing the procedures, entitled "ACMS.C," and (2) a file containing an example main program to demonstrate how the procedures can be used, entitled "ACMS_EXAMPLE.C." The encoding and decoding procedures listed here were used in investigating the performance of arithmetic coding applied to quantization (as discussed in Chapter 3).

The software presented in this appendix was developed and tested under the BSD (University of California, Berkeley) Unix version 4.3 operating system running on a Digital Equipment Corporation VAX 785 computer. The routines should be linked with the standard math library, and the example main program requires an external uniform random number generator — in the listing presented here, the "random()" function from the standard C library is used.

```
/* ACMS.C --- C routines to perform arithmetic coding and decoding
**          on a sequence of independent letters (memoryless source).
**
**      encode() --- encode a sequence of source letters
**      decode() --- decode a sequence of code bits into source letters
**      (for usage, see comments that precede declarations of the routines)
**
**  This code can be easily made to accommodate sources with memory
**  by modifying ENCODE and DECODE to accept a pointer to a procedure to
**  calculate the conditional letter probabilities, and having these routines
**  call that procedure and the rounding-off procedure OPTPRND
**  at every recursion.
**
**  Written by Ashok C. Popat, 3/89.  Based on algorithms described in
**  A.C. Popat, "Scalar Quantization with Arithmetic Coding,"
**  S.M. Thesis, MIT Dept. of Electr. Eng. and Computer
**  Science, June 1990.
**
*/

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

#define MAXK 128        /* max. allowable size of source alphabet */
#define MAXLEN 10000    /* max. allowable length of source sequence */
#define MAXCBITS 100000 /* max. allowable length of code bit sequence */
#define MAXPR 16        /* max. allowable precision */
#define MAXCCW 12       /* max. allowable width of carry-control field */

/* Encoder/decoder registers are implemented with ANSI C data type "unsigned
** long.  Define a new type, "codereg," for these registers: */
typedef unsigned long codereg;

/* variables: */
static int k;       /* size of source alphabet */
static int pr;      /* precision of arithmetic operations (e.g., 16) */
static int ccw;     /* width of carry-control field  (e.g., 8) */
static codereg twotothe[32];  /* array of masks that isolate individual bits */
static double *p;           /* source probabilities */
static codereg q[MAXK];     /* rounded source probabilities */
static codereg cumq[MAXK];  /* cumulative rounded source probabilities */
static int verbose = 0;     /* set to zero to make silent */

/* routines: */
static int optprnd(); /* routine to round probs to admissible vals */
static void printulong();    /* debugging utility */
static int codereglen();     /* returns length, in bits, of type codereg */
static int bad_pr_ccw();     /* checks for validity of pr and ccw */
static void inittwotothe();  /* sets twotothe[i] = 2^i */
static double entropy();     /* returns entropy of source (not necessary
   for encoding and decoding) */

/* Macro for testing individual bits:
** synopsis: bitvalue = bittst(register,bitnumber); */
#define bittst(r,n) (((r)&twotothe[n]) > 0 ? 1:0)

/* Macro for setting value of individual bits:
** synopsis: bitset(register,bitnumber,bitvalue); */
#define bitset(r,n,b) if (b>0) r |= twotothe[n]; else r &= ~twotothe[n]

/************************* Encode() ********************************
**  void encode (k, probs, letters, nletters, pr, ccw, codebits, nbits)
**  int k, pr, ccw, *letters, *codebits;
**  double *probs;
**  long nletters, *nbits;
**
**  The user supplies the size of the source alphabet ("k"),
**  an array of probabilities of the source letters ("probs"), an
**  array (sequence) of letters to encode ("letters"), the length of that
**  sequence ("nletters"), the precision of the code ("pr"),
**  and the width of the carry-control field in the code register ("ccw").
**  The routine returns an array (sequence) of codebits ("codebits") and the
**  number of codebits ("nbits").
**
**  The encoder has three registers: ex, ew, and ed.  The code is generated
**  in register ex.  That register has two contiguous fields: the arithmetic
**  field (the low-order pr bits) and the carry-control field (next ccw bits).
**
**  The arithmetic field is abbreviated "af"; the carry-control field, "ccf".
**  Register ew has only one field consisting of pr bits.
```

