Optimum Analog to Digital Conversion

Optional Part II of Final Project for

ECE 475 Fall 2006

by John O'Flaherty

II Optimum Analog to Digital Conversion (optional part of project)

Contents:

Problem description

It is necessary to design a quantizer and encoder system. It is to be optimized for minimum quantization error power by adjusting quantization levels, and for maximum code efficiency by Huffman encoding. The input stream arrives at 20,000 samples per second (a fact that remains irrelevant until the end of the project). Sixty-four levels are specified for the quantizer. The project thus involves two parts, designing a quantizer, and designing an encoder.

Quantizer

The input is Gaussian distributed with a mean of zero and a variance of 10, i.e., a standard deviation of $\sqrt{10}$. A uniform quantizer is the simplest kind of design, but it uses identically-sized intervals to encode ranges of values, regardless of whether the signal spends much time in that range. The average noise power can be reduced if more closely-spaced intervals are used to encode the more common values that occur at the center of the normal distribution curve. This can be an improvement in a system where average noise power is an important parameter. In a system that must limit the maximum possible error, no matter how infrequently it occurs, a uniform system might be required.

The method to achieve a minimum quantization error is that described in classroom instruction. The quantization process involves symbolizing the presence of the signal in a voltage interval by a single value. This is a kind of rounding or approximation, and it occasions an error in the representation of the signal that is simply the difference between the actual signal and the symbol. The associated error power is the square of this difference. A specific number of symbols N, or quantization levels, are used, and the set of quantization levels can be called $\{qL_i\}$. The intervals that are represented by the levels are non-overlapping and cover all or a specified part of the real number line without interruptions. The intervals may be called $\{Q_i\}$. They have upper and lower boundaries that form another set, which can be called $\{X_i\}$. If we propose to adjust the intervals $\{Q_i\}$ to decrease error, we must redistribute the interval boundaries $\{X_i\}$ and the quantization levels $\{qL_i\}$.

The total error power can be measured as a sum of the individual error powers per interval, each calculated by an integral with appropriate limits, weighted according to the PDF of the distribution in that part of the curve. This relationship is shown in this formula:

$$E\left[e_q^{\,2}\right] = \sum_{qL} \int_{X_i}^{X_{i+1}} \left(x - qL_i\right)^2 fX(x)\,dx \qquad 1.$$

where the $(x-qL_i)$ are the error amplitudes, $fX(x)$ is the PDF of the input distribution, X and X_{i+1} are the boundaries of the interval that will be represented by qL_i.

Minimizing this sum requires adjusting the quantization levels and the interval boundaries to achieve minimum noise. Since there are N quantization levels, and N-1 interval boundaries, there are 2N-1 numbers that can affect the results. (This assumes that $\pm\infty$ form the limits of the outermost intervals; these can be difficult to adjust.) According to a paper by Stuart Lloyd [1], "It is a classical result that such a moment assumes its minimum value when each [$\{qL_i\}$ - my notation] is the center of mass" of the corresponding interval. This can be expressed as follows

$$qL_i = \frac{\int_{X_i} x \times fX(x)dx}{\int_{X_i} fX(x)dx} \qquad 2.$$

that is, the denominator of this expression finds the total mass of the interval in question, and so normalizes the PDF of the numerator to unity, and the numerator locates the center of mass of the interval, or the expected value of X on the interval. This minimization expression applies separately to each and every interval. Now with a relationship for the quantization levels, we turn to the intervals, as described by their boundaries. Again from Lloyd [2], "It is straightforward that the best [$\{X_i\}$ - intervals] are determined… as the intervals whose endpoints bisect the segments between successive $\{qL_i\}$…". That is, the interval boundaries are midway between the quantization levels. The crux of the argument is simply that, as formula 1 shows, any movement of the interval boundary decreases the error from one side and increases it from the other side, but the increase is greater than the decrease because of the squaring. This gives us another formula

$$X_i = \frac{qL_i - qL_{i-1}}{2} \qquad 3.$$

The system of simultaneous non-linear equations made from these two formulas is tractable only with numerical methods. It happens that if a uniform distribution is adopted at the start, and then the quantization levels are adjusted to the centers of mass of the intervals, the result will have smaller total noise. At this point, a further reduction can be achieved with formula 3, by setting the interval boundaries to the midpoints between the quantization levels. Then, the quantization levels are no longer at the centers of mass of the intervals, so formula 2 may be applied again. To summarize, this procedure may be repeated, applying formula 2 and then formula 3, in a loop, and all the numbers will converge to a set that gives minimum total error power.

Quantizer implementation

This is an ideal problem for solution by computer, and Matlab is a very apt tool for this purpose, and has been employed here. The program in Appendix B achieves this optimization.

The program is invoked as a function with arguments of the number of quantization levels, the mean and standard deviation of the input distribution (assumed to be Gaussian), and a flag to suppress screen output. It returns three vectors: (1) the optimized quantization levels, (2) the interval boundaries, and (3) the probabilities of the input lying in each interval. The standard deviation and mean are ignored and a distribution based on zero mean and sd = 1 is used. After the optimization is done, it is a simple matter to multiply the results by the sd and add the desired mean.

Within the program, a vector is created to represent the interval boundaries, and stand-ins for $\pm\infty$ are set at 10 σ at the extremes of the vector. The PDF at this distance from the distribution center is at a level of 10^{-22}, so these levels are equivalent to infinity. Given a set of boundaries, a function (`newQlevels`, which appears at the very end of the program) is called, which represents formula 2 by a `trapz` function in a `for` loop, to return a vector of interval boundaries. The program then uses a `for` loop to adjust the interval boundaries midway between the new quantization levels, and uses the new interval boundaries to get yet another new set of quantization levels, in a loop which is terminated only when the new quantization levels deviate from the last set by less than one part in 1 million. This test is performed on only the most negative element of the level vector, since the inner elements are scaled smaller and will all have smaller deviations.

