ANALYSIS OF WYNER'S

ANALOG ENCRYPTION SCHEME

by

Burton Stephen Kaliski, Jr.

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the
requirements of the degree of

BACHELOR OF SCIENCE IN

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1984

Signature of Author Burton S. Kaliski Jr.
Department of EECS May 11, 1984

Certified by Ronald L. Rivest
Thesis supervisor

Accepted by David Gifford
Chairman, EECS Department Committee

ACKNOWLEDGEMENTS

I am grateful to Professor Victor Zue and Scott Cyphers of the M.I.T. Speech Laboratory for their assistance in obtaining and studying speech samples; to Professor Ronald Rivest for his advising of this project; and to Aaron Wyner, for a timely visit to M.I.T. one week before the completion of this thesis.

I also wish to thank Cathy Oehl for her love, patience, and support during the long hours spent in research and writing.

Most importantly, for the blessings of an M.I.T. education and the strength to endure one, I thank the Lord Jesus Christ, to Whom my career and life are dedicated.

TABLE OF CONTENTS

SECTION 1 -- INTRODUCTION

In 1979, Aaron Wyner of Bell Telephone Laboratories published a pair of papers which answered the theoretical question, can speech be scrambled and then transmitted through a telephone? He made a mathematical argument for existence of a scrambler with such a property, and showed it could be extremely secure.

Since that time, Bell has developed a need for a scrambler for mobile radio with the same property, and all of telecommunications is becoming aware of the need for security in any transmission. Wyner's scheme is one of those being considered for a standard for all devices in a product line requiring speech encryption.

This thesis is concerned with the need for that scrambler, its properties, and its effectiveness in providing secure and accurate speech encryption. A simulation of a speech scrambler is tested on a Lisp machine to verify the theoretical properties, in preparation for a future hardware implemenation.

There are four main sections to the thesis. The first is a statement of the problem, and an attempt at solving it in a naive fashion. The second contains the results of the simulation. It is suggested that section be read before the remaining two, then reread in light of the description of the scrambler and its simulation. The third and fourth sections describe the properties of the scrambler, and how one is simulated on a Lisp machine.

SECTION 2 -- THE PROBLEM

Modern telecommunications is becoming increasingly in need of security, both for digital and analog transmission. A great deal of effort has been devoted to digital encryption, and research is underway in secure methods of scrambling speech signals. An ideal scrambler would produce output that could be transmitted over any channel that the input could be. The question is, how fast, how secure, and how accurate can such a scrambler be?

Section 2.1 reviews the current needs for such scrambling systems. In particular, a need for narrow-band scrambling is explained. In section 2.2, a variety of naive solutions are given in an effort to show the difficulty of designing a scrambler which does not expand bandwidth. The scrambler, performing its encryption in the digital domain, must meet certain performance constraints; these are shown in section 2.4. Wyner's scrambling method, as an intuitive argument, is presented in section 2.3.

2.1 NEED FOR SCRAMBLER

Security in communications is becoming more important in the modern world. Law enforcement agencies desire secure scrambling of voice transmission [Nelson]. Radio scrambling has long been used in the military. Mobile radio transmission of telephone communication is easily intercepted. But most conventional systems rely on wide-band transmission channels, and hence are unsuited for narrow-band applications.

What is needed is a narrow-band scrambler. One case in which this is evident is that of the Dallas police department. It installed a relatively modern scrambling system between patrol cars and police headquarters in the 1970s. But the transmission line between the two points had several relays between radio antennas, and used telephone lines in some places. What worked well for plaintext speech failed for ciphertext speech. The narrow-band parts of the channel destroyed much of the information in the scrambled signal.

A narrow-band scrambler would provide users of telephone networks a simple method of producing security in communication. An accurate scrambler would enhance operation even more, allowing users to identify one another (as much as in ordinary telephone speech, as opposed to merely understanding what is said). This would allow a way of "authentication" similar to that provided in most data encryption systems.

Fear of Soviet interception of U.S. microwave broadcast of unencrypted telephone speech led to a desire for security in that area in the early 1970s. Wyner's original research then answered the

key theoretical question: Are bandlimitedness and security mutually exclusive? The answer, in theory, is yes. Bell Laboratories, according to [Wyner 84], is now seeking to make secure the transmission of telephone communication at FM frequencies, which is easily tapped and in fact carries no legal penalty for such interception. A new project to develop a standard encryption unit for all mobile radio requires a scrambling system such as Wyner's and may in fact use his.

There are some problems with building such a system, however. Wide-band scramblers can use the large channel capacity to reduce the work necessary to scramble a signal, and hence most are relatively cheap and simple. A narrow-band scrambler must produce a signal with the same band characteristics of its input, and thus more complicated equipment is needed. In addition, permutation of the frequency spectrum (a method used by many simple scramblers) is no longer effective, and the operation of the scrambler moves into the digital domain. This requires a more powerful processor than a driver for some analog circuitry. A scheme must combine security, bandlimitedness, and speed to be an acceptable solution for the telecommunications industry.

2.2 POSSIBLE ANALOG ENCRYPTION SCHEMES

We consider here several simplified analog encryption schemes, and show why decomposition into prolate spheroidal sequences is the preferred method.

The first scheme is time-sample scrambling. Let the input $x[n]$ be broken into blocks $x_k[n]$, where

$$x_k[i] = \begin{cases} x[i+kN], & 0 \leq i < N \\ 0, & \text{otherwise} \end{cases}$$

Each $x_k[n]$ is then a vector in R^N. The output of the scrambler is blocks $y_k[n]$, where $y_k = M_k\, x_k$, for some randomly-selected orthogonal matrix M_k. The output $y[n]$ is then the concatenation of the smaller blocks, or

$$y[i] = \sum_{k=-\infty}^{\infty} y_k[i-kN]$$

Clearly if we desire $y[n]$ to be a sequence approximately limited to the band $[W_1, W_2]$ (where the band is within $[0, .5]$), this method is not likely to work. Indeed it provides high security, but the randomness of the M_k causes the y_k to be distributed uniformly on R^N. Hence sequences outside the band are equally well represented, and the independence of successive M_k makes the expected frequency distribution of $y[n]$ uniform across the band $[0, .5]$.

The problem is the size of the M_k. We seek to preserve the lack of frequencies outside $[W_1, W_2]$, but an N x N matrix M_k scrambles into those frequncies as well. The desired output really only has $2(W_2 - W_1)N$ degrees of freedom, and the M_k really should scramble no more than that number of items. One naive solution immediately comes to mind -- frequency scrambling, in which only those components in the desired band are exchanged by the scrambler.

Let $X_k[f]$ be the inverse discrete Fourier transform of $x_k[n]$ defined above. Then set $Y_k = M_k\, X_k$. In this version, M_k is generated such that elements corresponding to frequencies outside the band cause no permutation, but those within the band are scrambled.

Now $y_k[n]$ is the forward DFT of $Y_k[f]$. By concatenation, we can construct the output $y[n]$. We now show some properties of these $y_k[n]$. Suppose $x[n]$ is a sinusoid of some frequency in the band $[W_1, W_2]$, with an integer number of periods in each N points. Consider the time window

$$h[n] = \begin{cases} 1, & 0 \leq n \leq 63 \\ 0, & \text{otherwise} \end{cases}$$

This has frequency transform

$$H(w) = \sum_{n=0}^{63} e^{-i2\pi wn} = \frac{1}{64}\left(\frac{1 - e^{-64i2\pi w}}{1 - e^{-i2\pi w}}\right)$$

Clearly, this is not uniformly distributed across $[-.5, .5]$. In fact, tabulation of values at various points shows

w	$H(w)^2$
0.	1.0
.01	.20
.02	.037
.03	.0017
.04	.015
.25	0.0
.48	.00014
.49	.00020

What this all means is that although the sequence $h[n] = 1$ for all n has a frequency transform which is an impulse, the sequence limited to some set of n has a spread-out frequency transform. In the same way, although the sinusoidal $x[n]$ above has two impulses in its frequency transform, taking just $x_k[n]$ will yield a less focused frequency

transform.

The reason is that the discrete Fourier transform assumes the time input is part of an infinite sequence of period N. When $x_k[n]$ is defined so that it is zero outside of a certain range, the DFT will give undesired results. Thus scrambling the $x_k[n]$ in this naive way will have the unintended effect of producing energy outside the band $[W_1, W_2]$.

2.3 AN INTUITIVE SOLUTION

It seems plausible that the N-point sequences bandlimited to $[-W, W]$ over a spectrum of $[-.5, .5]$ form a space of dimension 2WN. All N-point sequences are in R^N, of course. If we restrict them to be defined over a range of 2W in each unit in the frequency domain, then they are likely to be of the space R^{2WN}.

This is Wyner's assumption also. There exists a set of sequences of which 2WN are nearly bandlimited and the rest are nearly outside the band completely. As we discovered above, mere frequency scrambling is insufficient, because segments of sinusoids are not bandlimited.

What intuition says is that a space of dimension 2WN should have that many linearly independent components. This independence implies orthogonality, and it is conventional to represent components as having a length of 1.

Suppose we have these components. Then we can use a digital computer to measure them and mix them, and the output is guaranteed to stay in the vector space R^{2WN}. The problem is the need for a digital computer, as such sequences are unlikely to have any simple physical

meaning which can be adapted in an analog fashion. The benefit, of course, is the near-perfect security provided by a digital scrambler.

2.4 SPEED VS. ACCURACY

This dependence on 2WN components of the N-dimensional space may cause some error. Some information can be lost in the decomposition into a set of weights indicating the relative presence of the components in the basis. The actual permutation of those weights is designed to be orthogonal so that any error in the signal will not be expanded.

We can improve performance by increasing the rate at which samples are taken, introducing redundancy and hence reducing errors. Increasing the number of points N in a block of samples is another way to lower error, because a greater value of N places more energy in the 2WN components.

Meanwhile, we are faced with digital operations which are likely to take time proportional to N^2. A simple matrix-vector multiplication is of this complexity. Suppose N is 32, and sample time T is 125 us. Then the time available to process one set of samples is 4 ms. In this time, perhaps 2048 operations are required, assuming two matrix-vector multiplications. This is only 2 us per operation, and if N is doubled to reduce error, we have only 1 us for each operation.

As a comparative measure of complexity, consider that a typical mainframe computer takes 1 ms to perform a 32-point FFT [vander Steen]. The FFT algorithm is $O(n \lg n)$.

Clearly the required speed presents a problem to the designer.

A small N or large T would require less processor power, but could produce errors greater than some acceptable level. Conversely, a very accurate scrambler may require a dedicated signal processor. The optimal solution needs careful study.

SECTION 3 -- THE RESULTS

This section briefly presents the results obtained from a simulation of the scrambler. It is primarily concerned with the selection of parameters N, v, T, $[W_1, W_2]$, and $[\overline{W}_1, \overline{W}_2]$ that keep the error level low. Samples from an utterance of the word "potato" are used to simulate the properties of speech, and test the many arts of the scrambler.

Section 3.1 gives a general overview of the testing, and the assumptions made which may differ from a real environment. Some features of speech and their effects on the performance of the scrambler are given in the next section. Section 3.3 reviews the trial-and-error methods used to find an optimal set of parameters, leading to a reasonable error level. The effect of channel noise is discussed briefly in section 3.4.

The remaining sections present results not relevant to the actual testing, but of general interest. A possible security flaw and its solution are given in section 3.5. Section 3.6 presents a scenario in which public-key algorithms could be used to make a protocol for the scrambler. Finally, a fast, but as yet unknown, method to perform the scrambling is presented in section 3.7.

3.1 RESULTS OF TESTING

Results of a simulation of Wyner's algorithm show that it can operate at acceptable levels of error without requiring a very powerful processor to perform the encryption. An error level (signal-to-noise ratio) of 13 dB is possible for a sample rate of 8 kHz and block size of 32 points. This error level is called "poor" by some standards (see section 5.5), but it is nonetheless reasonable, given the security provided.

One assumption on which this result is based is that the equalizer used to compensate for linear distortion in the channel is very accurate. The one used in the simulator is accurate to within 30 dB. This requires knowledge of the channel frequency response during design, which is not possible for a real device which can be used on a variety of channels.

The equalizer modeled is also incompatible with a real tapped delay line, in that it uses some 7N taps to restore a signal. Conventional tapped delay lines may have at most 64 taps. Real inaccuracies, however, would affect scrambled speech as much as real speech. For this reason, the results obtained are valid approximations. A real telephone transmission line may introduce some error, but as long as it is, say, 10 dB less than the scrambler's built-in error, the built-in error will be dominant.

The best simulated model of the scrambler uses an input band of [0, 2700] and an output band of [300, 3200]. These bandwidths allow a value of 22 for v, the number of weights scrambled (see section 4.3). Now scrambling can be viewed as an $O(N^2)$ operation if we precompute

a set of matrices which translate an input sequence into weights, scramble them, and convert the weights to an output sequence. In this case, a matrix-vector multiplication is all that is needed, requiring 1024 multiply and 1024 add operations. The 32-point block time of 4 ms allows about 2 us per operation, a speed which can be achieved by many special-purpose array processor chips, including one used by Bell Telephone Laboratories [Wyner 84].

A poor error level, of course, may be unsuitable for many uses. By doubling N, and reducing the time per operation, it may be possible to move the error level down somewhat, but this has not been tested. Some types of errors are acceptable, such as phase shifts. For this reason, errors are measured in a different way than they are defined in section 4.4 -- all phase is removed when making a comparison between actual output and expected output. The human ear operates in the same way.

3.2 CHARACTERISTICS OF SPEECH

Speech signals obtained from the M.I.T. Speech Laboratory appear to be somewhat harder to scramble than Wyner's analysis implies. An utterance of the word "potato" has significant energy in the band [0, 200], ostensibly the pitch of the three vowels. The effect is important to the ear, and hence cannot be removed without making the utterance sound whispered, according to the laboratory.

Most of the energy in a discrete Fourier transform of a block of size 32, 64, 128, or 256 taken from the utterance is in the band [0, 3000].

Smaller sample block sizes have greater concentrations at the DC value, and are narrower. Of course, the entire signal has no DC value, but the samples taken in blocks smaller than one period of the slowest sinusoid will reflect that value into the DC component. This is the result of the discrete Fourier transform.