```
**
**   Register ed is a double-precision register, having one field consisting
**   of 2pr bits. However, currently, its contents are always truncated to
**   pr bits.  The use of double precision was originally to facilitate
**   experimentation with regards to the type of probability quantization
**   performed.
**
**   All registers are implemented using the same data type, codereg.
**
**   Processing steps (procedure encode):
**   Note:  trunc(a) denotes truncating a to pr bits.
**     1.  Set up masks, tables, etc.  Initialize ex = 0; ew = .1111111(binary)
**     2.  Get q[i] and cumq[i] from probs using procedure optprnd().
**     3.  For each source letter do...
**         3b.  ex = ex + trunc(ew*cumq[letter]).
**         3c.  ed = trunc(q[letter]*ew).
**         3d.  while most significant bit of ed is 0 do...
**              3da. emit most significant bit of carry-control field.
**              3db. shift ex left.
**              3dc. if necessary, insert a zero-bit to trap a carry.
**              3dd. mask out emitted bit of ex.
**              3de. shift ed left.
**              (end of "while")
**         3e.  ew = trunc(ed).
**         (end of "for each")
**     4.  Repeat pr+ccw times...
**         4a.  emit most significant bit of carry-control field.
**         4b.  shift ex left.
**         4c.  mask out emitted bit of ex.
**         (end of "repeat")
**     (end of "encode")
**
*/

/*****************************************************************/
/* Macro for counting the number of consecutive ones in the high end of
** the carry-control field.  Needed by "carryalert." Result is in "nco": */
#define countnco nco=0; while( bittst(ex,ccfmsb-nco) ) nco++

/*****************************************************************/
/* Macro for putting a bit in the code string:
** synopsis: sendbit(bitvalue); */

#define sendbit(b) codebits[(*nbits)++] = b;\
                                            \
              /* if 1 and if there is no carry trap currently in ex,\
              ** then update ncojt.  Otherwise, zero ncojt: */\
                                            \
              if ((b==1)&&(ctbptr>ccfmsb)) ncojt++;\
              else ncojt = 0

/*****************************************************************/
/* Macro for conditionally inserting a carry-trap bit. */

#define carryalert countnco;\
              if ((nco+ncojt) >= ccw) {\
                /* carry trap needs to be set */\
                                            \
      nco = ccw - ncojt; \
                                            \
              /* establish position of carry-trap bit */\
                                            \
      ctbptr = ccfmsb + ncojt - ccw + 1;\
                                            \
      /* if carry trap bit is outside ex, then \
              ** it must be in position ccfmsb+1, so   \
      ** simply send zero: */\
                                            \
              if (ctbptr>ccfmsb) {sendbit(0);}\
                                            \
      /* othersize, zero the trap bit and send \
              ** the most significant bit in ccf, which \
              ** must be a 1: */\
                                            \
              else { \
                bitset(ex,ctbptr,0);\
            {sendbit(1);}\
      } \
      }


/*****************************************************************/
/* Macro for shifting ex left, emitting the shifted bit, and conditionally
** setting carry traps. */

#define shiftexout \
     sendbit(bittst(ex,ccfmsb));\
     ex += ex;        /* shift ex left by one */ \
     ex &= efullmask;  /* keep unused part of register clean  */ \
     if (ctbptr <= ccfmsb) ctbptr++; /* if carry trap in ex, shift it */\
                                                  \
     /* if carry trap now outside ex, then conditionally set new trap: */\
     if (ctbptr > ccfmsb) {carryalert;}\


/*****************************************************************/
void encode (alphabetsize, probs, letters, nletters, precision,
     ccwidth, codebits, nbits)
    int alphabetsize, precision, ccwidth, letters[], codebits[];
    double probs[];
    long nletters, *nbits;