The only other feature of this part of the program worth mention is that it repeats the optimization at three different levels of precision; the current precision level is passed to the optimization function where the integration is done on 10, 100 or 1000 size vectors accordingly. The criterion for going to the next precision level is also modified, so about equal numbers of optimizations are done at each level. This procedure seems to speed the optimization considerably. The 64-level quantizer for this project is produced in about 90 seconds

The last part of the program produces a pair of plots that make the results of the optimization clearer. The plots for the 64-level encoder follow in figures 1 and 2. The first shows the distribution of input voltages, and the optimized interval boundaries and quantization levels. The second graph shows the probability that the input will lie within each optimized interval. Since the plot for 64 levels is a little crowded, a plot for 16 levels is also included as figure 3.

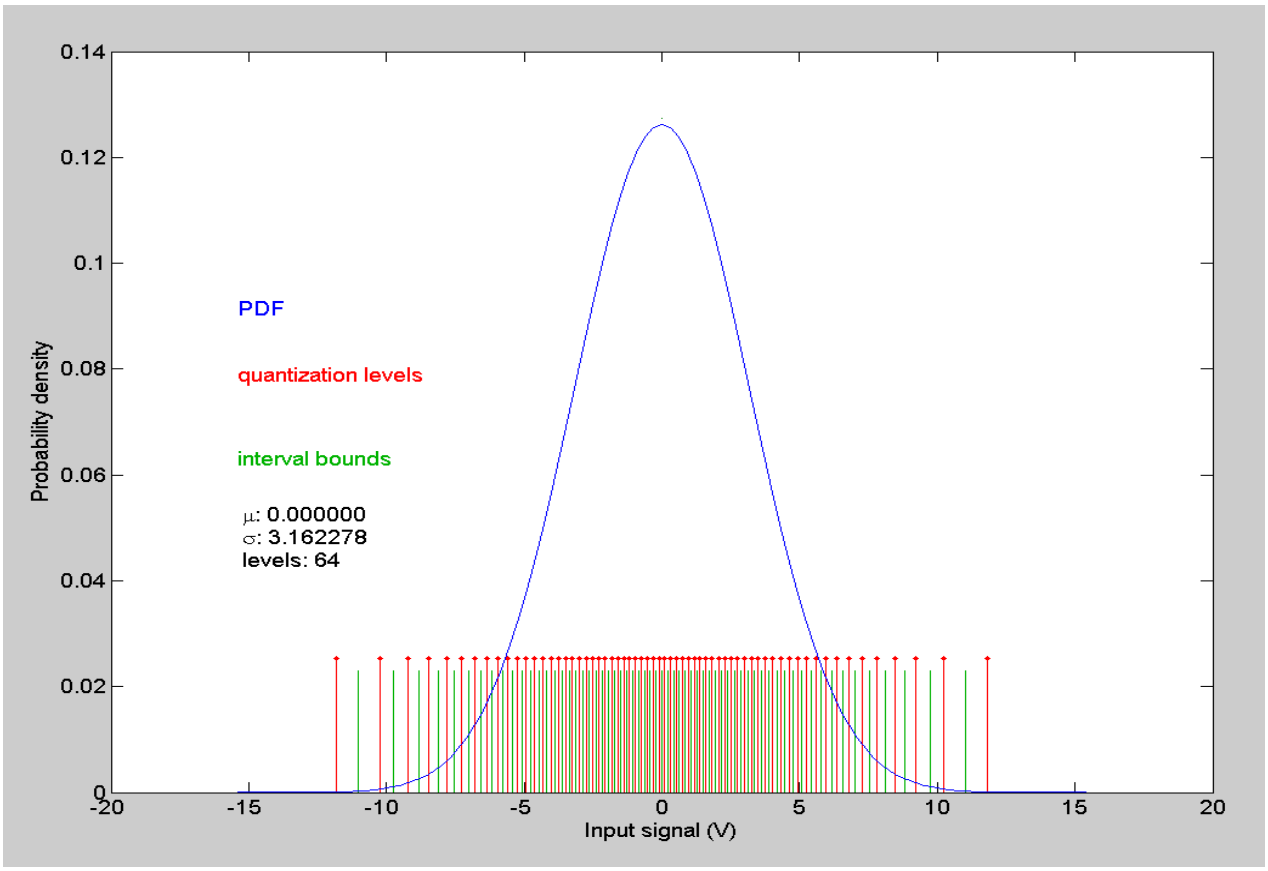

Figure 1. Optimized quantization levels and interval boundaries, and the input Gaussian.