To scramble all the energy in the speech, a prolate spheroidal basis with a large bandwidth is necessary. In fact, to cap ure all the energy, weights up to those representing vectors almost entirely outside the desired band must be used. As section 4.3 shows, a large input band requires an even larger output band. The question is, will a larger output band still be small enough for the channel?

The question, however, is probably not that important. Like most of the effects and errors studied in the simulation, the consideration of the band of the input is not limited to the scrambler. Any system transmitting real speech will have the same limitations, although to a lesser extent. So one need not consider too greatly the band of the input, because many real systems cut off quite a bit (the ear is a poor receiver of high frequencies) and conform to the channel.

Without an initial bandlimiting of the input signal from "potato" the simulation faces large errors in its transmission line. The 13 dB level mentioned above comes from a non-bandlimited signal. The way in which this level is achieved is described below.

3.3 CHOICE OF PARAMETERS

Testing of the scrambler began with 32-point blocks of points taken

at rates of 16 kHz, the same rate at which they were recorded in a file from original speech samples. The output band $[\overline{W}_1, \overline{W}_2]$ was set to [400, 2600], apparently the largest band the simulated transmission line could handle without great distortion.

This set of parameters failed to produce output resembling the input. The output bandwidth -- only 2200 Hz -- required the number of weights scrambled, v, to be less than 8.8. This in turn required the input bandwidth to be only 1500, from 0 to 1500 Hz. As described above, the speech samples are not well-suited to such bandlimiting. They lost significant energy during the scrambling, and the channel introduced sizeable errors as well.

An initial solution was to reduce the sample rate, thus providing a larger value v and widening the input band. A rate of 10 kHz was tried first, but it made the simulation very slow, since a non-integral down-sampling was necessary. A rate of 8 kHz was tried instead, and with it an input band of [0, 2200], the same bandwidth as the output.

In the second set of trials, the input scrambling again introduced large errors, sometimes cutting half the energy from the input. The band apparently was too small, and the basis itself too inconsistent, since not all the weights for vectors in the desired band were used.

The solution lay in the output bandwidth. Were it larger, more of the input could be scrambled. A change in the definition of the desampler allowed the transmission line to pass a wider band of frequencies. The original desampler had "cut off" the frequencies at $\overline{W}_1$ and $\overline{W}_2$; this was not required of the device in a real system. By extending the band

passed, as one would in a real system, the equalizing was also ma e more effective. The result was a transmission line band of [300, 3200] typical of a real system.

The wider band for output both increased v and extended the input band. An input bandwidth of 2700, with the output bandwidth of 2900, were reasonable for v of 22, and typical errors were 13 dB. Table 3-1 has a list of the energy and error at various parts of the block diagram of the scrambler.

A larger N would probably allow the input and output bandwidth to be brought closer, since it would increase v and the rate of convergence of the vectors around it. Most of the 13 dB error is from the channel. Apparently the input scrambler outputs some vectors which are not entirely in the output band -- 13 dB would be consistent with one in 22. The unscrambler, of course, has no error, since it follows the narrow-band channel. The input scrambler, surprisingly, has little error as well. This serves to confirm that a larger N would improve the channel error, although no such test has been run.

3.4 EFFECT OF CHANNEL NOISE

The only observation of the effect of noise in the channel is from the set of parameters in which input bandwidth was 2200. The simulator introduces a three-block delay between input and the output, due to the nature of the transmission line. The first three blocks output, then, should be near zero energy.

Adding noise equal to about 20 dB less than the typical signal

level in the simulated channel appeared to introduce about half as much noise in the output. For most of the cases with that set of parameters, the built-in error outweighed the added noise.

3.5 SECURITY FLAWS

The scrambler, which relies on orthogonal transformation, does not expand any errors introduced in transmission. But the same property gives away a clue to the original signal. It is easily seen from theory and from experiment that the energy in the channel is proportional to that in the input. Hence an adversary can get a rough image of the speech being transmitted.

Wyner reports that speech experts at Bell feel that the adversary will not be able to learn much from short-time energy measurements [Wyner 84]. The measurements can only determine whether a person is speaking, and the person's gender. It still seems uncomfortable to reveal even that much information.

One solution, Wyner suggests, is to reserve one of the output vectors for use as a dummy. Its weight is defined such that the energy of all the output vectors is constant for all blocks. Indeed, the identity of that vector also will have to be kept random, so an adversary cannot simply filter it away (in the prolate spheroidal sense) and defeat its purpose. Selecting this one dummy vector at random for each block would remove any worry about the security of the system.

3.6 A POSSIBLE PUBLIC-KEY PROTOCOL

Establishing communication between two parties Alice and Bob has two steps. They must first agree on a set of random orthogonal matrices to do the scrambling, and then, each time they want to talk, they must decide in what order to use those matrices. If the matrices are made on the fly (see section 4.5), the first step is unnecessary.

This type of communication seems a good use for a public key protocol. Since Alice knows Bob's phone number when she calls, it's not unreasonable to assume she can also look up his public key. Alternatively, she could access a network designed for this protocol that would determine Bob's public key based on his phone number from a large data file.

The callers can first identify themselves by communicating in digital form using the public keys. Now the only problem is the same one on a real phone call -- Carla could pretend to be Alice to fool Bob, and he would not know until she had spoken a few words. In the same way, Bob does not know who is calling him at the first transmission, but he will at the second, requiring Alice to sign her message.

After these three transactions -- Alice to Bob, Bob to Alice, and Alice to Bob again, the seed for a random number generator can be exchanged. Bob may send it to Alice, with his signature; she can then return it, with her signature, and both can switch from digital to analog mode.

When they're done, they just hang up.

3.7 A FAST TRANSFORM?

One final note concerns an algorithm which would vastly improve the speed of the scrambler. The key time-consuming operations are those to transform a signal from time domain to prolate spheroidal weights. If a block of 32 samples is converted to 22 weights, those weights are scrambled, and then they are built into another block of 32 samples, then 22*32 + 22*22 + 22*32 = 1892 operations are required. (For this reason a 32 x 32 matrix multiplication is preferred.)

However, if a fast transform of O(n lg n) were available, then only 5*32 + 22*22 + 5*32 = 804 operations are needed. Although no such algorithm has been found, the prolate spheroidal sequences do have many interesting properties which may make their weights easier to find. An algorithm to do this would be of great mathematical interest.

Scrambler			Unscrambler		
input	output	error	input	output	error
41.68	41.04	18 dB	*		
60.8	57.77	13	*		
70.9	68.9	15	*		
150.3	138.1	11	32.9	32.8	25 dB
35.6	34.4	15	51.5	51.4	27
54.8	52.0	13	63.3	63.1	25
69.3	67.2	15	131.3	131.2	31
157.4	141.7	10	27.95	27.92	30

Channel			Complete system**		
input	output	error	input	output	error
41.04	32.9	7 dB	41.68	32.8	15 dB
57.77	51.5	10	60.8	51.4	14
68.9	63.3	11	70.9	63.1	19
138.1	131.3	13	150.3	131.2	11
34.4	27.95	7	35.6	27.92	13

* - no input because of three-block delay

** - error measured with phase removed; is always less than error between energy levels

Table 3-1 Sources of errors in simulated scrambler on 8 blocks of input

SECTION 4 -- THE SOLUTION

Spheroidal functions, which have several applications in physics, provide the key to a narrow-band scrambling scheme. They relate the indexlimited sequences to the bandlimited ones, and show how well one can be represented in terms of the other. By using these functions, in a form called discrete prolate spheroidal sequences, it is possible to scramble certain characteristics of a signal with high security and little error.

Section 4.1 reviews the history of spheroidal functions, and their application to the problem at hand. The nature of the discrete prolate spheroidal sequences is discussed in section 4.2. Using their properties, a basic version of the scrambler is presented in section 4.3, and a complete version is analyzed in section 4.4. These provide the basis for the software model in section 5. Finally, section 4.5 gives an overview of another issue related to the scrambler: the selection of encryption matrices for maximum security.

4.1 PROLATE SPHEROIDAL WAVE FUNCTIONS

In this section we discuss the history of prolate spheroidal wave functions and some of their mathematical properties. In particular, we show the relationship between indexlimited sequences and bandlimited sequences.

As described above, a voice scrambler must not expand the bandwidth of its input if the output is intended for transmission over a telephone channel. Since we scramble in finite blocks, we must find some way of representing these finite blocks as combinations of indexlimited sequences which are approximately bandlimited.

Consider the following problem, presented in [Slepian]. Suppose we have a sequence $h[n]$. The sequence has a frequency transform $H(w)$, where

$$h[n] = \int_{-1/2}^{1/2} H(w)\, e^{-2\pi i n}\, dw, \quad n = 0, \pm 1, \pm 2, \ldots$$

Assume $h[n]$ is bandlimited to W (that is, $H(w) = 0$ for $W < |w| \leq 1/2$). Since $H(w)$ is periodic, it is also true, for instance, that $H(w) = 0$ for $1 + W < |w| \leq 3/2$.

We seek the sequence $h[n]$ for which a particular set of indices contains the greatest proportion of the energy in the entire sequence, that is, to maximize the ratio of the energy in some set of indices $[N_0, N_0 + N - 1]$ to that in the entire sequence. Recall this is an infinite sequence, so that bandlimitedness really means the same in a periodic sense. The problem then is to find the largest λ, defined as

$$\lambda = \frac{\sum_{i=N_0}^{i=N_0+N-1} \|h[i]\|^2}{\sum_{i=-\infty}^{i=\infty} \|h[i]\|^2}$$

over all sequences $h[n]$ bandlimited to W. The solution is based on the discrete prolate spheroidal wave functions with parameters N and W.

Also note that since $h[n]$ is concentrated to $[N_0, N_0 + N - 1]$, if we set to zero all values outside that range, we lose very little energy. Hence the sequence, now indexlimited, is approximately bandlimited.

Before giving the solution, let us first review a little history of the prolate spheroidal sequences. Mathieu functions and spheroidal functions are special functions of physics first documented in the 19th century. Flammer of Stanford and Stratton, Chu, and Corbato of M.I.T. continued study of these functions in the 1950s. They are the simplest functions of physics arising from a time-independent wave equation [Meixner].

Prolate spheroidal waveforms are of special interest in physics because they are the only eigenfunctions of the finite Fourier transform. In particular, for any real γ and ξ, define $PS_n(z;\gamma^2)$ to be the solution to the integral equation

$$\int_{-1}^{1} e^{i\gamma\xi\eta}\, PS_n(\eta)\, d\eta = 2i^n \alpha_n(\gamma)\, PS_n(\xi)$$

and its iterate

$$\int_{-1}^{1} \frac{\sin \gamma(\xi - \eta)}{\gamma(\xi - \eta)} \, PS_n(\eta) \, d\eta = 2\alpha_n(\gamma)^2 \, PS_n(\xi)$$

The first integral equation shows why the waveforms are useful for mathematical physics. The second equation explains why they are of interest to us. The multiplication by the sin x / x is really a convolution; such a convolution is like filtering or windowing the waveform. The solutions then are those sequences or functions which, over a certain interval in one domain (time or frequency), are only scaled by a constant when filtered or windowed in the other domain. The interval is [-1, 1] in the equation above.

The discrete prolate spheroidal wave functions, on which Wyner bases his scheme for scrambling, are successors to the continuous wave functions discussed above. Landau, Pollak, and Slepian at Bell Laboratories investigated the sequences related to the wave functions as well in a series of five papers published between 1962 and 1977.

Discrete prolate spheroidal sequences are based on a certain interval, or set of indices (just as their continuous counterparts rely on the interval [-1, 1].) Let this set be [1, N]. The sequences also depend on a parameter W, like the γ^2 above. Define the set $v_k(N; W)$ for k = 1 .. N to be the solutions to

$$\sum_{m=1}^{N} \frac{\sin 2\pi W(n - m)}{\pi(n - m)} \, v_k(N; W)[m] = \lambda_k(N; W) \, v_k(N; W)[n]$$

for n = 0, ±1, ±2, ...

It is this relationship which makes the DPSS useful. A large λ_k for some N, W will produce a sequence v_k which is highly concentrated to the indices $[1, N]$; in other words, it loses little energy when it is indexlimited. This produces the most "concentrated" sequence and solves the problem above. It is conventional to let λ_1 be the largest (i.e. closest to 1) of the values λ_k, so that v_1 solves the original problem.

As an additional result, note that since this DPSS is already bandlimited, when it is now limited to a set of indices, it strays little from the desired band. Hence bandlimiting the indexlimited version also loses little energy. It is the set of indexlimited versions we choose for the scrambler, and these are explored in the next section.

4.2 DERIVATION OF PROLATE SPHEROIDAL SEQUENCES

As shown above, a prolate spheroidal sequence does not change its direction, or relative set of values, in a certain set of indices when it is bandlimited. If we look at bandlimiting as a discrete convolution then we can find the discrete prolate spheroidal sequences.

Bandlimiting a sequence $x[n]$ is the same as convolving it with the non-causal impulse response of an "ideal" filter. If the desired band is $[W_1, W_2]$, the filter is

$$\Gamma(w) = \begin{cases} 1, & W_1 < |w| < W_2 \\ 0, & |w| < W_1;\ W_2 < |w| < .5 \end{cases}$$

Its non-causal impulse response is

$$\gamma[n] = \frac{1}{\pi n} (\sin 2\pi W_2 n - \sin 2\pi W_1 n), \quad -\infty < n < \infty$$

Then if we bandlimit some $x[n]$, we produce

$$y[n] = \sum_{i=-\infty}^{\infty} x[i]\gamma[n-i] = \sum_{i=0}^{n-1} x[i]\gamma[n-i]$$

or, for the indices from 0 to N - 1, since $y[n] = \lambda x[n]$,

$$y[0] = x[0]\gamma[0] + x[1]\gamma[-1] + \ldots + x[n-1]\gamma[-n+1] = \lambda x[0]$$

$$y[1] = x[0]\gamma[1] + x[1]\gamma[0] + \ldots + x[n-1]\gamma[-n+2] = \lambda x[1]$$

$$\vdots$$

$$y[n-1] = x[0]\gamma[n-1] + x[1]\gamma[n-2] + \ldots + x[n-1]\gamma[0] = \lambda x[n-1]$$

This can be expressed more compactly if we define an N x N matrix K such that

$$K_{ij} = \gamma[i-j]$$

and observe $\vec{y} = K\vec{x} = \lambda\vec{x}$. This eigenvector equation has N solutions which are indeed the discrete prolate spheroidal sequences for the parameters N and $[W_1, W_2]$.