{
 /* Encoder registers: */
 codereg ex,ew,ed;

 /* Encoder masks:
 **        efullmask isolates the low-order (pr+ccw) bits of a register */
 codereg efullmask;

 int status;      /* status holds ret'd val from optprnd */
 long n;          /* n is the source letter index */
 int let;        /* let holds current letter */

 /* the following variables are used in handling carries */
 int ncojt;       /* number of consecutive ones just transmitted */
 int nco;         /* number of consecutive ones in high end of ccf. */
 int ctbptr;      /* position of carry-trap bit */
 int ccfmsb;      /* position of most signif. bit in carry-control field */

 /* set file-local variables: */
 p = &probs[0];
 k = alphabetsize;
 pr = precision;
 ccw = ccwidth;

 if(bad_pr_ccw()) {
  fprintf(stderr,"pr and/or ccw have bad values in encode\n");
  exit(-1);
 }

 /* Initialize masks */
 inittwotothe();
 efullmask = twotothe[pr+ccw] - 1;
 ccfmsb = pr + ccw - 1;

 /* Round probabilities to admissible values: */
 status = optprnd();
 if (status!=0) {
  fprintf(stderr,"optprnd failed in encode\n");
  exit(-1);
 }

 /* set ex=0 and ew = 0.111111...: */
 ex = 0;
 ew = twotothe[pr] - 1;

 *nbits = 0;          /* no code bits generated yet */
 ncojt = 0;           /* no consecutive ones just transmitted */
 nco = 0;             /* no ones yet in the high end of ccf */
 ctbptr = ccfmsb + 1; /* initially, the carry-trap bit is not in ccf */

 /** main encoding loop: */
 for (n=0;n<nletters;n++) {

  let = letters[n];     /* current source letter */

  /* add trunc(ew*cumq[let]) to ex: */
  ex += (ew*cumq[let])>>pr;

  /* set ed = truncated product of ew and q[let]: */
  ed = ew*q[let];
  ed = (ed>>pr) << pr;     /* truncate */
  if (ed==0) {
   fprintf(stderr,"subinterval width is zero in encode\n");
   exit(-1);
  }

  /* normalize ed, and in the process, emit code bits: */
  while (bittst(ed,2*pr-1) == 0) {
   shiftexout;
   ed += ed;        /* shift ed left by one for next iteration */
  }
  /* truncate ed to pr bits to get new ew.  since the radix point of ed
  ** is to the left of bit (2*pr-1) and the radix point of ew is
  ** to the left of bit pr-1, obtain ew by shifting ed right by pr bits: */
  ew = ed >> pr;
  /* end of the encoding operation for the nth letter */
 }

 /* conclude by flushing out the code register ex. */
```

```
  for (n=0;n<(pr+ccw);n++) {
    shiftexout;
  }
}

#define shiftdxin \
      dx += dx;         /* shift dx left by one */ \
      if (m<nbits) dx += codebits[m++]; \
      if (bwacptr < ccw) bwacptr++; \
      if (bwacptr >= ccw) { \
        if ((dx&dccmask) == dccmask) { \
          dx += codebits[m++];\
          bwacptr = 0; \
  }\
     }

void decode (alphabetsize, probs, letters, nletters, precision,
      ccwidth, codebits, nbits)
     int alphabetsize, precision, ccwidth, letters[], codebits[];
     double probs[];
     long nletters, nbits;
{
  /* The decoder has five registers.  The code is received in dx, which
  ** consists of two fields:  the carry control field, which occupies the
  ** least significant ccw bits, and the arithmetic field, which occupies
  ** the next pr bits.  Registers dw, dy, and dz have a single field each,
  ** consisting of pr bits. Register dd is a double-precision register
  ** having a single field consisting of 2*pr bits:  */
  codereg dx,dw,dd,dy,dz;

  /* decoder masks:
  **         dccmask isolates the carry-control field (bits o thru ccw-1)
  **         (these masks are initialized in "init()" ):  */
  codereg dccmask;

  int status;   /* status holds ret'd val from init */
  long n;          /* n is the source letter index */
  int let;         /* let holds current letter */
  int m;         /* number of code bits used so far */
  int l;        /* loop index for candidate letters */

  /* the following variables are used in checking for processing carries: */
  int bwacptr;  /* position of bit with carry added to it */

  /* set file-local variables: */
  p = &probs[0];
  k = alphabetsize;
  pr = precision;
  ccw = ccwidth;

  if(bad_pr_ccw()) {
    fprintf(stderr,"pr and/or ccw have bad values in decode\n");
    exit(-1);
  }

  /* Initialize masks */