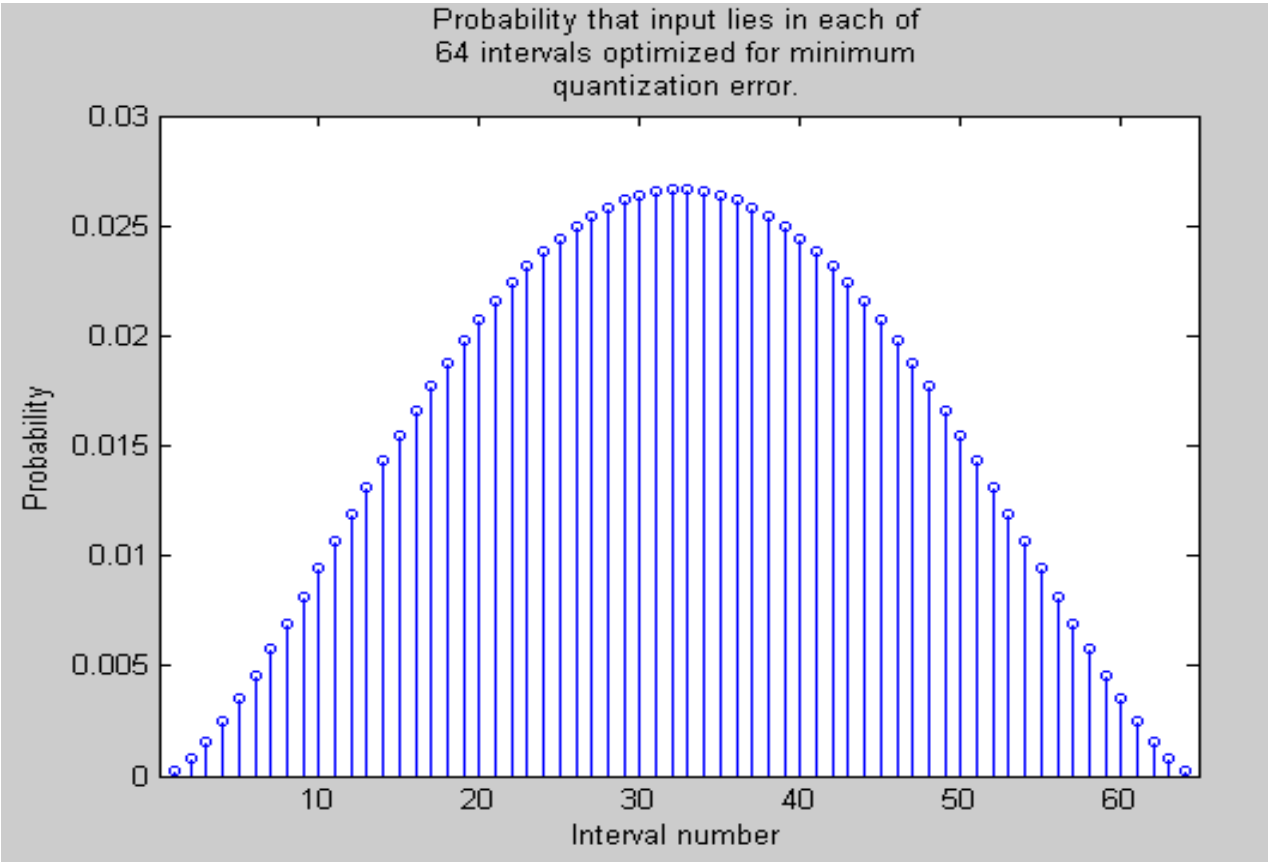

Figure 2. Probabilities of qL_i.