The discrete prolate spheroidal sequences, being eigenvectors of a real symmetric matrix, are hence orthogonal to one another, and they form a basis for R^N. Assume they are normalized, i.e. $\|x_j\| = 1$.

Consider the eigenvalue λ_j corresponding to each eigenvector x_j. If we define

$$\phi_j[i] = x_{ji}, \quad 1 \leq i \leq N$$

$$c_j = B\phi_j$$

that is, the sequence resulting from bandlimiting x_j to $[W_1, W_2]$, then

$$c_j[i] = \lambda_j \phi_j[i], \quad 1 \leq i \leq N$$

The energy of ϕ_j in the desired band is

$$\int_{W_1 \le |w| \le W_2} |\Phi_j(w)|^2 \, dw = \int_{-1/2}^{1/2} |\Phi_j(w) \Gamma(w)|^* \Phi_j(w) \, dw = \langle c_j, \phi_j \rangle$$

$$= \sum_{i=1}^{N} c_j[i] \phi_j[i] = \sum_{i=1}^{N} \lambda_j \phi_j[i]^2 = \lambda_j \|\phi_j\|^2 = \lambda_j$$

Hence the energy of ϕ_j in the desired band is exactly λ_j, and since energy cannot be <u>restricted</u> to a given band and set of indices, we know $0 < \lambda_j < 1$.

Label the λ_j such that $\lambda_1 \ge \lambda_2 \ge \ldots \ge \lambda_N$. We know that bandlimiting ϕ_j reduces its energy from 1 to λ_j; also, the amplitudes in the range $[1, N]$ are reduced by the factor λ_j, so that their combined energy becomes λ_j^2. Thus the energy outside the indices increases from 0 to $\lambda_j(1 - \lambda_j)$. We would like to minimize the energy which strays outside the set of indices and which is cut off by bandlimiting. This requires λ_j close to 1. The question is, how many and how close to 1 are the λ_j?

Observe that, since the sum of the eigenvalues of a matrix is the same as the sum of the elements along the diagonal,

$$\sum_{j=1}^{N} \lambda_j = \text{trace } K = 2(W_2 - W_1)N$$

Next, by squaring the eigenvalues, we have, using a trick in [Landau],

$$\sum_{j=1}^{N} \lambda_j^2 = \text{trace } (K^T K) = \sum_{n,m=1}^{N} \gamma^2(n-m)$$

$$\le 2(W_2 - W_1)N - O(\log N)$$

as $N \to \infty$. For any δ, the proportion of eigenvalues λ_j such that $\delta < \lambda_j$

$< 1 - \delta$, approaches 0 as $N \to \infty$. All λ_j approach either 1 or 0 to satisfy the limit above.

Wyner and [Slepian] prove other useful theorems about the rates at which the λ_j approach 0 or 1. In particular, the separation occurs at λ_{2WN}, in that

$$\lambda_n \approx 1, \quad 1 \leq n \leq 2(W_2 - W_1)(1 - \epsilon)$$

$$\lambda_n \approx 0, \quad 2(W_2 - W_1)(1 + \epsilon) \leq n \leq N$$

These sequences form a basis for R^N, since they are orthogonal, and, as desired, the first 2WN or $2(W_2 - W_1)N$ of them form an approximate basis for the near-bandlimited sequences on $[1, N]$.

By performing a prolate spheroidal transform, we are able to determine $2(W_2 - W_1)N$ values which describe accurately a section of a bandlimited signal. These can be scrambled and recomposed.

4.3 SIMPLIFIED SCRAMBLING SCHEME

Having shown how to find a basis for the approximately bandlimited sequences over $[1, N]$, we now show how to use such a basis to scramble sequences. Define the operator $\mathcal{B}$ to bandlimit a sequence to $[W_1, W_2]$, and define C(a) for a sequence a[n] to be

$$C(a) \triangleq \frac{\|\mathcal{B}a\|^2}{\|a\|^2}$$

that is, the concentration of a[n] to the desired band. Note that a[n] is defined for all integers n, but is zero outside the region $[1, N]$.

For any such sequence a[n], the "contribution" or "weight" α_j of the discrete prolate spheroidal sequence ϕ_j is the dot product, or

$$\alpha_j = \sum_{i=1}^{N} a[i]\,\phi_j[i]$$

To see this is true, observe the recomposition of the weights, recalling that $\langle \phi_j, \phi_k \rangle = 1$ if and only if $j = k$, since the eigenvectors are orthogonal.

$$a[i] = \sum_{j=1}^{N} \alpha_j \phi_j[i]$$

$$\sum_{i=1}^{N} a[i]\phi_k[i] = \sum_{i=1}^{N} \left(\sum_{j=1}^{N} \alpha_j \phi_j[i] \right) \phi_k[i]$$

$$= \sum_{k=1}^{N} \alpha_j \sum_{i=1}^{N} \phi_j[i]\,\phi_k[i]$$

$$= \alpha_k$$

Now if we expand C(a), it is seen that

$$C(a) = \frac{\|Ba\|^2}{\|a\|^2} = \frac{\left\| \sum_j \alpha_j B\phi_j \right\|^2}{\|a\|^2} = \frac{\sum_{j,k} \alpha_j \alpha_k \langle B\phi_j, B\phi_k \rangle}{\|a\|^2}$$

$$= \frac{\sum_{j,k} \alpha_j \alpha_k \lambda_j}{\|a\|^2} \text{ iff } j = k, \text{ or } \frac{\sum_j \alpha_j^2 \lambda_j}{\sum_j \alpha_j^2}$$

We have seen that $\lambda_j \approx 0$ for $j = 2(W_2 - W_1)(1 + \epsilon)$. Thus, for large N,

$$C(a) \approx \frac{\sum_{j=1}^{2(W_2 - W_1)N} \alpha_j^2}{\|a\|^2}$$

The scrambler permutes these first $2(W_2 - W_1)N$ weights.

Let M be a v x v orthogonal matrix, where $v = 2(W_2 - W_1)N$. Then let the encrypted weights $\vec{\beta}$ be defined as $\vec{\beta} = (\beta_1, \ldots, \beta_v) = M\vec{\alpha}$ The output sequence of the scrambler, for one block, is then

$$a[i] + \sum_{j=1}^{v} (\beta_j - \alpha_j)\, \phi_j[i]$$

so that we do not lose the parts of a[n] not in the first v weights. This is the same as

$$b[i] = \sum_{j=1}^{v} \beta_j \phi_j[i] + \sum_{j=v+1}^{N} \alpha_j \phi_j[i]$$

It is clear we can recover a[n] from b[n] using M^T. The key result comes from analysis of C(b):

$$C(b) = \frac{\sum_{j=1}^{v} \beta_j^2 \lambda_j + \sum_{j=v+1}^{N} \alpha_j^2 \lambda_j}{\|a\|^2}$$

The denominator is obtained from the fact that $\|a\| = \|b\|$, because the transformation is orthogonal.

$$\|a\|^2 (C(a) - C(b)) = \sum_{j=1}^{v} \alpha_j^2 \lambda_j - \beta_j^2 \lambda_j)$$

$$\leq \sum_{j=1}^{v} (\alpha_j^2 - \beta_j^2 \lambda_v)$$

$$= (1 - \lambda_v) \sum_{j=1}^{v} \alpha_j^2 \leq (1 - \lambda_v) \|a\|^2$$

Hence $C(a) - C(b) \leq 1 - \lambda_v$, and if we choose λ_v close to 1, both sequences are equally concentrated. In this case, b[n] is no worse

than a[n] for transmission through a bandlimited channel.

But choosing λ_v close to 1 has a drawback: It requires v less than $2(W_2 - W_1)N$. We would like to scramble as much energy in a[n] as is possible, and it seems a larger v is necessary. To see this, consider

$$\|\mathcal{B}a\|^2 = \sum_{j=1}^{N} \alpha_j^2 \lambda_j = \sum_{j=1}^{v} \alpha_j^2 \lambda_j + \sum_{j=v+1}^{N} \alpha_j^2 \lambda_j$$

$$\leq \sum_{j=1}^{v} \alpha_j^2 + \lambda_v \sum_{j=v+1}^{N} \alpha_j^2 = \sum_{j=1}^{v} \alpha_j^2 + \lambda_v \left(\|a\|^2 - \sum_{j=1}^{v} \alpha_j^2\right)$$

$$= (1 - \lambda_v) \sum_{j=1}^{v} \alpha_j^2 + \lambda_v \|a\|^2$$

Hence, recalling the definition of C(a) and casting out a factor,

$$\frac{C(a) - \lambda_v}{1 - \lambda_v} \leq \frac{\sum_{j=1}^{v} \alpha_j^2}{\|a\|^2}$$

This means we should keep λ_v small to bring the energy in the first v weights λ_j close to $\|\mathcal{B}a\|^2$. This also leads to a better method for scrambling. Suppose we have two bands, one for input and one for output which we denote by $[W_1, W_2]$ and $[\bar{W}_1, \bar{W}_2]$. Now if $v = 2(W_2 - W_1)N(1 + \epsilon)$ we can scramble nearly all the energy in the input signal, since λ_v will be near zero. And if v also is equal to $2(\bar{W}_2 - \bar{W}_1)N(1 - \epsilon)$, the output signal will be very much concentrated to the desired band, since $\bar{\lambda}_v$ will be near 1.

The result is that we choose $\bar{W}_2 - \bar{W}_1 = (1 + 2\epsilon)(W_2 - W_1)$, for some

small ϵ. When we use separate bases (corresponding to the different bands), we can no longer preserve the energy in $a[n]$ not in the first v weights. Thus in this method, we have two bases ϕ_j and $\bar{\phi}_j$; we compute α and β as above, and set

$$b[i] = \sum_{j=1}^{v} \beta_j \bar{\lambda}_j[i]$$

4.4 COMPLETE SCRAMBLING SCHEME

We are ready to describe the full scrambling scheme presented by Wyner. Refer to Figure 4-1 for a block diagram. There are seven components in the full scrambler system from transmitter to receiver. We have shown above how to permute v weights; what is shown here is the analysis of a complete transmission system using the scrambler.

What we have discussed above is a mapping from R^N to R^N through some matrix M. In the complete version, the input is separated into blocks of size N, and the output is the concatenation of their scrambled versions. Hence input $a[n]$ for $-\infty < n < \infty$ to the simple scrambler produces output $b[n]$, where

$$a[kN]\ldots a[kN+N-1] \longleftrightarrow b[kN]\ldots b[kN+N-1], \quad -\infty < k < \infty$$

It is these infinite $a[n]$ and $b[n]$ which are discussed here.

The input waveform $x(t)$ is taken from the voice source, and sampled at some period T to produce $a[n]$, where $a[n] = x(nT)$. This $a[n]$, in blocks of N values, is scrambled using the scheme above. The output of the scrambler, $b[n]$, is modulated. We use a reconstructing filter $g_0(t)$ which is restricted to the desired band, to produce

$$y(t) = \sum_{n=-\infty}^{\lfloor t/T \rfloor} b[n]\, g_0(t-nT)$$

The signal $y(t)$ is ready for transmission through a channel. The channel will have some frequency response $H_c(w)$, so that the output is

$$Z_c(f) = H_c(f)\, Y(f)$$

where $Z_c(f)$ is the inverse Fourier transform of $z_c(t)$, and $Y(f)$ that of $y(t)$. The channel also has some random stationary noise $u(t)$; let $z(t) = z_c(t) + u(t)$ be the channel output.

Now it's the receiver's turn. A sampler produces values (impulses) at intervals of time T, and these are passed through a tapped delay line or equalizer of length $2K + 1$ to compensate for the effects of the channel and modulator. Call the output of the equalizer $b[n]$. Note that $C(w)$ is the frequency spectrum of the discrete-time equalizer.

Finally, $b[n]$ is unscrambled and yields $a[n]$, the receiver's estimate of the original sequence. A desampler allows the final output

$$x(t) = \sum_{n=-\infty}^{\lfloor t/T \rfloor} a[n]\, g_1(t - nT)$$

We examine the effects of the various components on the expected error between $x(t)$ and $x(t)$. Define P_{av}, the average power of the transmitted signal $y(t)$, to be

$$P_{av} = \lim_{\tau \to \infty} E\left[\frac{1}{2\tau} \int_{-\tau}^{\tau} y^2(t)\,dt \right]$$

and the mean-squared error ϵ^2

$$\epsilon^2 = \lim_{\tau \to \infty} E\left[\frac{1}{2\tau} \int_{-\tau}^{\tau} [x(t) - x(t)]^2\, dt \right]$$

Wyner proves two theorems about the upper bounds on these quantities. The first theorem bounds P_{av} based on the input energy. A typical voice signal can be characterized as a stationary random process with power spectrum $P_x(f)$. Let the expected energy

$$\sigma_x^2 = E[\, x^2(t)\,] = \int_{F_1 \le |f| \le F_2} P_x(f)\, df$$

The theorem is

$$P_{av} \le \sigma_x^2 \left[\frac{N}{v\,T} \int_{F_1 \le |f| \le F_2} G_0(f)^2\, df + A_1 \frac{1}{v} \sum_{j=1}^{v} (1 - \bar{\lambda}_j) \right]$$

$G_0(f)$ is the frequency spectrum of the modulator above. N, T, v, F_1, and F_2 are the system parameters, and $\bar{\lambda}_j$ are the eigenvalues for the basis on N and $[W_1, W_2]$. (F_1 and F_2 are the "real" frequencies desired, whereas W_1 and W_2 are those in the discrete version. They are related by $W = FT$.) $T^2 A_1$ is the largest magnitude in G_0 outside $[F_1, F_2]$, or

$$A_1 = \sup_{|f| \notin [F_1, F_2]} \frac{|G_0(f)|^2}{T^2}$$

The term summing the $(1 - \lambda_j)$ is small, according to the way we chose v above, as all λ_j are close to 1.