  inittwotothe();
  dccmask = twotothe[ccw] - 1;

  /* Round probabilities: */
  status = optprnd();
  if (status!=0) {
    fprintf(stderr,"optprnd failed in decode\n");
    exit(-1);
  }

  /* set dx=0 and dw = 0.111111...: */
  dx = 0;
  dw = twotothe[pr] - 1;

  m = 0;          /* no code bits have been processed yet */
  bwacptr = ccw;  /* bit with added carry is not initially in ccf */

  /* shift in pr+2*ccw code bits: */
  for (n=0;n<(pr+ccw+ccw);n++) {
    shiftdxin;
  }

  /** main decoding loop: */
  for (n=0;n<nletters;n++) {

    /* decode current letter by determining what was added to ex. */
    /* note: The use of the following simple search is for */
    /* clarity.  In practice, a binary search should be used to */
    /* prevent linear dependence of computation time on k  */
    dy = dx >> ccw;       /* get arith field of dx in dy */
    let = -1;             /* initial value serves as flag if let unchanged */
    for (l=1;l<k;l++) {
      dz = (cumq[l]*dw) >> pr;
      if (dy < dz) {
        let = l-1;
        break;
      }
    }
    if (let < 0) let = k-1;

    letters[n] = let;
    dx -= ( (cumq[let]*dw) >> pr) << ccw;

    /* set dd = double precision product of dw and q[let]: */
    dd = dw*q[let];

    dd = (dd>>pr) << pr;     /* quantize */
    if (dd==0) {
      fprintf(stderr,"subinterval width is zero in decode\n");
      exit(-1);
    }
    /* normalize dd, and in the process, shift in code bits: */

    while (bittst(dd,2*pr-1) == 0) {
      shiftdxin;
      dd += dd;         /* shift dd left by one */
    }

    /* truncate dd to pr bits to get new dw.  since the radix point of dd
    ** is to the left of bit (2*pr-1) and the radix point of dw is
    ** to the left of bit pr-1, obtain dw by shifting dd right by pr bits: */
    dw = dd >> pr;

    /* end of the decoding operation for the nth letter */
  }

}

/************************* optprnd() *******************************/
/* finds the optimum admissible values for q[], the numbers used in
** place of the actual probabilities p[] by the arithmetic coder.  That is,
** finds even integers {q[i]} that minimize sum p[i]*(-log q[i])
** subject to q[i] >=2 for all i, and sum q[i] = twotothe[pr].  Also computes
** cumq[i] = sum of q[j] for j strictly less than i.
**
** Processing steps:
**    1.  Assign 2 to q[i] for all i.
**    2.  Find benefit[i] of increasing q[i] by one, for all i.
**    3.  While sum of all q[i] < twotothe[pr] do...
**        3a. Find j = arg max(benefit[]).
**        3b. q[j] = q[j] + 1.
**        3c. Recalculate benefit[j] (all others remain the same).
**        (end of "while")
**    4.  Calculate cumq[i] = sum of q[j] for j < i
**    (end of "optprnd")
*/

#define BIGNUM 1.0e20     /* huge value to assign "infinite" cost */

static int optprnd()
{
  double benefit[MAXK], maxbenefit, cost[MAXK], mincost;
  codereg sumq;
  int i,j,l;

  for (i=0;i<k;i++) q[i] = 2;

  /* calculate relative benefit of increasing q[i] by 1: */
  for (i=0;i<k;i++) benefit[i] = p[i] * log(((double) q[i]+1)/((double) q[i]));

  /* keep adding 1's until sumq = twotothe[pr]: */

  sumq=0; for(i=0;i<k;i++) sumq += q[i];

  while (sumq < twotothe[pr]) {

    /* find j = arg max(benefit[]): */
    maxbenefit = 0.0;
    for (i=0;i<k;i++) {
      if (benefit[i] > maxbenefit) {
maxbenefit = benefit[i];
j = i;
      }
    }
    /* increase q[j] by 1 and compute benefit of increasing it further: */
    q[j] += 1;
    sumq += 1;
    benefit[j] = p[j] * log(((double) q[j]+1)/((double) q[j]));
  }
  cumq[0] = 0;
  for (i=1;i<k;i++) cumq[i] = cumq[i-1] + q[i-1];
  return(0);
}
#undef BIGNUM
```

```
/******************* some useful routines **********************/

static void printulong(str,r)   /* prints out a register as 1's and 0's */
     codereg r;
     char *str;
{
  int i;
  if (verbose) {
    printf("%s ",str);
    for (i=31;i>=0;i--) {
      printf("%d",bittst(r,i));
      if ((i==pr)||(i==ccw)) printf(" ");
    }
    printf("    %9.6lf",((double) r)/((double) twototho[pr]));
    printf("\n");
  }
}

static int bad_pr_ccw()       /* checks validity of pr and ccw */
{
  int rw;
  /* pr and ccw should be chosen so that the arithmetic and