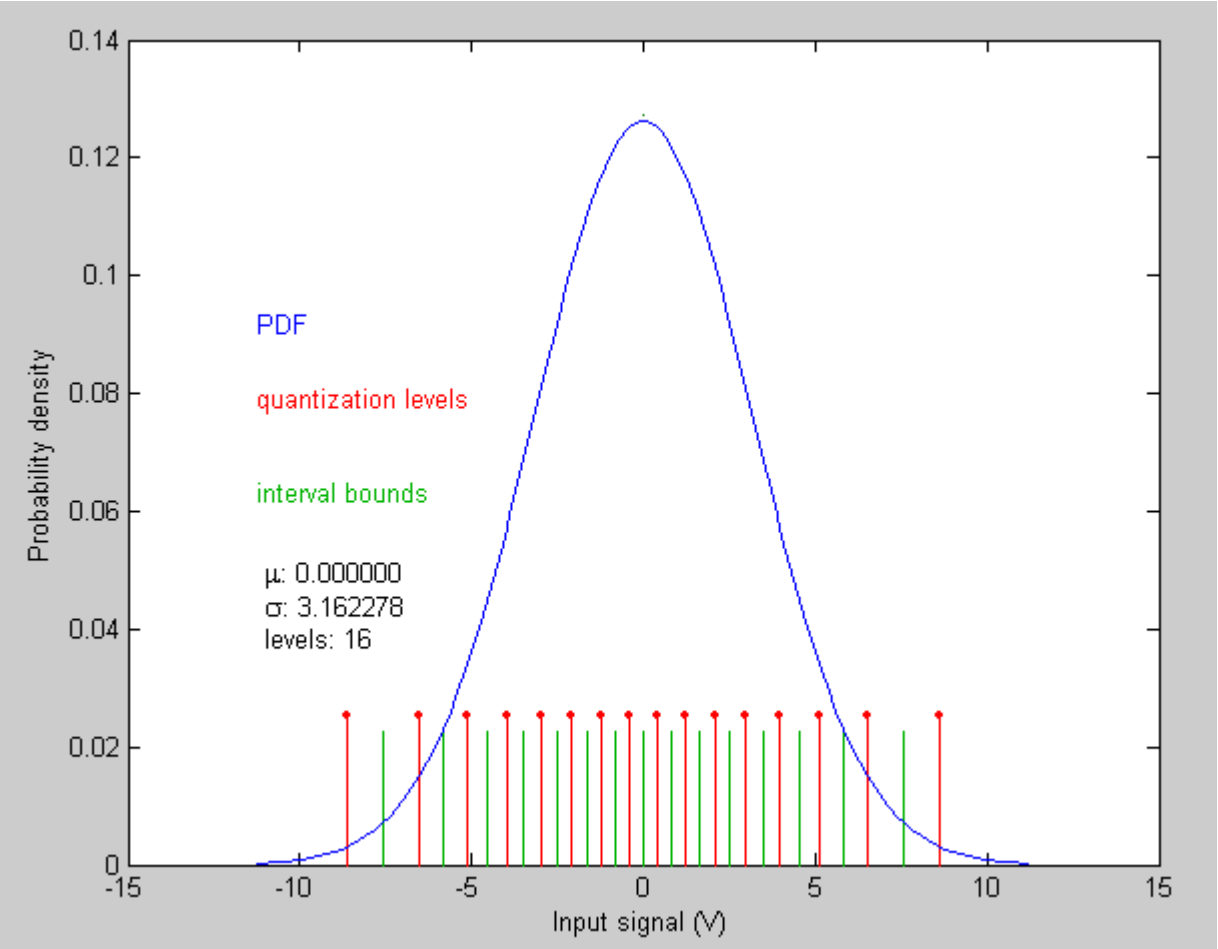

Figure 3. Levels and boundaries for 16-level quantizer.

Quantizer summary

The quantization levels, interval boundaries, and probabilities for the 64-level quantizer of this project appear in a table in Appendix A.

To verify the accuracy of the algorithm, a 1964 paper [3] with tables of optimized quantization levels and interval boundaries was used for comparison. The program presented here produced results essentially identical to those of the published paper for quantizers of various numbers of levels ranging from 4 to 32, including not only numbers that are not integer powers of two, but odd numbers as well. This gives confidence that the results for the 64-level quantizer of the project are accurate.

Encoder theory

Improvements in efficiency for non-uniformly distributed inputs similar to those from quantizer optimization can be had from improvements in encoding. When a set of symbols has a non-uniform distribution, it will carry less information than it could otherwise. When this occurs, it may be possible to use a variable length code and end up with an average bit rate that is closer to the minimum needed to encode the information. The information in a symbol *k* is shown by this formula

$$I_k = \log_2 \frac{1}{P_k} \qquad 4.$$

A simple example clarifies the meaning of this formula -- in a universe of two symbols, say {0,1}, if 1 has probability ½, its information content is $\log_2(1/(½))$, or 1 bit. If this symbol's probability deviates from ½, its information content changes, and it decreases faster for increasing probability than it increases for decreasing probability. When the symbols are considered as a mutually exclusive set, with each symbol in force ½ the time, and with each symbol's information content equal to 1 bit, the total average information of the set is 1 bit. Now, do this calculation when one symbol has probability of 1/3, and the other 2/3. The resulting average information content is 0.9183. The symbol pair can carry one bit of information, but with the indicated probabilities, it carries on average only 0.9183 bits, so its efficiency as a symbol set with this set of probabilities is only 91.83 %. This reasoning can be extended to a larger set of symbols. Such a set can be characterized by its average information content, which is a summation of the information of the individual symbols times the likelihood of each symbol:

$$E[I] = \sum_k I_k P[k] = H \quad or \quad entropy \qquad 5.$$

As in the case of the symbol set {0,1}, average information is maximum when all symbols in the set are equiprobable and the total probability across the set is unity. When this is not the case, it may be desirable to use a variable length code. This will have the more common symbols represented by fewer bits of information, and the less common ones by more. The efficiency of a code can approach but not exceed 100 %, that is, it is impossible to make a code with less average bits per symbol than the entropy of the information encoded. The average bits per symbol in an encoding scheme is the summation of the number of bits representing each symbol, times the probability of that symbol.

$$E[n] = \sum_k n_k P[k] \qquad 6.$$

The value $H \leq E[n]$, and the code efficiency can be represented as the percentage H / E[n].

Encoder implementation

The output of a quantizer such as the one in this project, with its 64 levels, is a symbol set, which must be encoded to a binary symbol set for convenient transmission in a PCM system. It is in this encoding step that efficiency can be improved. It is known that 64 different levels can be transmitted with a uniform length binary code of 6 bits, since $2^6 = 64$ is the number of possible patterns. This scheme would give an efficiency of 100 % if the 64 levels all had the same probability, since, in the formulas above, the P[.]s would all be equal at 1/64, the n_k in formula 6 would all be equal to 6, and the I_k in formula 5 would also all be equal 6, since $\log2(1/(1/64) = 6$.

However, our input has a Gaussian distribution (even after passing through the optimized quantizer), so the code efficiency will be less than 100 %. Its efficiency can be improved by a variable length code, and it has been shown that a Huffman code is optimal for this purpose. Such a code uses fewer bits for more probable levels, and it also has the virtue of being prefix-free, that is, no complete code word is the prefix of any longer code word. This is important in a variable length code, since otherwise one code word could be mistaken for another. As long as the decoder never loses synchronization, there is never any ambiguity about which symbol was sent.

The Huffman code algorithm uses a list of the input symbol probabilities in decreasing order. The two bottom entries in the list are combined into a single symbol with the combined probabilities. The list is then sorted again and the process repeated until all symbols have been combined into a single symbol, forming a binary tree. At each combining step, a one is assigned to the branch contributing more probability to a combined symbol, and a zero to the other. When the tree has been completed, the code words associated with each input symbol can be constructed by proceeding from the final combined symbol back to the original symbol, collecting ones and zeros along the way to form the code word. Since the number of junctions varies from one input symbol to another, a variable length code is produced; because the first probabilities joined were the lowest ones, they will have the largest number of branches, and so the most bits in their encoding. In the case of probability ties between symbols, it is known empirically that the variance in the word length of the output code stream will be less if tied combinations with more sub-branches are sorted higher in the list at each step.

One other thing remains to be said: Huffman codes are not necessarily unique for the same input probabilities, and this goes beyond simply inverting the code. The exact output code will vary, but the number of bits assigned to each input of different probability will be the same, so the average bit rate of the codes will always be the same - optimal. Also, the varying codes will all be prefix-free.