The second theorem bounds ϵ^2 as the sum of errors from noise, the channel, and the scrambling itself:

$$\epsilon^2 = \epsilon_n^2 + \epsilon_c^2 + \epsilon_s^2$$

$$\epsilon_n^2 \le \int_{\bar{F}_1 \le |f| \le \bar{F}_2} C(fT)^2\, P_u(f)\, df + A_2 \frac{1}{v} \sum_{j=1}^{v} (1 - \bar{\lambda}_j)$$

where $P_u(f)$ is the power spectrum of the stationary noise u(t). The

second term, again, is small; the noise error is no more than the power of the noise in the desired band, as modified by the equalizer.

$$\epsilon_c^2 \leq \sigma_x^2 \left[T \int_{F_1 \leq f \leq F_2} \left| \frac{1}{T} C(fT) H_c(f) G_0(f) - 1 \right| df + A_3 \frac{1}{N} \sum_{j=1}^{v} (1-\lambda_j) \right]$$

Here, again, the second addend is small; A_3 depends on the energy outside $[\bar{F}_1, \bar{F}_2]$. It is desirable to make a very accurate delay line, so that the integrand is near 0. Of course, the effect of that term is not related to the scrambler, and would introduce the same error for typical speech as for encrypted speech.

$$\epsilon_s^2 \leq \left[\max_f \frac{P_x(f)}{T} \right] \frac{1}{N} \sum_{j=v+1}^{N} \lambda_j$$

Finally, the error implicit in the scrambler using two bases corresponds to the energy omitted in the input transformation. This is bounded by the power spectrum of the input. These errors, as revealed by simulated results, are discussed in section 3.2.

4.5 SELECTION OF RANDOM MATRICES

The key property of the matrices used for scrambling in Wyner's method is their orthogonality. This property is necessary to avoid expanding errors during the decryption. Theoretically, the matrix used should have no effect on the system's performance, as long as it is orthogonal. But certain matrices are clearly unsuitable from a security standpoint. We look at the generation and properties of the matrices here.

Wyner shows that if the scrambling matrices M_k, $-\infty < k < \infty$, are

chosen from a uniform distribution on the set of all orthogonal matrices then the scrambled weights β_j have no correlation from one block to the next. Hence matrices with uniform Haar measure provide perfect security, because an adversary is unable to gain any information from the transmitted signal about the exact form of the original speech.

The time required to generate these matrices can be quite large -- say $O(n^3)$ for Gram-Schmidt orthogonalization -- and a more desirable method is to select from a set of precomputed matrices.

A method based on Hadamard matrices is discussed in [Sloane]. These matrices are said to exist for n = 1, 2, or a multiple of 4, and each entry of such a matrix H_n is $1/\sqrt{n}$ or $-1/\sqrt{n}$. There is also an algorithm known as the fast Hadamard transform which allows multiplication of a vector by H_n in time proportional to n lg n.

The suggested use of Hadamard matrices is within permutations and diagonal matrices, that is, defining

$$M = D_1 \, P \, H_n \, Q \, D_2$$

where P and Q are arbitrary permutation matrices, and D_1 and D_2 are 1 or -1 on the diagonal, and 0 elsewhere. These products can be precomputed and saved; when one is needed, it is selected at random from the set.

Let S_n be the set of all such matrices. Sloane shows the following about the security of scrambling with matrices in S_n:

- The covering radius of S_n is $\sqrt{2n(1 - 1/\sqrt{n})}$. A covering radius is a measure of how well a set covers the orthogonal group O(n). A small covering radius is more secure.
- The deep holes in S_n -- the matrices in O(n) furthest from S_n --

are the monomial matrices. These matrices have exactly one non-zero element in each column, and the element is 1 or -1.

The "tighter" S_n is, then, the less likely it is that a matrix will become a deep hole. A larger value of n, as expected, increases security, in that degenerate cases are unlikely. One other observation is that S_n is very large:

$$S_n = \frac{2^{2n}(n!)^2}{(n-1)(n^2-2n-2)}$$

when n-1 is a prime greater than or equal to 19, and the prime is of the form 4a-1.

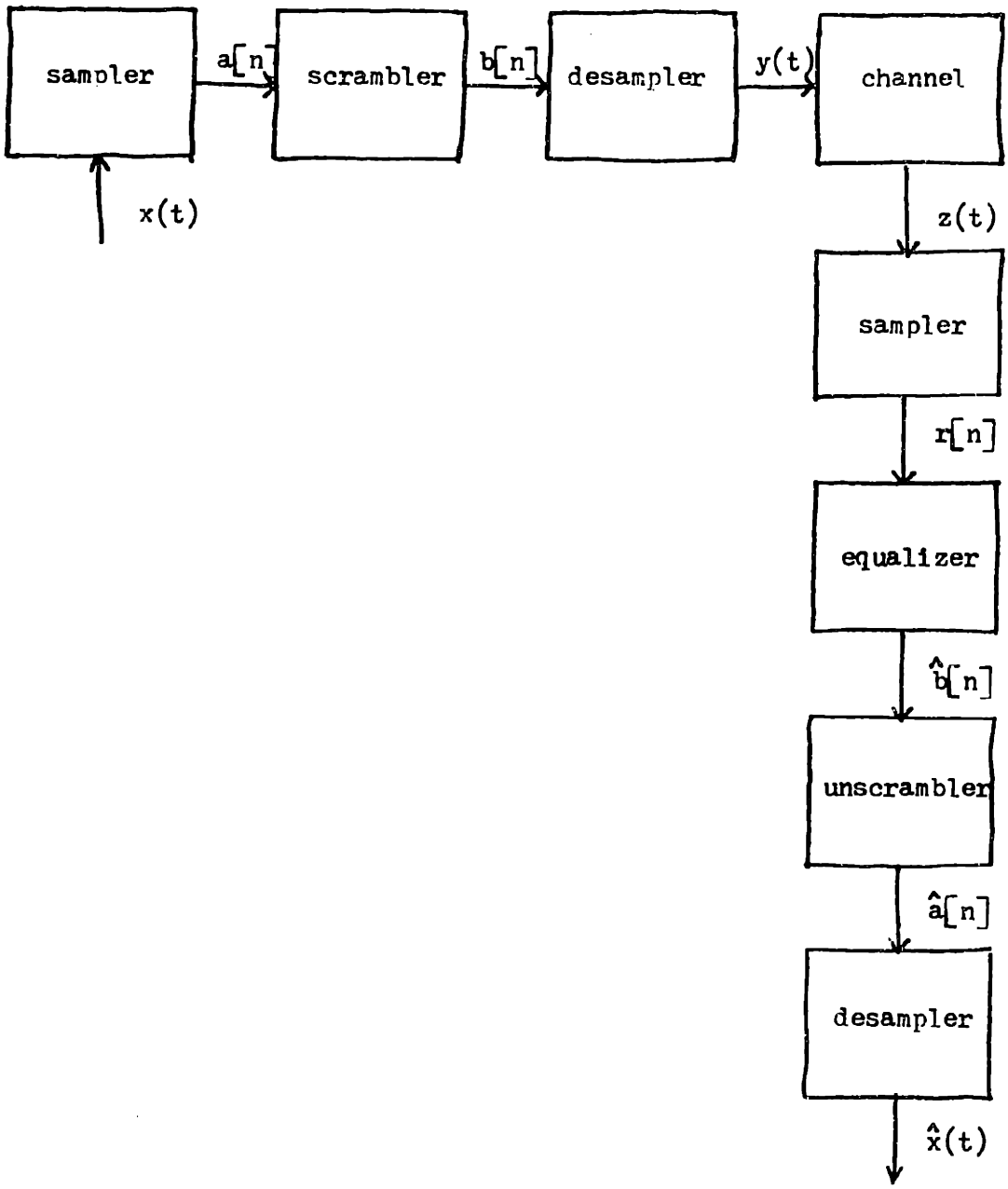


Figure 4-1 Block Diagram of Scrambler

SECTION 5 -- THE SOFTWARE

Software simulation of Wyner's model requires about 40 pages of programs, given in appendix A. The organization of the programs, and the assumptions underlying the simulation, are discussed here. The simulation is performed on a Lisp machine, useful because of its graphical output and debugging facilities. The Lisp language also provides a clever system of object-oriented programming used to build a model of the scrambler.

The basic components are presented in section 5.1, with an introduction to the concepts of flavors in Lisp. The following section describes the components of the block diagram as they are simulated. Three of these components -- the desampler, channel, and equalizer -- are discussed in greater detail in section 5.3. The simulation of channel noise, a somewhat tricky phenomenon, is explained in section 5.4. The last section reviews the major algorithms used in various parts of the simulation.

5.1 SOFTWARE COMPONENTS

An ideal test of the scrambler would be provided by an actual hardware implementation, using a real channel and measuring real errors in speech quality. Since the purpose of this thesis is to find the required complexity of such a hardware implementation, software is used instead to model real operation.

M.I.T. has many Lisp machines, ideal tools for building a model of the scrambler. The use of Lisp is a drawback for the required mathematical operations, but this drawback is easily outweighed by the debugging system and graphical capabilities of the machine.

In addition, Lisp data abstraction simplifies the representation of the scrambler and its components. Objects called flavors provide a means of message-passing and object-oriented programming. These objects maintain states called instance variables and operate on those variables through messages, or methods.

One basic data type of the scrambler is the discrete-signal. It has five instance variables:

- o time-values, an array containing the discrete samples;
- o time-base, the time at which the first sam le is taken;
- o time-increment, the sampling period;
- o freq-values, an array containing the discrete inverse Fourier transform of the time values; and
- o sample-size, the number of samples.

A discrete signal corresponds to a finite segment of a voice signal transferred among the various components of the scrambler. It can also

approximate a continuous signal, when the time increment is sufficiently small.

The methods for the discrete-signal objects include, among the obvious ones,

sample period

bandlimit low high

indexlimit low high

convolve signal2

Note that the methods have arguments, which are also passed as part of a message to the object. These and other methods are described in full in appendix A, which contains the programs.

The other basic data type is the matrix, which is used for the transforms to and from the prolate spheroidal basis, and the scrambling of weights in that basis. It has, of course, the instance variables

- rows
- columns
- elements

and the operations

- multiply matrix2
- eigenvectors-and-eigenvalues

The second operation above is relevant only for square matrices in this system; in fact, it is used only for real symmetric matrices. The flavor system allows mixing (hence the name flavors) of operations and of data types. A real-symmetric-matrix is a matrix with a few more methods.

Much of the software is directed toward management of the data types listed above. The other feature of flavors -- their use in message passing -- is exploited in the functions which drive the model.

5.2 BLOCK DIAGRAM OF THE SCRAMBLER

A data object is defined for each component in the block diagram of the scrambler, explained in section 4.4. These objects have two operations, namely

- o process signal
- o connect module

The process operation performs appropriate functions on the input signal -- scrambling, desampling, adding noise, etc., then passes the output signal to other modules. The connect operation adds a module to the list of successors to which output is sent. A signal is sent by passing a process message, with the signal, to a module.

This system creates a dataflow-like network. The operation is sequential, but the message passing allows the components to operate independently. A driver routine connects the modules, and enters a loop from which it sends the initial module a series of input signals. These signals correspond to blocks of samples of speech.

The following modules are used:

- o sampler -- samples signal at specified rate;
- o scrambler -- scrambles values in signal and outputs scrambled signal;
- o unscrambler -- reverses scrambling;

- o comparator -- prints energy of error between most recently received pair of signals;
- o display -- plots values of signal on screen;
- o desampler -- outputs "continuous" signal derived from an impulse response applied to two most recent signals;
- o channel -- applies typical channel impulse response to recent signals, and adds noise;
- o delay -- introduces a wait between receiving a signal and sending it to another module;
- o equalizer -- applies equalizer impulse response to simulate tapped delay line.

One more module synchronizes the operation of the scrambler and unscrambler:

- o random-selector -- chooses a random orthogonal matrix.

It is intended the random-selector will pass the matrix to the scrambler and unscrambler. The complete diagram of the scrambler as it is used for simulation is found in the documentation for the programs, in appendix A.

The display module enhances testing of the scrambler, using the graphics of the Lisp machine to produce a plot of the time sequence a signal represents. It is connected to the input and output parts of the block diagram to provide a visual method of inspection of the signals, in addition to the comparator's quantitative error measure.

The driver routine starts the data flow by reading a set of voice samples from a file and passing them to the input samples. These voice

samples are stored at a rate of 16 kHz, and the input sampler converts them to a slower rate. The signals move through the network until they reach the display and comparator modules, which do not send them any further; at this point, the driver sends another set of samples.

The operation of the scrambler, sampler, random-selector, and comparator are relatively straightforward. The other modules -- desampler, channel, and equalizer -- require more careful implementation and are discussed below.

5.3 DESAMPLER, CHANNEL, AND EQUALIZER

Consider first the model of a desampler. It transforms a sequence of time samples into a "continuous" waveform limited to some band, say $[F_1, F_2]$. (The actual model uses a slightly wider band, for reasons explained in section 3.3.) One simple method which comes to mind is the following:

1. Convert the input sequence, of size N, to one of size 8N, by putting 7 zeroes between each sample.
2. Compute the discrete Fourier transform and set to zero the values at those frequencies outside the desired band.

This method does not work, for reasons much like those explaining why sinusoids cannot be used as the basis for the scrambler. One cannot perform operations like these "in place," because the discrete Fourier transform assumes the time sequence is repeated over and over again; it has no provision for non-periodic sequences.

We would like to avoid, however, explicit convolution with an

impulse response, as that method is slow, running in time $O(N^2)$. The discrete Fourier transform can perform a circular convolution in time $O(N \lg N)$. We can avoid the circularity by a simple trick.