  ** carry-control fields can occupy the same physical register, and so
  ** that the result of a double precision multiplication can be stored
  ** in a register.  If the register width is rw, then these restrictions
  ** can be stated as pr+ccw <= rw   and 2*pr <= rw: */
  rw = codereglen();
  if (pr+ccw > rw) {
    fprintf(stderr,"sum of pr and ccw too high in bad_pr_ccw:");
    fprintf(stderr,"%d + %d >  %d (register width)\n",pr,ccw,rw);
    return(-1);
  }
  else return(0);
}

static void inittwototho()    /* initialize array twototho[] */
{
  int i;
  twototho[0] = 1;
  for (i=1;i<32;i++) twototho[i] = 2*twototho[i-1];    /* = 2^i */
}

static int codereglen()   /* determines the length, in bits, of type codereg */
{
  codereg r = ~0;               /* initialize r to all ones */
  int len = 0;
  while (r <<= 1) len++;        /* see how many shifts are require to clear */
  return(len+1);
}

#define SMALLNUM 1.0e-20
static double entropy()     /* returns entropy of source */
{
  int i;
  double sum=0.0;
  for (i=0;i<k;i++) if (p[i] > SMALLNUM) sum -= p[i]*log(p[i]);
  return(sum/log(2.0));
}
#undef SMALLNUM


--------------------------------------------------------------------------------


/* ACMS_EXAMPLE.C */
/* Example main program to demonstrate arithmetic coding and decoding */
/* Ashok Popat 5/89 */

#include <stdio.h>
#include <math.h>  /* for log and pow */

#define MAXK 128    /* max allowable size of source alphabet */
#define MAXN 10000  /* max allowable length of source string */
#define MAXM 90000  /* max allowable length of codestring (be conservative) */

extern void encode(),decode();   /* routines we're trying to demonstrate */

void main()
{
  void gensource(); /* generates random source sequence according to probs*/
  double entropy(); /* computes source entropy */
  int strings_match();  /* to check for correct decoding */

  double avelength; /* average number of bits per letter */
  long nlet,nbits;  /* length of source string and code string */
  int alphsize = 5; /* number of letters in source alphabet */

  /* example probs; change as desired: */
  static double probs[MAXK]={0.2,0.2,0.5998,0.0001,0.0001};
  int sourcebuf[MAXN],codebuf[MAXM],destbuf[MAXN];

  nlet = 10000;

  /* fill up source buffer: */
  gensource(sourcebuf,alphsize,nlet,probs);

  /* encode source, putting resulting bits in codebuf, and set nbits: */
  encode(alphsize,probs,sourcebuf,nlet,16,15,codebuf,&nbits);

  avelength = ((double) nbits)/((double) nlet);  /* ave. per-letter length */

  /* decode, putting result in destbuf: */
  decode(alphsize,probs,destbuf,nlet,16,15,codebuf,nbits);

  /* verify results: */
  if (strings_match(nlet,sourcebuf,destbuf))
    printf("No errors.\n");
  else
    printf("Errors were found.\n");

  /* report performance: */
  printf("Entropy = %lf\nActual codebits per letter = %lf\n",
  entropy(alphsize,probs),avelength);
}

int strings_match(n,a,b)
     long n;
     int *a,*b;
{
  long i;
  for (i=0;i<n;i++) {
    if (*(a+i) != *(b+i)) return(0);
  }
  return(1);
}

#define SMALLNUM 1.0e-14
#define happylog(x) ( x>SMALLNUM ? log(x) : log(SMALLNUM))

double entropy(k,p)
     int k;
     double *p;
{
  double sum = 0.0;

  while (k-- > 0) sum -= *(p+k) * happylog(*(p+k));

  return(sum/log(2.0));
}

void gensource(buf,k,n,p)
     double *p;
     int *buf,k;
     long n;
{
  long random();
  double maxran;
  double ranvar,thresh[MAXK];
  int *bufcopy;
  int i;

  /* divide unit interval into k segments, each corresponding to a
  ** different source letter, and each having a width equal to the
  ** probability of the corresponding source letter.  Select a letter
  ** by generating a uniformly-distributed random number between
  ** 0 and 1, then finding the letter that corresponds to the segment
  ** that contains the random number. */

  maxran = pow(2.0,31.0) - 1.0;  /* max poss. val. of random() */
  bufcopy = buf;
  /* set up thresholds to be cum. prob of letters: */
  thresh[0] = 0;
  for (i=1;i<k;i++) thresh[i] = thresh[i-1] + *(p+i-1);

  while (n-- > 0) {
    ranvar = ((double) random())/maxran;
    for (i=k-1;i>=0;i--) {
      if (thresh[i] < ranvar) {
*bufcopy++ = i;
break;
      }
    }
  }
}
```