A Matlab program was constructed to accomplish this encoding task. A simplification of the algorithm was made, in that the list of symbols at each level of combination is not sorted. Matlab is able to pick out the two lowest probabilities without sorting the list, and the result is that it is much easier to keep track of the sequences of ones and zeros.

The Matlab program composed to do this encoding is shown in Appendix D. It is constructed as a function, with the input argument being a vector of symbol probabilities or weightings. If the input vector doesn't sum to unity, it is assumed that arbitrary weightings were supplied, and the vector is normalized to sum to unity. A square identity matrix, `trac`, the size of the input vector is created, which tracks the combinations of probabilities that occur. It shows the child nodes for each node, and since it is an identity matrix, it starts out with each node having only itself as a child. The combinations are produced by calling an internal program function `reduc`, at the very end of the program. Its argument is the vector of probabilities. It combines the two least, and returns the vector with the eliminated least probability marked as `NaN`; this symbol will be ignored in subsequent calls to the function. It also returns the index numbers of the two probabilities that were combined. These are put into the `trac` matrix, which then is no longer an identity matrix: the column associated with the higher probability index number will have ones at rows corresponding to itself and to the lower probability index number that is now its child. As mentioned, there was no need to sort the listings at each step, since the Matlab instruction `min` can pick out the two lowest values in an unsorted vector. The output codes are represented as strings, and it greatly simplified matters to construct these at each combining step, rather than trying to trace each symbol back from the output after the construction of the whole tree. At each combining step, then, a one is added to the higher probability's string, and to all those of its children as identified in the `trac` matrix, and a zero is similarly added to the string for the lowest probability, and to those of its children.

Encoder summary

The output of this program is just a sequential list of the discovered Huffman code, along with the entropy of the input probability set, the average number of bits per symbol of the Huffman code, and the implied efficiency. The resulting code may vary a little from other implementations because of the large numbers of exact ties due to the bilateral symmetry of the input vector. However, the number of bits associated with probability levels is the same, and it is the same as the number of bits produced for each probability in the same list by the built-in Matlab function, `huffmandict`. The comparison provides confidence that the program presented here is working correctly.

Overall summary

The overall project results require calculations of the quantization noise of the resulting system. This is accomplished in the program of Appendix C, which is also constructed as a function which returns quantization error power given the quantization levels, the interval boundaries, and an optional mean and sd. If the latter are omitted, they are set at zero and unity respectively. The code implements formula 1, repeated here for convenience,

$$E[e_q^2] = \sum_{qL} \int_{X_i}^{X_{i+1}} (x - qL_i)^2 fX(x) dx$$

to calculate the quantization error power. To calculate the quantized signal power, the code implements the following discrete probability formula:

$$Psig = \sum_{qL} qL_i^2 \times P[qL_i] \qquad \textbf{7.}$$

Rather than pass the already known probabilities per symbol to this program, it recalculates them from the quantization levels, interval boundaries, and statistical parameters, so it can be used as an independent SNR calculator.

The signal to quantization error power is then

$$SNR_Q = \frac{Psig}{E[e_q^2]} \qquad \textbf{8.}$$

The results of these calculations for the project system are shown in Appendix A.

The output of this program was also compared to the error powers shown in a published paper [3], for quantizers of various numbers of levels up to 32, and the results were virtually identical, which again gives confidence that the program presented here is accurate for the 64-level quantizer of the project.

The last project requirement is to estimate the encoder output in bits per second, given a system sample rate of 20,000 per second. This is easiest of all, because the average bits per symbol, 5.75, were calculated by the Huffman encoder program, and the result is just the product of the two, or 115 kb/s.

All of the quantitative project results are presented in Appendix A.

References:

1,2 Stuart Lloyd, *Least squares quantization in PCM*, IEEE Transactions on Information Theory, 28(2), March 1982.

3 Joel Max, *Quantizing for minimum distortion*, IRE Transactions on Information Theory, March, 1960

Appendix A Project part II results

A table with results by quantization level appears on the next page.
The quantization levels and interval boundaries were found by the Matlab program in Appendix B for a 64 level quantizer with a Gaussian input of zero mean and variance = 10. The quantization error power is from the program in Appendix C. The Huffman code is from the program in Appendix D. (The interval boundaries in the table are offset upward by ½ table position.)

Error power: $\underline{Eq^2 = 0.006442\ W\ (1\Omega)}$

Signal power: 9.99354 W (1Ω)

Signal to quantization error ratio: $\underline{SNR_{Eq} = 1551 = 31.91\ db}$

Entropy of the quantized input: 5.710

Avg. word length of Huffman code = 5.750 (weighted by the probability of the word)
The word length varies from 5 to 12, with the unweighted mean being 6.64.

The code efficiency is 99.3 % (compare this to a uniform 6-bit code, with 95.2 % for the given probability set).