The idea is to represent the impulse response as 16N points, where the second 8N are all 0. Then any circular convolution with an input sequence of 16N points will produce output which is correct for the second 8N points. In other words, let $h[n]$ be the impulse response and $x[n]$ the input. We have, in circular convolution of 16N-point samples,

$$y[n] = \sum_{i=0}^{n} x[i]\, h[n-i] + \sum_{i=n+1}^{16N-1} x[i]\, h[n-i+16N]$$

Clearly this dependence on values of $x[i]$ for $i < n$ is undesirable in a model of a causal system. But since $h[i] = 0$ for $i \geq 8N$, we have,

$$y[n] = \sum_{i=n-8N+1}^{n} x[i]\, h[n-i] \qquad \text{if } n \leq 8N-1$$

$$y[0] = \sum_{i=0}^{n} x[i]\, h[n-i] + \sum_{i=n+8N+1}^{16N-1} x[i]\, h[16N-i-n] \qquad \text{otherwise}$$

The values $y[i]$ for $8N-1 \leq i \leq 16N-1$ are dependent only on those values of previous $x[i]$. Other values of $y[i]$ we can discard, and the correct $y[i]$ values become the output of the desampler. In fact, it is this method we use for each of the modules discussed in this section.

The desampler expands an input into 8N values as described in step 1 of the naive algorithm. These 8N values are combined with the previous 8N, and the resulting 16N are convolved with the $h[n]$. The upper half of the output is the result.

The channel, meanwhile, has an input of size 8N already (from the desampler). It does not require expansion. Finally, the equalizer operates on the output of the sampler, and performs a convolution of size 8N -- a long response is necessary to compensate for the other two components.

We next discuss the creation of the impulse responses for the modules above. The desampler is quite simple -- the 8N relevant points are those surrounding time 0 in an "ideal" filter, shifted by 4N. Here an impulse at time 4N is bandlimited to the desired $[F_1, F_2]$.

One may wonder if 8N points are sufficient to represent the entire impulse response with little error. Recall the impulse response of an ideal filter is

$$h(t) = \frac{1}{\pi t} (\sin 2\pi F_2 t - \sin 2\pi F_1 t)$$

Suppose F_1 is about 300 Hz and F_2 is about 3000 Hz. Assume also that a sample is taken every 100 us, and N is 32 (or, each continuous block 256 points). Then the value at 800 us is the first not included when approximating the impulse response. We have

$$h(0) = 2(F_2 - F_1) = 5400.$$

$$h(.0008) = -163.$$

$$h(.0016) = -214.$$

Hence the energy at the center of the response is 1000 times that at the cutoff, and 640 times that twice as far away. Approximating the response is reasonably accurate, according to these calculations.

The impulse response of the channel is derived in a similar way,

except that a real filter is applied to the impulse at 4N. The values of the real filter are taken from a graph in [Rabiner]. Since the majority of the 16N-point frequency domain is outside $[F_1, F_2]$, only a few values of the real filter can be used. For example, if N = 32 and the fundamental frequency is 312.5 Hz, then only 18 of the 512 points in the continuous representation are non-zero.

These points are very rough. The typical telephone channel frequency response is smooth when plotted in decibels, but such smoothness is lost when the decibels are converted to magnitudes. See graphs 5-2 and 5-3 for a comparison. The other function of the channel module, to add noise to a signal, is discussed in the next section.

The equalizer, finally, compensates for the effects of the desampler and the channel, excluding noise. Its impulse response is the inverse of the combined impulse response of the desampler and the channel. The frequency response of the equalizer is computed by finding the impulse response required to convert the combined response above to an "ideal" filter over $[F_1, F_2]$. The required response is long, and includes a delay of 3N points (that is, three blocks), to provide the most accurate output. The equalizer, since it follows the second sampler in the block diagram, operates in sampled form, not continuous form as the other modules above.

5.4 NOISE SIMULATION

Noise in an actual transmission line is a phenomenon which is very

difficult to measure. Comparing the average power of noise to that of the intended signal is not normally useful. [Bell]

Indeed, noise level and voice level are measured in different units to avoid their comparison. Voice levels are measured on a qualitative scale of volume units, or vu; noise is measured in dBrnc, with respect to a specific noise threshold.

The average power P_{av} of a voice signal with volume level V depends on the channel load T_L, that is, on the percentage of use of the channel being measured. This relationship holds:

$$P_{av} = V - 1.4 + \log_{10} T_L$$

A typical volume level is between -14 and -25 vu, with typical load of .25. Hence P_{av} is generally between -12.4 and -23.4 dBm. (A dBm is defined such that 1 mW is 0 dBm.)

Noise is measured in dBrnc, where 0 dBrnc is -90 dBm, and the measurement is scaled with respect to something called "C-weighting." Understanding of a voice signal is related to noise level as follows:

Understanding	Noise Level
excellent	29.5 dBrnc
good	39.0
fair	48.0
poor	55.5

Excellent understanding is possible, then, when noise is at -60.5 dBm, or 37.1 dB less than the lowest voice level, in this simplified analysis. Poor understanding occurs at only 11.1 dB less than the voice level.

Two observations confirm these noise estimates. A typical stereo casette deck has separation between left and right channels of "better than 30 dB at 1 kHz" and signal to noise ratio of "57 dB without Dolby." A Lincoln Laboratories report [Holsinger] estimates low-level noise in a telephone channel is between 20 and 50 dB less than the signal energy.

Since, as shown below, the variance of a random variable with a normal distribution is also its expected energy, we need only to multiply a random variable by the square root of the desired noise level and divide this by the square root of the sample time, to determine the amplitude of the noise to be added in the channel. The resulting value, when squared and multiplied by the sample period to compute its energy, has the desired noise level.

5.5 ALGORITHMS

Four algorithms are central to the implementation of the scrambler. To generate random orthogonal matrices we need, of course, a random number generator and a method to make a matrix orthogonal. To compute the discrete prolate spheroidal basis, we must be able to find eigenvectors. And to switch between time and frequency representations of a signal, we need a Fourier transform algorithm.

The FFT algorithm used is an iterative method adapted from [Sedgewick]. It operates only on sets of data whose lengths are powers of 2. (Another algorithm is included for lengths not powers of 2, but it is much slower.) The algorithm relies on complex number operations. Some Lisp machines support them, and others do not; the scrambler has its own.

The algorithm is basically in three parts:

1. Reordering the data in bit-reversed fashion.

2. Performing successive FFT passes, in place, over the data.

3. Scaling the results if the transform is from time to frequency.

Running time is about 1.5 n lg n, in ms. This is about 15 seconds for a 1024-point FFT, and 3 seconds for a 256-point FFT.

We turn next to generation of random numbers. The Lisp machine provides a uniform generator over the range [0, 1). What is needed for random orthogonal matrices and for noise is a pseudo-random value with normal distribution, mean 0, and variance 1.

There are several good generators of this type [Sloane]. We choose the oldest and simplest, because it is easy to implement, and because an exact normal distribution is not necessary. The main purpose of the random matrix is to scramble the signals, but the choice of the matrix has little effect on the performance of the system.

The algorithm used is straightforward:

1. Compute uniformly distributed random numbers U_1, ..., U_{12}.

2. Return $U_1 + \ldots + U_{12} - 6$.

The computation of the actual distribution produced by this algorithm is complicated, but experiment shows the distribution is roughly normal (graph 5-1).

Now a word on normal distributions and their energy contents. The distribution p(x) with mean 0 and variance v can be expressed as

$$p(x) = \frac{1}{2\pi v} e^{-x^2/2v}$$

where

$$v = \int_{-\infty}^{\infty} x^2 \, p(x) \, dx$$

Note that v, the variance, is also the expected energy $E[x^2]$.

These random numbers can either be used to add noise to a transmitted signal or to create a random orthogonal matrix. When they are used to make a matrix, the matrix must be orthogonalized, and the Gram-Schmidt algorithm from [Strang] is used.

This algorithm is iterative on a matrix A of size n x n, making columns 1, ..., n orthogonal and of length 1. For each column i, the components of columns i + 1, ..., n are removed. Denote column i by c_i. Then

$$c_i' := \frac{c_i - \sum_{j=i}^{n} \langle c_i, c_j \rangle c_j}{\left\| c_i - \sum_{j=i}^{n} \langle c_i, c_j \rangle c_j \right\|}$$

The contribution of c_j in c_i is their inner-product. For only those columns following c_i are the components removed, because we have already made previous columns orthogonal to c_i.

The adjusted c_i is then divided by its length so that it becomes orthonormal. When this process terminates, the matrix is orthonormal. The generation of such a matrix requires n^2 operations to produce the initial entries, and n^3 for the orthogonalization. Since the matrices are generated infrequently, such expense is negligible. Other methods in which matrices are generated during scrambling may also be used; one such method is discussed in section 4.5.

Finally, we adapt an algorithm in [Goldstein] based on Jacobi's method for computing the eigenvalues and eigenvectors of a real symmetric matrix A. The algorithm is iterative, and it computes the desired quantities by eliminating off-diagonal elements of A. The algorithm terminates when the sum of the squares of the diagonal elements does not change much between successive passes. The diagonal values are then eigenvalues.

Let R_{ij} be a rotation to eliminate some element A_{ij}. Since A is symmetric, R_{ij}^T can eliminate A_{ji}. So both are removed by calculating

$$A' := R_{ij} A R_{ij}^T$$

In each pass, all A_{ij}, $i \quad j$, greater than a small value are eliminated by rotation. The matrices R_{ij} are used also to compute the eigenvectors as follows:

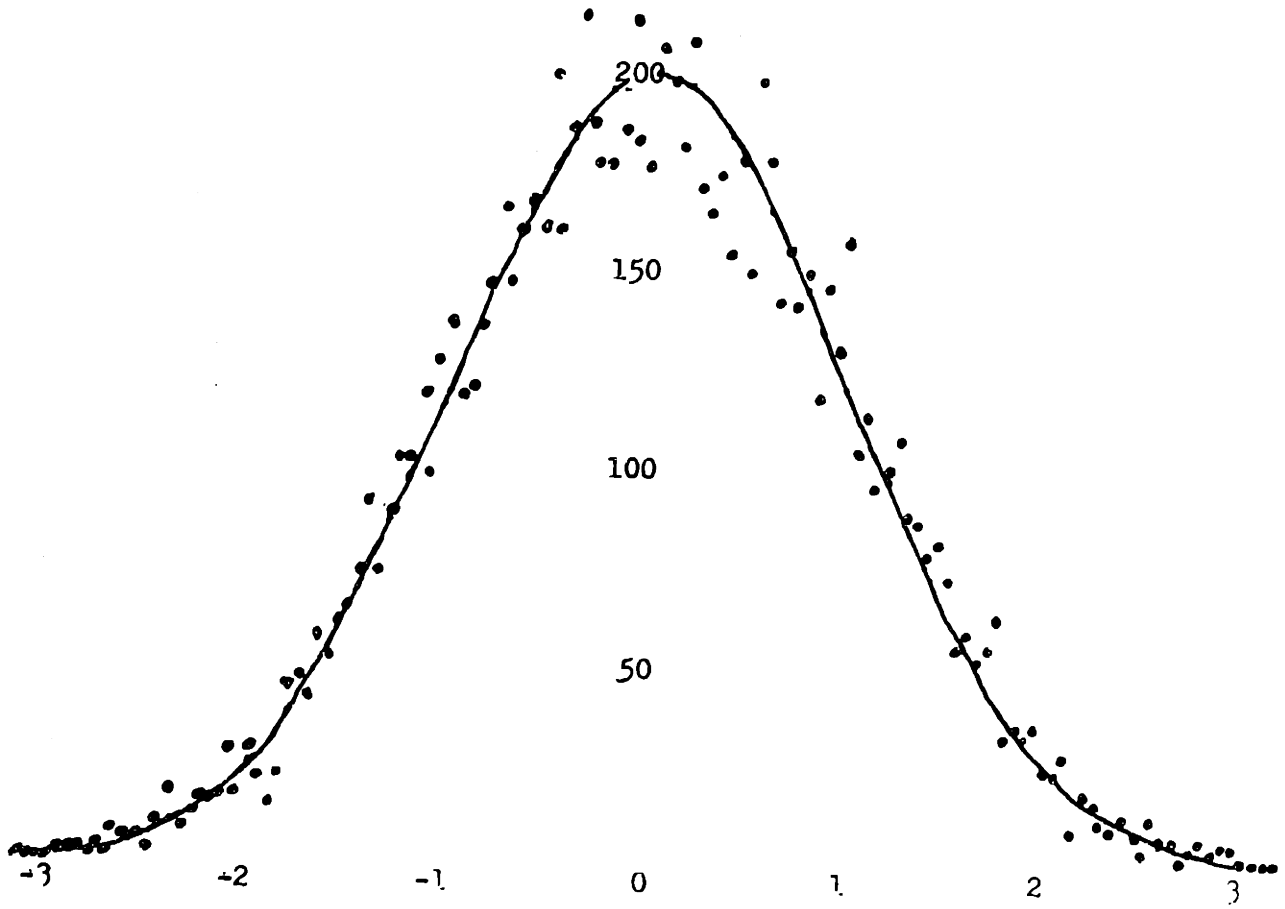
$$E' := R_{ij} E$$

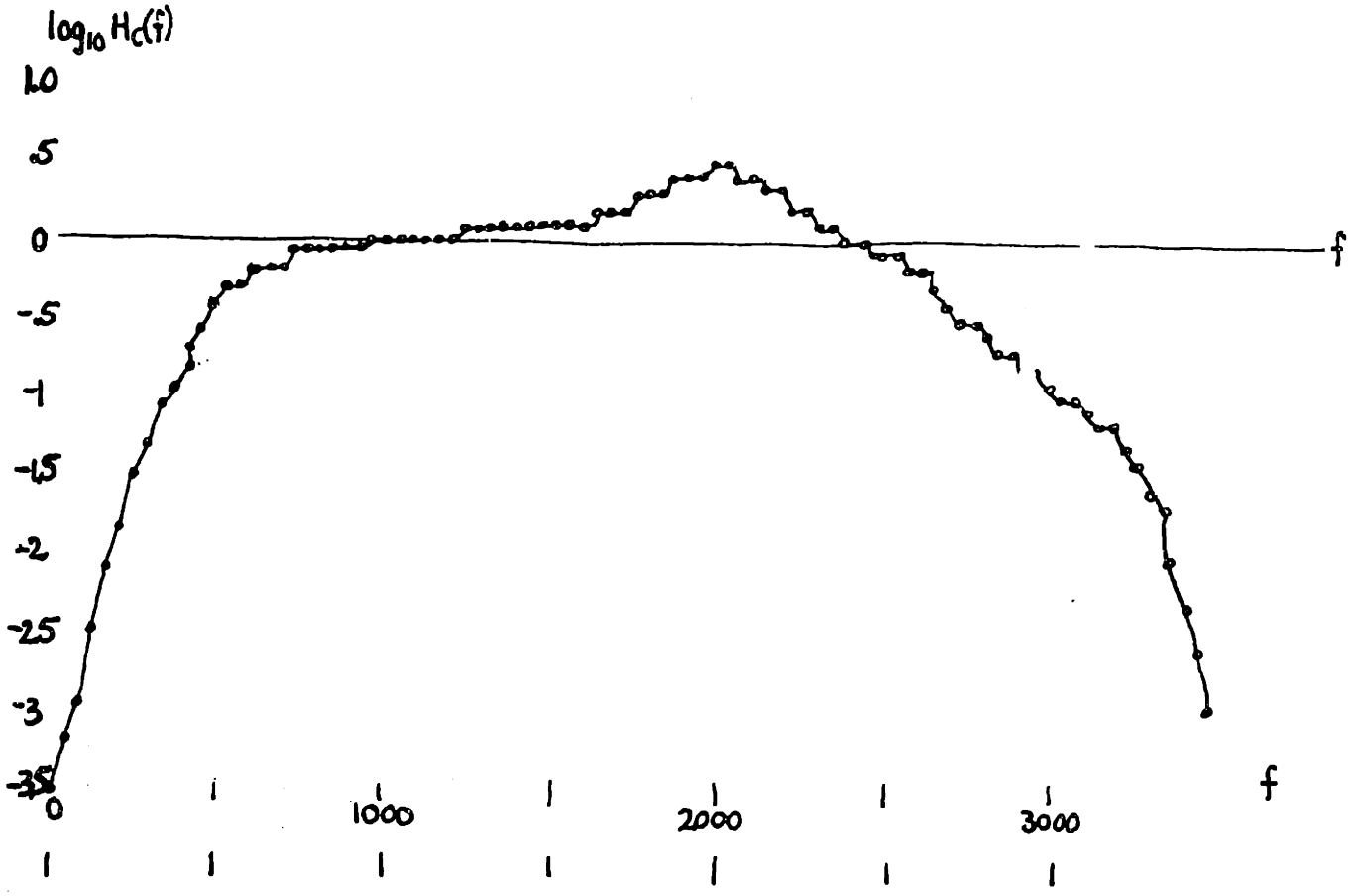
where E is initialized to the identity matrix. The final E will contain the eigenvectors. Column i of E will hold the eigenvector corresponding to eigenvalue A_{ii}.

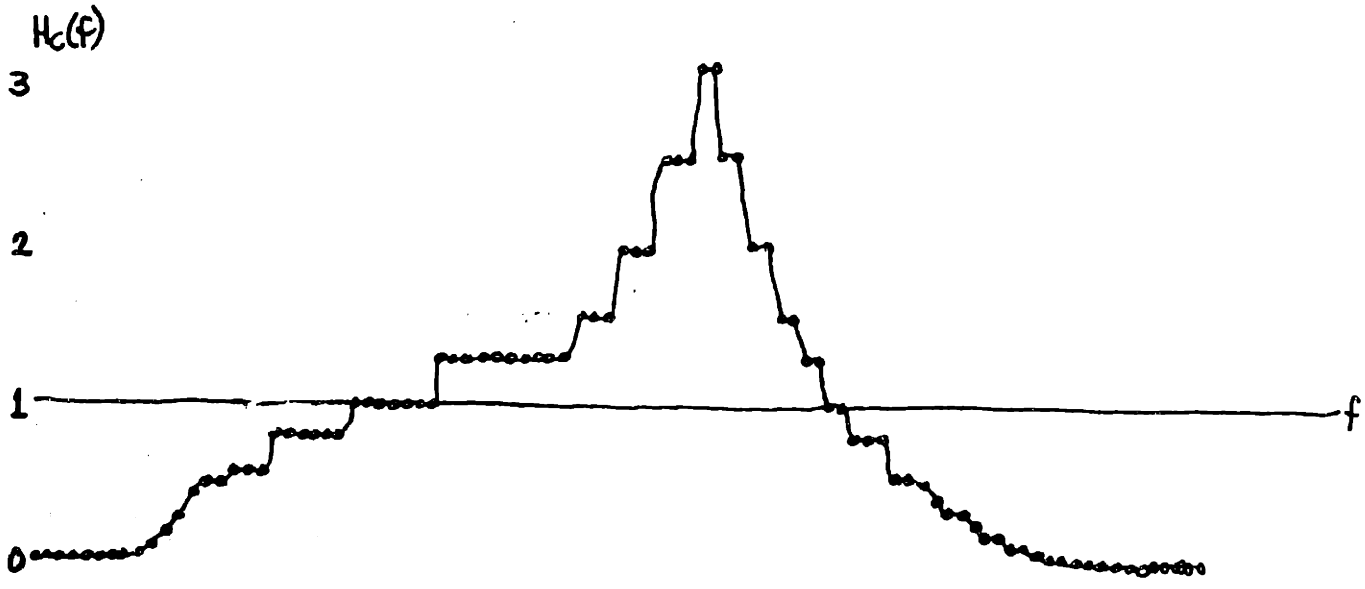
Graph 5-1

Expected normal distribution for 10,000 samples at .25 intervals vs. actual distribution from pseudo-random generator

Mean = 0, variance = 1







Graphs 5-2 and 5-3 Comparison of log-magnitude and amplitude plots for simulated channel filter.

SECTION 6 -- CONCLUSION

The scrambler appears to meet its requirements, to the extent of producing a poor but intelligible output signal. Its security is sufficient; its operating speed is reasonable. But the results presented here are far from conclusive, for there are other issues at hand.

A complete hardware implementation of the scrambler is needed to verify that its simulated properties hold in real systems with nonlinear distortion, and that complicated forms of modulation do not adversely affect the quality of the scrambled signal. Telephone transmission lines are designed for speech, not scrambled, but bandlimited, signals -- will they operate as expected?

Modern integrated circuit technology will make the digital part of the scrambler fast enough for poor or fair output at a low price. In the near future, high quality output will also be possible, as processor speeds increase. But at the same time, digital encryption and transmission will become more accurate, and may compete against analog forms of encryption on any potential market.

Wyner's method is a clever solution to a theoretical problem. With some further research, it could be determined whether the scrambler solves a practical problem as well.

APPENDIX A -- PROGRAMS

The following programs are written in Zetalisp, and will run on either an M.I.T. or a Symbolics Lisp Machine. Ther are four files, including

- o the matrix operations;
- o the discrete signal operations;
- o system utilities; and
- o a block diagram for the scrambler.