The expected value of the bit rate is simply the input sample rate times the average bits per value,

$2*10^3$ samples/sec * 5.750 bits / sample = $\underline{115\ kbits / sec.}$

Appendix A, continued

Level #	Quantize Level	Probability of level	Interval Boundary	Huffman Code	#bits
			+∞		
1	-11.8380	0.000240	-11.0420	101100100101	12
2	-10.2450	0.000801	-9.7347	10110010011	11
3	-9.2239	0.001560	-8.8363	1101100110	10
4	-8.4487	0.002466	-8.1309	110110010	9
5	-7.8131	0.003483	-7.5406	10011101	8
6	-7.2681	0.004582	-7.0275	10110011	8
7	-6.7869	0.005741	-6.5700	11100111	8
8	-6.3530	0.006944	-6.1544	1001001	7
9	-5.9557	0.008173	-5.7716	1011000	7
10	-5.5875	0.009417	-5.4153	1100010	7
11	-5.2430	0.010664	-5.0805	1110010	7
12	-4.9181	0.011905	-4.7638	1111100	7
13	-4.6096	0.013130	-4.4623	010110	6
14	-4.3151	0.014331	-4.1739	100101	6
15	-4.0326	0.015502	-3.8965	101000	6
16	-3.7605	0.016637	-3.6289	101011	6
17	-3.4973	0.017730	-3.3697	101101	6
18	-3.2420	0.018775	-3.1178	110000	6
19	-2.9936	0.019769	-2.8724	110010	6
20	-2.7512	0.020707	-2.6327	110100	6
21	-2.5141	0.021586	-2.3979	111000	6
22	-2.2816	0.022402	-2.1674	111011	6
23	-2.0532	0.023153	-1.9408	111101	6
24	-1.8283	0.023836	-1.7174	111111	6
25	-1.6065	0.024448	-1.4969	00001	5
26	-1.3872	0.024988	-1.2787	00100	5
27	-1.1702	0.025453	-1.0626	00101	5
28	-0.9550	0.025844	-0.8482	01000	5
29	-0.7413	0.026157	-0.6350	01010	5
30	-0.5287	0.026393	-0.4228	01100	5
31	-0.3169	0.026551	-0.2113	01110	5
32	-0.1056	0.026630	0.0000	10000	5
33	0.1056	0.026630	0.2113	10001	5
34	0.3169	0.026551	0.4228	01111	5
35	0.5287	0.026393	0.6350	01101	5
36	0.7413	0.026157	0.8482	01001	5
37	0.9550	0.025844	1.0626	00111	5
38	1.1702	0.025453	1.2787	00110	5
39	1.3872	0.024988	1.4969	00011	5
40	1.6065	0.024448	1.7174	00010	5
41	1.8283	0.023836	1.9408	00000	5
42	2.0532	0.023153	2.1674	111100	6
43	2.2816	0.022402	2.3979	111010	6
44	2.5141	0.021586	2.6327	110111	6
45	2.7512	0.020707	2.8724	110101	6
46	2.9936	0.019769	3.1178	110011	6
47	3.2420	0.018775	3.3697	101111	6
48	3.4973	0.017730	3.6289	101110	6
49	3.7605	0.016637	3.8965	101010	6
50	4.0326	0.015502	4.1739	101001	6
51	4.3151	0.014331	4.4623	100110	6
52	4.6096	0.013130	4.7638	010111	6
53	4.9181	0.011905	5.0805	1111101	7
54	5.2430	0.010664	5.4153	1101101	7
55	5.5875	0.009417	5.7716	1100011	7
56	5.9557	0.008173	6.1544	1001111	7
57	6.3530	0.006944	6.5700	1001000	7
58	6.7869	0.005741	7.0275	11100110	8
59	7.2681	0.004582	7.5406	11011000	8
60	7.8131	0.003483	8.1309	10011100	8
61	8.4487	0.002466	8.8363	101100101	9
62	9.2239	0.001560	9.7347	1101100111	10
63	10.2450	0.000801	11.0420	1011001000	10
64	11.8380	0.000240	+∞	101100100100	12