This version of programs is not the same as that actually used for simulation, and it may include minor bugs. The real versions contain few comments; the ones printed were documented on another computer system. Minor reorganization of code may have introduced some errors.

```
; -*- Mode:LISP; Fonts:CPTFONT; Base:10 -*-

; Software for simulation of Aaron Wyner's analog encryption scheme.
; Written by Burt S. Kaliski as part of an undergraduate thesis
; project.
;
; Massachusetts Institute of Technology, May 1984.
;
;
; ------------------------ Utility functions ----------------------
;
; (top x)                 returns the least integer greater than or equal
;                         to x
; (bottom x)              returns the greatest integer less than or equal
;                         to x
; (find-char s i)         returns character i (at least 1) of string s
; (exp10 x)               returns 10 ** x
; (array-size values)     returns the length of an array
; (array-dim values n)    returns dimension n (at least 0) of an array
; (gamma n w1 w2)         evaluates the impulse response of an ideal
;                         filter over [w1, w2] at time n
; (normal-random)         returns a random number taken from a normal
;                         distribution with mean 0 and variance 1

(defun top (x)
  (let ((trunc (fix x)))
    (if (= trunc x) trunc (1+ trunc))))

(defun bottom (x)
  (fix x))

(defun find-char (str i)
  (character (substring str i (1+ i))))

(defun exp10 (x)
  (^ 10. x))

(defun array-size (values)
  (array-#-dims values))

(defun array-dim (values n)
  (array-dimension-n (1+ n) values))

(defun gamma (n w1 w2)
  (let ((pi-sum (* 3.1415926 (+ w2 w1)))
        (pi-diff (* 3.1415926 (- w2 w1))))
    (* .6366197835 (cos (* pi-sum n))
       (if (= n 0) pi-diff (// (sin (* pi-diff n)) n)))))

(defun normal-random ()
  (do ((i 0 (1+ i))
       (value 0.0 (+ value (si:random-in-range 0.0 1.0))))
      ((= i 12.) (- value 6.0))
    nil))


; --------------------- Complex number operations -------------------
;
; The typical complex number constructors, selectors, and operators
; are supported. Max-fun-complex computes the maximum value of an
```

```
; arbitary function of each element over an array of complex numbers.
;
; Complex numbers are represented as flonums (indicating a zero
; imaginary part), or as dotted pairs of flonums.

(defun make-complex (a b)
  (cons a b))

(defun make-polar (angle)
  (make-complex (cos angle) (sin angle)))

(defun add-complex (z1 z2)
  (make-complex (+ (real-part z1) (real-part z2))
                (+ (imag-part z1) (imag-part z2))))

(defun sub-complex (z1 z2)
  (make-complex (- (real-part z1) (real-part z2))
                (- (imag-part z1) (imag-part z2))))

(defun mult-complex (z1 z2)
  (make-complex (- (* (real-part z1) (real-part z2))
                   (* (imag-part z1) (imag-part z2)))
                (+ (* (real-part z1) (imag-part z2))
                   (* (imag-part z1) (real-part z2)))))

(defun div-complex (z1 z2)
  (let ((scale (float (+ (^ (real-part z2) 2)
                         (^ (imag-part z2) 2))))
        (real-sum (+ (* (real-part z1) (real-part z2))
                     (* (imag-part z1) (imag-part z2))))
        (imag-sum (- (* (imag-part z1) (real-part z2))
                     (* (real-part z1) (imag-part z2)))))
    (cond ((= scale 0.0) (make-complex real-sum imag-sum))
          (T (make-complex (// real-sum scale) (// imag-sum scale))))))

(defun real-part (z)
  (if (numberp z) z (car z)))

(defun imag-part (z)
  (if (numberp z) 0. (cdr z)))

(defun abs-complex (z)
  (let ((r (abs (real-part z)))
        (i (abs (imag-part z))))
    (cond ((< r 1.e-15) i)
          ((< i 1.e-15) r)
          (T (sqrt (+ (^ r 2) (^ i 2)))))))

(defun max-fun-complex (z-array fun)
  (let ((size (array-active-length z-array)))
    (do ((i 0 (1+ i))
         (m 0 (max (abs (apply fun (list (aref z-array i)))) m)))
        ((= i size) m)
      nil)))

; ---------------- Discrete prolate spheroidal basis ----------------
;
; The following functions are used to generate a discrete prolate
; spheroidal basis. A DPSS is defined by three parameters: w1, w2, and
; n, where [w1, w2] is the region of [.0, .5] on which part of the
```

```
; basis is concentrated, and n is the number of points in the basis.

(defun make-prolate-spheroidal-basis (n nu f1 f2 period)

  ; Returns the basis defined for f1 * period, f2 * period, and n,
  ; where nu is the number of vectors in the basis returned. The basis
  ; is represented as an n x nu matrix.

  (send (make-gamma-filter-matrix n (* f1 period) (* f2 period))
        ':eigenvectors-ordered nu))

; ---------------------- Signal constructors ----------------------

(defun make-discrete-signal (time-values time-base time-increment)

  ; Returns a signal object whose time representation is taken from
  ; time-values, with base and increment set to the corresponding
  ; arguments.
  ; Requires that time-values has a fill pointer.

  (let* ((sample-size (array-length time-values))
         (freq-values (make-array sample-size ':fill-pointer 0
                                  ':initial-value 0.0)))
    (make-instance 'discrete-signal
                   ':sample-size sample-size
                   ':time-values time-values
                   ':time-base time-base
                   ':time-increment time-increment
                   ':freq-values freq-values)))

(defun make-discrete-filter (freq-values time-base time-increment)

  ; Returns a signal object whose frequency representation is taken
  ; from freq-values, with time base and increment set to the
  ; corresponding arguments.
  ; Requires that freq-values has a fill pointer.

  (let* ((sample-size (array-length freq-values))
         (time-values (make-array sample-size ':fill-pointer 0
                                  ':initial-value 0.0)))
    (make-instance 'discrete-signal
                   ':sample-size sample-size
                   ':time-values time-values
                   ':time-base time-base
                   ':time-increment time-increment
                   ':freq-values freq-values)))

; ---------------------- Matrix constructors ----------------------
;
; (make-matrix elements)              returns a matrix object with size
;                                     on the array elements, which may
;                                     have one or two dimensions
; (make-symmetric-matrix elements)    returns a symmetric-matrix object;
;                                     elements must have two equal
;                                     dimensions
; (make-identity-matrix n)            returns an n x n identity matrix
; (make-gamma-filter-matrix n w1 w2)  returns the n  x n matrix used for
;                                     making the DPSS of w1, w2, n
; (make-random-orthogonal-matrix n)   returns a randomly-generated,
;                                     orthogonal, n x n matrix
```

```
    ; Make column i orthogonal: Subtract components of columns 0..i-1,
    ; then normalize

    (do ((i 0 (1+ i)))
        ((= i n) (make-matrix elements))
      (do ((k 0 (1+ k)))
          ((= k n) T)
        (aset 0.0 sum k))

      ; Compute dot product of columns i and J; accumulate in sum

      (do ((J 0 (1+ J)))
          ((= J i) T)
        (let ((dot
               (do ((k 0 (1+ k))
                    (dot 0.0 (+ dot
                                (* (aref elements k i)
                                   (aref elements k J)))))
                   ((= k n) dot)
                 nil)))
          (do ((k 0 (1+ k)))
              ((= k n) T)
            (aset (+ (* dot (aref elements k J))
                     (aref sum k)) sum k))))

      ; Subtract sum from column i and normalize

      (let ((length
             (do ((k 0 (1+ k))
                  (length 0.0 (+ length
                                 (^ (aref elements k i) 2))))
                 ((= k n) (sqrt length))
               nil)))
        (do ((k 0 (1+ k)))
            ((= k n) T)
          (aset (// (aref elements k i) length) elements k i))))))

; -------------------- Random orthogonal matrices --------------------
;
; A scrambling matrix for a sample is selected from a set of
; previously-generated random orthogonal matrices.
;
; (make-random-orthogonal-matrices n count) returns a set of count
;                                           n x n such matrices
;
; (choose-random-orthogonal-matrix m) selects a single such matrix
;                                     from the set m

(defun make-random-orthogonal-matrices (n count)
  (let ((matrices (make-array count)))
    (do ((i 0 (1+ i)))
        ((= i count) matrices)
      (aset (make-random-orthogonal-matrix n) matrices i))))

(defun choose-random-orthogonal-matrix (matrices)
  (aref matrices (random (array-length matrices))))

; -------------------- Loading and dumping objects --------------------
;
; The following operations allow the matrices and signals which
```

```
; characterize the scrambler to be dumped to a file, in human-readable
; form, and later loaded, thus saving the time of generating them.
;
; The dump- operations take two arguments, the object to be dumped,
; and the stream on which to write. These operations are:
;
;         (dump-matrices matrices stream)
;         (dump-matrix matrix1 stream)
;         (dump-discrete-signal signal1 stream)
;         (dump-discrete-filter signal1 stream)
;
; The load operations take only one argument, a stream.  They require
; that the contents of the stream be in the proper form; no error
; checking is provided. They return the object dumped to the stream.
; They are:
;
;         (load-matrices stream)
;         (load-matrix stream)
;         (load-discrete-signal stream)
;         (load-discrete-filter stream)

(defun dump-matrices (matrices stream)
  (let ((count (array-length matrices)))
    (princ count stream) (princ " ;count" stream)
    (do ((i 0 (1+ i)))
        ((= i count))
      (terpri stream)
      (send (aref matrices i) ':dump stream ";Random orthogonal matrix"))))

(defun load-matrices (stream)
  (let* ((count (read stream))
         (matrices (make-array count)))
    (do ((i 0 (1+ i)))
        ((= i count) matrices)
      (aset (load-matrix stream) matrices i))))

(defun dump-matrix (matrix1 stream)
  (send matrix1 ':dump stream))

(defun load-matrix (stream)
  (let* ((rows (read stream))
         (columns (read stream))
         (elements (make-array (list rows columns))))
    (do ((i 0 (1+ i)))
        ((= i rows) T)
      (do ((j 0 (1+ j)))
          ((= j columns) T)
        (aset (read stream) elements i j)))
    (make-matrix elements)))

(defun dump-discrete-signal (signal1 stream)
  (send signal1 ':dump-time stream))

(defun dump-discrete-filter (signal1 stream)
  (send signal1 ':dump-freq stream))

(defun load-discrete-signal (stream)
  (multiple-value-bind (origin inc points)
      (load-signal stream))
  (make-discrete-signal points origin inc))
```

```
(defun load-discrete-filter (stream)
  (multiple-value-bind (origin inc points)
      (load-signal stream))
  (make-discrete-filter points origin inc))

(defun load-signal (stream)

  ; Loads either a discrete-signal or a discrete-filter from a stream.
  ; Returns three values: the origin, the increment, and the array of
  ; points.
  ; Requires the contents of stream were produced by a dump.

  (let* ((origin (read stream))
         (inc (read stream))
         (size (read stream))
         (points (make-array size ':fill-pointer size)))
    (do ((i 0 (1+ i)))
        ((= i size) T)
      (aset (read stream) points i))
    (values origin inc points)))


; ------------------ Transmission line constructors ------------------

(defun make-channel (n period)

  ; Creates a typical telephone channel impulse response, in
  ; "continuous" format, of length 2 * n * period. The response is
  ; taken from a Bell Telephone Laboratories model.  The response
  ; lasts two samples so it can be used for convolution without
  ; circularity in the upper half.

  (let ((full-filter (make-array 91. ':fill-pointer 91.))
        (rotate (make-array n ':fill-pointer n ':initial-value 0.0))
        (filter (make-array n ':fill-pointer n ':initial-value 0.0)))

    (fillarray full-filter
      (mapcar 'exp10
              '(-3.5 -3.2 -2.8 -2.5 -2.2 -1.8 -1.5 -1.3
                -1.1 -0.9 -0.7 -0.6 -0.4 -0.3 -0.3 -0.2
                -0.2 -0.2 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.1
                 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1
                 0.1  0.2  0.2  0.2  0.3  0.3  0.3  0.4
                 0.4  0.4  0.5  0.5  0.4  0.4  0.3  0.3
                 0.3  0.2  0.2  0.1  0.1  0.0  0.0 -0.1
                -0.1 -0.1 -0.2 -0.2 -0.3 -0.4 -0.5 -0.5
                -0.6 -0.7 -0.7 -0.8 -0.8 -0.9 -1.0 -1.0
                -1.1 -1.2 -1.2 -1.3 -1.4 -1.6 -1.7 -2.0
                -2.3 -2.6 -2.9)))

    (aset 1.0 rotate (// n 2))

    ; Fundamental frequency is 1 / n * period.  Full-filter counts in
    ; increments of 39.0625 Hz. Transfer from full-filter (i1) to
    ; filter (i2):

    (do ((i1 0.0 (+ i1 (// (* 39.0625 n period))))
         (i2 0 (1+ i2)))
```

```
        ((>= i1 91.5) T)
      (aset (aref full-filter (fixr i1)) filter i2))

    ; Copy positive frequencies to negative frequencies:

    (do ((i 1 (1+ i)))
        ((>= i (// n 2)))
      (aset (aref filter i) filter (- n i)))

    (let ((filter-signal (make-discrete-filter filter 0.0 period))
          (rotate-signal (make-discrete-signal rotate 0.0 period)))

      ; Convert to continuous format, remove high-frequency aliasing,
      ; rotate peak into lower half of time sample, and extend for
      ; convolution.

      (send filter-signal ':desample)
      (send rotate-signal ':desample)
      (send filter-signal ':bandlimit 0.0 (// .5 period))
      (send filter-signal ':filter rotate-signal)
      (send filter-signal ':extend 2)
      filter-signal)))

(defun make-desampler (n f1 f2 period)

  ; Creates a desampler impulse response which, when applied to a
  ; signal converted to "continuous" form, bandlimits that signal to a
  ; band slightly wider than [f1, f2]. The second half of the response
  ; is zero, so it can be used for convolution.

  (let ((impulse (make-array n ':fill-pointer n ':initial-value 0.0)))
    (aset 1.0 impulse (// n 2))
    (let ((impulse-signal (make-discrete-signal impulse 0.0 period)))
      (send impulse-signal ':desample)
      (send impulse-signal ':bandlimit
            (// f1 2.0) (// (+ f2 (// .5 period)) 2.0))
      (send impulse-signal ':extend 2)
      impulse-signal)))

(defun make-equalizer (desampler channel n f1 f2 period)

  ; Creates a sampled impulse response for an equalizer.  The 8n-point
  ; response compensates for the effects of a desampler and channel to
  ; produce a flat attenuation over the frequency range [f1, f2].

  (let ((equalizer (send channel ':copy))
        (desired-array
          (make-array (* 8. n) ':fill-pointer (* 8. n) ':initial-value 0.0))
    (aset 1.0 desired-array (* 3 n))

    (let ((desired-signal (make-discrete-signal desired-array 0.0 period)))
      (send desired-signal ':bandlimit f1 f2)

      ; Compute combined impulse response, convert to sample rate, and
      ; pad with zeroes.

      (send equalizer ':filter desampler)
      (send equalizer ':sample period)
      (send equalizer ':extend 4)
```

```
        ; Equalize combined impulse response to produce necessary
        ; response of tapped delay line.

        (send equalizer ':equalize desired-signal)
        (send equalizer ':indexlimit 0. (* 7 n period))
        equalizer)))

(defun set-voice-sample (source n f1 f2 period)

  ; Returns a discrete-signal of length n, sample rate period. Source
  ; determines how the signal is produced.
  ;
  ;         'random     a random signal over [f1, f2]
  ;         'impulse    an impulse at 0 followed by n-1 0's
  ;         'zero       a series of n 0's
  ;         'band       a signal uniformly weighted over [f1, f2]
  ;         (other)     n values from source, assumed to be
  ;                     a stream

  (cond ((eq source 'random)
         (make-random-voice-signal n f1 f2 period))
        ((eq source 'impulse)
         (let ((voice-array
                 (make-array n ':fill-pointer n ':initial-value 0.0)))
           (aset 1.0 voice-array 0)
           (make-discrete-signal voice-array 0.0 period)))
        ((eq source 'zero)
         (make-discrete-signal
           (make-array n ':fill-pointer n ':initial-value 1e-7) 0.0 period))
        ((eq source 'band)
         (let ((voice-array
                 (make-array n ':fill-pointer n ':initial-value 0.0)))
           (aset 1.0 voice-array 0)
           (let ((voice-signal (make-discrete-signal voice-array 0.0 period
             (send voice-signal ':bandlimit f1 f2)
             voice-signal)))
        (T (let ((voice-array (make-array n ':fill-pointer n)))
             (do ((i 0 (1+ i)))
                 ((= i n) T)
               (aset (read source) voice-array i))
             (make-discrete-signal voice-array 0.0 period)))))

; ------------------------ System constants ------------------------

(defconst *n* 32. "Number of points in sample")
(defconst *f1* 0. "Input/output low frequency")
(defconst *f2* 2700. "Input/output high frequency")
(defconst *f1~* 300. "Transmitted low frequency")
(defconst *f2~* 3200. "Transmitted high frequency")
(defconst *nu* 22. "Number of weights scrambled")
(defconst *discrete-sample-period* 1.25e-4 "Discrete signal sample time")
(defconst *voice-sample-period* 6.25e-5 "Voice signal sample time")
(defconst *noise-level* 60.0 "Channel noise energy")

(defvar *prolate-spheroidal-basis* nil "Input/output scrambling basis")
(defvar *prolate-spheroidal-basis~* nil "Transmitted scrambling basis")
(defvar *random-orthogonal-matrices* nil "Set of scrambling matrices")
(defvar *channel-filter* nil "Channel transfer function")
(defvar *desampler-impulse-response* nil "Desampler impulse response")
(defvar *tapped-delay-line* nil "Discrete-time channel equalizer")
```

```
; ------------------- Loading and dumping scrambler ------------------
;
; The operations dump-scrambler and load-scrambler allow all of the
; system constants to be written to or read from a file. They are
; provided to avoid generating parameters.

(defun dump-scrambler (file)
  (with-open-file (s file ':direction ':output)
    (princ ";Scrambler" s)
    (print *n* s) (princ " ;n" s)
    (print *f1* s) (princ " ;f1" s)
    (print *f2* s) (princ " ;f2" s)
    (print *f1~* s) (princ " ;f1" s)