Appendix B

Matlab quantizer optimization program.

```
function [qL, Xbound, Pxx] = optQuant(Nlevels, mu, sigma, silent)

% Find an optimized quantization for a Gaussian-distributed input signal
% given the number of bits and the parameters of the distribution.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nlevels is the number of quantization levels    %
% mu and sigma are the Gaussian parameters.       %
% qL will be the set of quantization levels.      %
% Xbound will be the set of interval boundaries. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Track execution time, which can be considerable for Nbits >= 7.
begin = cputime;

% calculate the initial uniform interval boundaries

pinf =  10;     % This gives +/- 10 SDs.
ninf = -10;     % Note that the PDF of 10 sd is about 10^-22.
delta = (pinf - ninf) / (Nlevels);
Xbound = [ninf : delta : pinf];

% Use variable precision loop to speed execution.
% This only becomes important for 6 bits or more.
 loopcount = [0,0,0];

for precision = [1,2,3]

    done = false;    qTest = 1000;   criterion = 10^( -3 -precision);

    while not(done)

        % Position a new set of quantization levels at the PDF centers of gravity of
        % the current interval set.
        qL = newQlevels(Xbound, 10 ^ precision);

        % Now adjust the interval boundaries to midway between quantization levels
        % (except for the two extreme levels, which stay at effectively
        %  +/- infinity).
        for k = 2 : length(Xbound) - 1;
            Xbound(k) = (qL(k) + qL(k - 1)) / 2;
        end
        % See if any significant change since last iteration:
        if (abs(qTest - qL(1)) < criterion) done = true; end
        lastqL = qTest; % Hold this value only for printed feedback to user.
        qTest = qL(1);  % Update the test value.
        %The following is just for printed user feedback:
        loopcount(precision) = loopcount(precision) + 1;
        if (mod(loopcount(precision),100)==0) & (silent == false)
           disp( sprintf('%d : %f',loopcount(precision), lastqL - qL(1)))
        end
    end
end
% Only now are sigma and mu required, since sigma
% is a linear multiplying factor, and mu merely an addition.
qL = qL * sigma + mu;
Xbound = Xbound * sigma + mu;
lastqL = lastqL * sigma + mu;
% Find the cumulative probability in each interval.
% The following statement will sharpen up the cumulative probability to 1.00:
Xbound(1)= -1000; Xbound(length(Xbound))=1000;
```

```
Pxx = zeros(1, Nlevels + 1);
for k = 1 : length(qL);
  cp = normcdf([Xbound(k), Xbound(k + 1)], mu, sigma);
  Pxx(k) = cp(2) - cp(1);
end

Xbound = Xbound(2:length(Xbound)-1); % Here we drop the two outer values
% of the interval matrix, which are stand-ins for +/- infinity. They were
% only needed to get the centers of gravity of the two outer intervals
% in order to position the two outer quantization levels.

% The following is for displaying output, and has no effect on the
% optimization program operation.
if nargin > 3 & silent == false

    disp(sprintf('Loops at each precision level: %d %d %d ',loopcount))
    disp(sprintf('Final two values for qL(1): %f  %f',lastqL, qL(1)))
    disp(sprintf('Elapsed time: %f sec.', cputime - begin ))

% Make a graph if not silent mode.
    figure
    %Plot the quantization levels:
    stem(qL, ones(length(qL)) * normpdf(mu, mu, sigma) / 5, 'Marker','.','Color', [1,0,0])
    hold all;
    %Plot the interval boundaries:
    stem(Xbound, ones(1, length(Xbound)) * normpdf(mu, mu, sigma) / 5.5,'Marker','none',
'Color', [0,.7,0])
    %Plot the PDF optimized to:
    delt = (qL(end)*1.3 - qL(1)*1.3) / 100;
    xrg = qL*1.3 : delt : qL(end)*1.3;
    plot(xrg, normpdf(xrg, mu, sigma), 'Color', [0,0,1])

    % Matlab won't track colors in a mixed stem/continous graph, so
    % put in a manually constructed legend:
    th = xrg(1);
    tv = normpdf(mu, mu, sigma) / 2;
    text(th, tv , 'interval bounds', 'Color', [0,.7,0]);
    text(th, tv * 1.25, 'quantization levels', 'Color', [1,0,0]);
    text(th, tv * 1.45, 'PDF','Color', [0,0,1]);
    s1 = sprintf(' \\mu: %f \n \\sigma: %f \n levels: %d', mu, sigma, Nlevels);
    text(th, tv * 0.7,  s1)

    %Plot a single point so peak of the normal PDF
    % isn't coincident with the top of graph:
     plot(mean(xrg), normpdf(mu, mu, sigma) * 1.01)

    %Label the axes:
    ylabel('Probability density'); xlabel('Input signal (V)');
    figure
    stem(Pxx,'MarkerSize',2)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function qL = newQlevels(X, precision);

qL = zeros(1, length(X) - 1);
% Position the levels to the center of gravity of the current interval,
% in the PDF.
for k = 1 : length(qL);
    x = [X(k) : ( (X(k + 1) - X(k)) / precision) : X(k + 1)]; % The trapz argument vector.
    weight = trapz( normpdf(x) ) / precision;   % Factor to normalize the PDF.
    qL(k) = trapz( x .* (normpdf(x))) / precision / weight;
end       % The weight factor is a constant for each of these integrations.
```

Appendix C

Matlab quantization error power calculator.

```
function Eq = quantNoise(qLevel,xVal, mu, sigma);

%  This function calculates quantization error power (Eq^2)
% for a set of input signal ranges (xVal), and a set of
% quantization levels (qLevel), and the implied SNRe or
% signal-to-quantization error ratio. It assumes the signal is
% Gaussian distributed, with mu = 0 and sigma = 1, unless
% these are specified in the last two input arguments.

%  If the quantization level vector is longer than the boundary
% vector by 1, the program assumes that the dynamic range is
% +/- infinity and includes that range in the error total.
% If the boundary level vector is longer by one, the program
% assumes input is strictly limited to the boundary vector range,
% that is, no separate overload error is calculated.
% If an overload error level is needed for this case, shorten the
% boundary level vector by one on each side, and run the function
% again. This will give a total error including infinite range, and
% the specific overload error will be the difference between the
% total and the first run.

% Set default values for mu and sigma if necessary.
if nargin < 4 sigma = 1; end
if nargin < 3 mu = 0; end

lq = length(qLevel);
lx = length(xVal);

% Set some numbers to stand in for infinity.
% The normal pdf at 10 sigma is about 10^-22.
if lx < lq
pinf =  sigma * 10 + mu;     % This gives +/- 10 SDs,
ninf = -sigma * 10 + mu;     %  centered on mu.
xVal = horzcat(ninf, xVal, pinf);
end

lx = length(xVal);


% Calculate quantization error over each interval and sum.
Eq = 0; IntvProb = zeros(1,length(lq));
for k = 1 : lq
    xstep = (xVal(k + 1) - xVal(k)) / 1e5; % Fine division (1e5) only needed to maintain
    x = [xVal(k) : xstep : xVal(k + 1)];   % accuracy in the end ranges, but still runs fast.
    Eq = Eq + trapz( (x - qLevel(k) ).^2 .* normpdf(x, mu, sigma)) * xstep;
    IntvProb(k) = trapz(normpdf(x, mu, sigma)) * xstep;
end

% Calculate the signal power of the discrete distribution of quantization
% levels. This amounts to a sum over all the intervals of the signal voltage for each
interval
% squared and weighted by the probability of the interval.

SigPwr = sum(qLevel.^2 .* IntvProb);
SNR = SigPwr / Eq;
SNRdb = 10 * log10(SNR);

disp(sprintf('Signal power %0.5f (Watts into 1 ohm)' ,SigPwr))
disp(sprintf('Quantization noise power: %0.5f {Watts into 1 ohm}',Eq))
disp(sprintf('SNRe: %0.2f  =  %0.2f db',SNR,SNRdb ))
```

Appendix D

Matlab Huffman code generator.

```
% This function returns a Huffman code given a set of symbol
% probabilities, calculating the entropy of the input, and the average
% code word length and efficiency of the Huffman code produced.
% Neither the input nor the intermediate codes lists are sorted,
% since this complicates things without benefit.

function [code]= huffman(probs);
% The following 'if' allows use of an arbitrary list of weightings on the
% input, which will be normalized to a total probability of one.
if not(sum(probs)==1)
    xin = probs./sum(probs);
else xin = probs;
end
trac=eye(length(xin)); % a square identity matrix, size of input
code=cell(length(xin),1);

x1=xin;
for cnt=2:length(xin);
    [x1,ndx1,ndx2]=reduc(x1);
    for k=1:length(xin)
        if trac(k,ndx2)>0
            code{k}=horzcat('1',code{k});
        end
    end
    for k=1:length(xin)
        if trac(k,ndx1)>0
            code{k}=horzcat('0',code{k});
        end
    end
    % the trace vector marks combined symbols
    % by adding ones, departing from the identity matrix
    trac(:,ndx2)=trac(:,ndx2)+trac(:,ndx1);
 end

% Calculate entropy of original input, avg. word length
% of Huffman code, and efficiency of the code.
entropy=-sum(log2(xin).*xin);
avgLength = 0;
for cnt=1:length(xin)
    avgLength=xin(cnt)*length(code{cnt})+ avgLength;
end
% print the results...
 disp(' ')
disp(sprintf('%s %0.3f', 'entropy of input = ', entropy))
disp(sprintf('%s %0.3f', 'avg. word length of Huffman code = ', avgLength))
disp(sprintf('%s %0.1f %s', 'code efficiency', entropy/avgLength*100,'%'))
for cnt=1:length(xin)
 disp(sprintf('%d  %.5f:   %s',cnt,xin(cnt),code{cnt}))
end
disp(' ')
% Subfunction returns input array with two least member probabilities
% added into one, and NaN substituted for the least probable, and returns
% the index numbers of the two that were combined.
function [xout,ndx1,ndx2] = reduc(xin);
x1=xin;
[m1,ndx1]=min(x1);
x1(ndx1)=NaN;
[m2,ndx2]=min(x1);
xout=xin;
xout(ndx1)=NaN;
% Force a combined member to value higher in a tie by adding a small number to it.
xout(ndx2)=xin(ndx1)+xin(ndx2)+1e-6;
```

Appendix E

Matlab screen output from the three programs, for the problem of this project:

```
>>[level,intvl,prob]optQuant(64,0,sqrt(10),0);

100 : -0.019153
200 : -0.008250
300 : -0.004453
400 : -0.002702
500 : -0.001765
600 : -0.001214
700 : -0.000866
800 : -0.000636
900 : -0.000476
1000 : -0.000363
1100 : -0.000279
1200 : -0.000217
1300 : -0.000170
1400 : -0.000134
1500 : -0.000106
100 : 0.000489
200 : 0.000153
300 : 0.000072
400 : 0.000041
500 : 0.000027
600 : 0.000019
700 : 0.000014
800 : 0.000011
100 : 0.000013
200 : 0.000008
300 : 0.000006
400 : 0.000005
500 : 0.000004
600 : 0.000003
700 : 0.000002
800 : 0.000002
900 : 0.000001
1000 : 0.000001
Loops at each precision level: 1526 831 1036
Final two values for qL(1): -11.838244
-11.838247
Elapsed time: 89.078125 sec.

>> quantNoise(level,intvl,0,sqrt(10));

Signal power 9.99354 (Watts into 1 ohm)
Quantization noise power: 0.00644 {Watts
into 1 ohm}
SNRe: 1551.21  =  31.91 db

>> huffman(prob);

entropy of input =  5.710
avg. word length of Huffman code =  5.750
code efficiency 99.3 %

1  0.00024:   101100100101
2  0.00080:   10110010011
3  0.00156:   1101100110
4  0.00247:   110110010
5  0.00348:   10011101
6  0.00458:   10110011
7  0.00574:   11100111
8  0.00694:   1001001
9  0.00817:   1011000
10  0.00942:   1100010
11  0.01066:   1110010
12  0.01190:   1111100
13  0.01313:   010110
14  0.01433:   100101
15  0.01550:   101000
16  0.01664:   101011
17  0.01773:   101101
18  0.01878:   110000
19  0.01977:   110010
20  0.02071:   110100
21  0.02159:   111000
22  0.02240:   111011
23  0.02315:   111101
24  0.02384:   111111
25  0.02445:   00001
26  0.02499:   00100
27  0.02545:   00101
28  0.02584:   01000
29  0.02616:   01010
30  0.02639:   01100
31  0.02655:   01110
32  0.02663:   10000
33  0.02663:   10001
34  0.02655:   01111
35  0.02639:   01101
36  0.02616:   01001
37  0.02584:   00111
38  0.02545:   00110
39  0.02499:   00011
40  0.02445:   00010
41  0.02384:   00000
42  0.02315:   111100
43  0.02240:   111010
44  0.02159:   110111
45  0.02071:   110101
46  0.01977:   110011
47  0.01878:   101111
48  0.01773:   101110
49  0.01664:   101010
50  0.01550:   101001
51  0.01433:   100110
52  0.01313:   010111
53  0.01190:   1111101
54  0.01066:   1101101
55  0.00942:   1100011
56  0.00817:   1001111
57  0.00694:   1001000
58  0.00574:   11100110
59  0.00458:   11011000
60  0.00348:   10011100
61  0.00247:   101100101
62  0.00156:   1101100111
63  0.00080:   1011001000
64  0.00024:   101100100100
```