    (print *f2~* s) (princ " ;f2" s)
    (print *nu* s) (princ " ;nu" s)
    (print *discrete-sample-period* s) (princ " ;discrete-sample-period" s)
    (print *noise-level* s) (princ " ;noise-level" s) (terpri s)
    (dump-matrix *prolate-spheroidal-basis* s)
    (dump-matrix *prolate-spheroidal-basis~* s)
    (dump-matrices *random-orthogonal-matrices* s)
    (dump-discrete-signal *channel-filter* s)
    (dump-discrete-signal *desampler-impulse-response* s)
    (dump-discrete-signal *tapped-delay-line* s)

(defun load-scrambler (file)
  (with-open-file (s file ':direction ':input)
    (setq
      *n* (read s)
      *f1* (read s)
      *f2* (read s)
      *f1~* (read s)
      *f2~* (read s)
      *nu* (read s)
      *discrete-sample-period* (read s)
      *noise-level* (read s)
      *prolate-spheroidal-basis* (load-matrix s)
      *prolate-spheroidal-basis~* (load-matrix s)
      *random-orthogonal-matrices* (load-matrices s)
      *channel-filter* (load-discrete-signal s)
      *desampler-impulse-response* (load-discrete-signal s)
      *tapped-delay-line* (load-discrete-signal s))))

(defun make-scrambler ()

  ; Creates the scrambler database for *n*, *nu*, *f1*, *f2*, *f1~*,
  ; *f2~*, and *discrete-sample-period*. Six elements are created:
  ;
  ;         *prolate-spheroidal-basis*: for input and output [f1, f2];
  ;         *prolate-spheroidal-basis~*: transmission [f1~, f2~];
  ;         *random-orthogonal-matrices*: nu x nu;
  ;         *channel-filter*;
  ;         *desampler-impulse-response*: [f1~, f2~]; and
  ;         *tapped-delay-line*: [f1~, f2~].

  (setq
    *prolate-spheroidal-basis* (make-prolate-spheroidal-basis
                                 *n* *nu* *f1* *f2* *discrete-sample-period*)
    *prolate-spheroidal-basis~* (make-prolate-spheroidal-basis
                                  *n* *nu* *f1~* *f2~* *discrete-sample-perio
```

```
        *random-orthogonal-matrices* (make-random-orthogonal-matrices *nu* 2.)
        *channel-filter* (make-channel *n* *discrete-sample-period*)
        *desampler-impulse-response* (make-desampler
                                      *n* *f1~* *f2~* *discrete-sample-period*
        *tapped-delay-line* (make-equalizer
                             *desampler-impulse-response*
                             *channel-filter*
                             *n* *f1~* *f2~* *discrete-sample-period*)))
```

```
;;; -*- Mode: LISP; Fonts: CPTFONT; Base: 10. -*-

; ---------- Signal array operations ----------
;
; The set-xx-index-yy operations convert from 'real' time or frequency
; units to an index to the time or frequency array.  The difference
; between low and high is in the roundings of the index (low backs up,
; high advances).
;
; Make-empty and empty-p are used to indicate an array is not valid
; and test its validity.
;
; The first four fft operations are macros which cause an fft to be
; run if the time or frequency array is not valid.  The function fft is
; necessary because the FFT algorithm below is in-place.

(defun set-freq-index-low (freq time-increment time-size)
  (1- (top (* freq time-increment time-size))))

(defun set-freq-index-high (freq time-increment time-size)
  (1+ (bottom (* freq time-increment time-size))))

(defun set-time-index-low (time time-base time-increment)
  (1- (top (// (- time time-base) time-increment))))

(defun set-time-index-high (time time-base time-increment)
  (1+ (bottom (// (- time time-base) time-increment))))

(defun empty-p (values)
  (= 0 (array-active-length values)))

(defun make-empty (values)
  (store-array-leader 0 values 0))

(defmacro forward-fft ()
  `(cond ((empty-p time-values)
          (fft freq-values time-values sample-size 1))))

(defmacro inverse-fft ()
  `(cond ((empty-p freq-values)
          (fft time-values freq-values sample-size -1))))

(defmacro forward-fft2 ()
  `(cond ((empty-p time-values2)
          (fft freq-values2 time-values2 sample-size2 1))))

(defmacro inverse-fft2 ()
  `(cond ((empty-p freq-values2)
          (fft time-values2 freq-values2 sample-size2 -1))))

(defun fft (from to size exponent)
  (copy-array-contents from to)
  (store-array-leader size to 0)
  (fast-fourier-transform to size exponent))
```

```
; ---------- Discrete signal flavor ----------
;
; This flavor is used to represent discrete signals and to approximate
; continuous signals. It maintains an array of time values and an
; array of frequency values; at least one is valid at any time.
;
; Many operations are provided. They can be put into four categories.
;
; The selectors return information about the discrete signal:
;
; energy                          returns the energy of the signal, were
;                                 it a series of impulses
; energy-band low high            computes the energy in the frequency
;                                 spectrum between [low, high] and also
;                                 between [-high, -low]
; sequence                        returns the time values represented
;                                 as a 1-column matrix
; error signal2                   computes the error energy, as shown
;                                 by differences between frequency
;                                 values, between two signals
; extract low-i high-i            returns a new signal taken from the
;                                 time values indexed from low-i to
;                                 high-i; note that low, high in other
;                                 operations are abstract values
; copy &optional signal2          returns a copy of the signal; puts
;                                 it in signal2, if specified
;
; These operators mutate the discrete signal:
;
; sample period                   changes sample size so that sample
;                                 separated by period are represented
; desample factor                 expands sample size by placing zeroes
;                                 between original samples; factor is
;                                 amount of expansion
; indexlimit low high             sets to zero those time samples not
;                                 in the range [low, high]
; bandlimit low high              sets to zero those frequency values
;                                 not in the ranges [low, high] or
;                                 [-high, -low]
; extend factor                   expands sample size by factor by
;                                 adding zeroes to the end of the time
;                                 sequence
; add-noise level                 adds to the time values random noise
;                                 with mean 0 and variance level
; scale factor                    scales the time values by factor
;
; Some operators take another signal as an argument, and mutate the
; first signal:
;
; append signal2                  adds the time values of signal2 to
;                                 those already in the discrete signal
; filter signal2                  multiplies the signal's frequency
;                                 values by those in signal2
; convolve signal2                performs circular time-domain
;                                 convolution with signal2; can be used
;                                 when signals are of different sizes
; equalize signal2                modifies signal so that filtering it
;                                 with its original value would produce
;                                 signal2
;
```

```
; The remaining methods are used for output:
;
; dump-time stream                writes to stream a representation of
;  &optional doc                  signal's time values, with string doc
; dump-freq stream                writes to stream a representation of
;  &optional doc                  signal's frequency values, with string
;                                 doc
; display-time                    displays on window a plot of time
;  &optional fun window           values of signal, applying function
;                                 fun to each value
; display-freq                    displays on window a plot of frequency
;  &optional fun window           values of signal, applying function
;                                 fun to each value

(defflavor discrete-signal
           (time-values
            time-base
            time-increment
            sample-size
            freq-values)
           ()
           :initable-instance-variables
           :settable-instance-variables
           :special-instance-variables)

; ---------------------------- Selectors ----------------------------

(defmethod (discrete-signal :energy)
           ()
  (forward-fft)
  (do ((i 0 (1+ i))
       (e 0.0 (+ e (^ (abs-complex (aref time-values i)) 2))))
      ((= i sample-size) (* e time-increment))
    nil))

(defmethod (discrete-signal :energy-band)
           (low high)
  (inverse-fft)
  (let ((low-index (set-freq-index-low low time-increment sample-size))
        (high-index (set-freq-index-high high time-increment sample-size)))
    (let ((e (+ (do ((i (1+ low-index) (1+ i))
                     (e 0.0 (+ e (^ (abs-complex (aref freq-values i)) 2))))
                    ((or (>= i high-index)
                         (>= i (1+ (- sample-size high-index))))
                     e)
                  nil)
                (do ((i (1+ (- sample-size high-index)) (1+ i))
                     (e 0.0 (+ e (^ (abs-complex (aref freq-values i)) 2))))
                    ((or (>= i (- sample-size low-index))
                         (>= i sample-size))
                     e)
                  nil))))
      (* e sample-size time-increment))))

(defmethod (discrete-signal :sequence)
           ()
  (forward-fft)
  (make-matrix time-values))

(defmethod (discrete-signal :extract)
```

```
            (low-index high-index)
    (forward-fft)
    (let ((result (make-array (- high-index low-index -1)
                              ':fill-pointer (- high-index low-index -1))))
      (do ((i low-index (1+ i)))
          ((> i high-index)
           (make-discrete-signal result
                                 (+ time-base (* low-index time-increment))
                                 time-increment))
        (aset (aref time-values i) result (- i low-index)))))

(defmethod (discrete-signal :copy)
           (&optional signal2)
  (let ((time-values2
         (if signal2 (send signal2 ':time-values)
                     (make-array sample-size ':fill-pointer 0)))
        (freq-values2
         (if signal2 (send signal2 ':freq-values)
                     (make-array sample-size ':fill-pointer 0))))
    (copy-array-contents-and-leader time-values time-values2)
    (copy-array-contents-and-leader freq-values freq-values2)
    (if signal2 signal2
        (make-instance 'discrete-signal ':time-values time-values2
                                        ':freq-values freq-values2
                                        ':time-base time-base
                                        ':time-increment time-increment
                                        ':sample-size sample-size))))

(defmethod (discrete-signal :error)
           (signal2)
  (inverse-fft)
  (let ((sample-size2 (send signal2 ':sample-size))
        (time-values2 (send signal2 ':time-values))
        (freq-values2 (send signal2 ':freq-values)))
    (inverse-fft2)
    (do ((i 0 (1+ i))
         (e 0.0 (+ e (^ (abs-complex
                          (sub-complex (abs-complex (aref freq-values i))
                                       (abs-complex (aref freq-values2 i))))
                        2))))
        ((= i sample-size) (* e time-increment sample-size))
      nil)))

; ---------------------------- Mutators ----------------------------

(defmethod (discrete-signal :sample)
           (period)
  (forward-fft)
  (let ((count (top (// period time-increment))))
    (do ((i 0 (1+ i))
         (j 0 (+ count j)))
        ((>= j sample-size) T)
      (aset (aref time-values j) time-values i))
    (setq time-increment period)
    (setq sample-size (top (// sample-size count))))
    (store-array-leader sample-size time-values 0)
  (make-empty freq-values))

(defmethod (discrete-signal :desample)
           (&optional (factor 8.))
```

```
  (forward-fft)
  (adjust-array-size time-values (* factor sample-size))
  (adjust-array-size freq-values (* factor sample-size))
  (do ((i (1- sample-size) (1- i)))
      ((< i 0) T)
    (aset (aref time-values i) time-values (* factor i))
    (do ((j 1 (1+ j)))
        ((= j factor) T)
      (aset 0.0 time-values (+ (* factor i) j))))
  (setq time-increment (// time-increment factor))
  (setq sample-size (* sample-size factor))
  (do ((i 0 (1+ i)))
      ((= i sample-size) T)
    (aset 0.0 freq-values i))
  (store-array-leader sample-size time-values 0)
  (make-empty freq-values))

(defmethod (discrete-signal :indexlimit)
           (low high)
  (forward-fft)
  (let ((low-index (set-time-index-low low time-base time-increment)))
    (do ((i 0 (1+ i)))
        ((> i low-index) T)
      (aset 0.0 time-values i)))
  (let ((high-index (set-time-index-high high time-base time-increment)))
    (do ((i (1- sample-size) (1- i)))
        ((< i high-index) T)
      (aset 0.0 time-values i)))
  (make-empty freq-values))

(defmethod (discrete-signal :bandlimit)
           (low high)
  (inverse-fft)
  (let ((low-index (set-freq-index-low low time-increment sample-size))
        (high-index (set-freq-index-high high time-increment sample-size)))
    (do ((i 0 (1+ i)))
        ((> i low-index) T)
      (aset 0.0 freq-values i))
    (do ((i high-index (1+ i)))
        ((> i (- sample-size high-index)) T)
      (aset 0.0 freq-values i))
    (do ((i (- sample-size low-index) (1+ i)))
        ((>= i sample-size) T)
      (aset 0.0 freq-values i)))
  (make-empty time-values))

(defmethod (discrete-signal :extend)
           (factor)
  (forward-fft)
  (let ((new-size (* sample-size factor)))
    (adjust-array-size time-values new-size)
    (adjust-array-size freq-values new-size)
    (do ((i sample-size (1+ i)))
        ((= i new-size) T)
      (aset 0.0 time-values i)
      (aset 0.0 freq-values i))
    (setq sample-size new-size)
    (store-array-leader sample-size time-values 0)
    (make-empty freq-values)))
```

```
(defmethod (discrete-signal :add-noise)
           (level)
  (forward-fft)
  (do ((i 0 (1+ i)))
      ((= i sample-size) T)
    (aset (add-complex (aref time-values i)
                       (* level (normal-random))) time-values i))
  (make-empty freq-values))

(defmethod (discrete-signal :scale)
           (factor)
  (forward-fft)
  (inverse-fft)
  (do ((i 0 (1+ i)))
      ((= i sample-size))
    (aset (mult-complex (aref time-values i) factor) time-values i)
    (aset (mult-complex (aref freq-values i) factor) freq-values i)))

; ---------------------------- Operators ----------------------------

(defmethod (discrete-signal :append)
           (signal2)
  (forward-fft)
  (let ((sample-size2 (send signal2 ':sample-size))
        (time-values2 (send signal2 ':time-values))
        (freq-values2 (send signal2 ':freq-values)))
    (forward-fft2)
    (do ((i sample-size2 (1+ i)))
        ((>= i sample-size) T)
      (aset (aref time-values i) time-values (- i sample-size2)))
    (do ((i2 0 (1+ i2))
         (i (- sample-size sample-size2) (1+ i)))
        ((= i2 sample-size2) T)
      (aset (aref time-values2 i2) time-values i))
    (setq time-base (+ time-base (* sample-size2 time-increment)))
    (make-empty freq-values)))

(defmethod (discrete-signal :filter)
           (signal2)
  (inverse-fft)
  (let ((sample-size2 (send signal2 ':sample-size))
        (time-values2 (send signal2 ':time-values))
        (freq-values2 (send signal2 ':freq-values)))
    (inverse-fft2)
    (do ((i 0 (1+ i)))
        ((= i sample-size) T)
      (aset (mult-complex sample-size2
                          (mult-complex (aref freq-values i) (aref freq-valu
            freq-values i)))
  (make-empty time-values))

(defmethod (discrete-signal :convolve)
           (signal2)
  (forward-fft)
  (let ((sample-size2 (send signal2 ':sample-size))
        (time-values2 (send signal2 ':time-values))
        (freq-values2 (send signal2 ':freq-values))
        (convolution (make-array sample-size)))
    (forward-fft2)
    (do ((i 0 (1+ i)))
```

```
        ((= i sample-size) T)
      (do ((J2 0 (1+ J2))
           (sum 0.0 (add-complex sum (mult-complex
                                       (aref time-values2 J2)
                                       (if (> J2 i)
                                           (aref time-values
                                                 (+ sample-size (- i J2)))
                                           (aref time-values (- i J2)))))))
          ((= J2 sample-size2) (aset sum convolution i))
        nil))
    (do ((i 0 (1+ i)))
        ((= i sample-size) T)
      (aset (aref convolution i) time-values i))))

(defmethod (discrete-signal :equalize)
           (signal2)
  (inverse-fft)
  (let ((sample-size2 (send signal2 ':sample-size))
        (time-values2 (send signal2 ':time-values))
        (freq-values2 (send signal2 ':freq-values)))
    (inverse-fft2)
    (do ((i 0 (1+ i)))
        ((= i sample-size) T)
      (aset (div-complex (aref freq-values2 i)
                         (mult-complex (aref freq-values i) sample-size2))
            freq-values i)))
  (make-empty time-values))

; ------------------------ Dump and display ------------------------

(defmethod (discrete-signal :dump-time)
           (stream &optional (documentation "Time Values"))
  (forward-fft)
  (dump-signal time-values time-base time-increment
               sample-size stream documentation))

(defmethod (discrete-signal :dump-freq)
           (stream &optional (documentation "Freq Values"))
  (inverse-fft)
  (dump-signal freq-values time-base time-increment
               sample-size stream documentation))

(defun dump-signal (points origin inc size stream documentation)
  (princ documentation stream)
  (print origin stream) (princ " ;time base" stream)
  (print inc stream) (princ " ;time increment" stream)
  (print size stream) (princ " ;sample size" stream)
  (do ((i 0 (1+ i)))
      ((= i size) (terpri stream))
    (print (aref points i) stream)
    (princ " ;" stream) (princ i stream)))

(defmethod (discrete-signal :display-time)
           (&optional (fun 'real-part)
            (window terminal-io))
  (forward-fft)
  (display-signal time-values time-base time-increment
                  sample-size window "Time (sec)" "Amplitude" fun))

(defmethod (discrete-signal :display-freq)
```

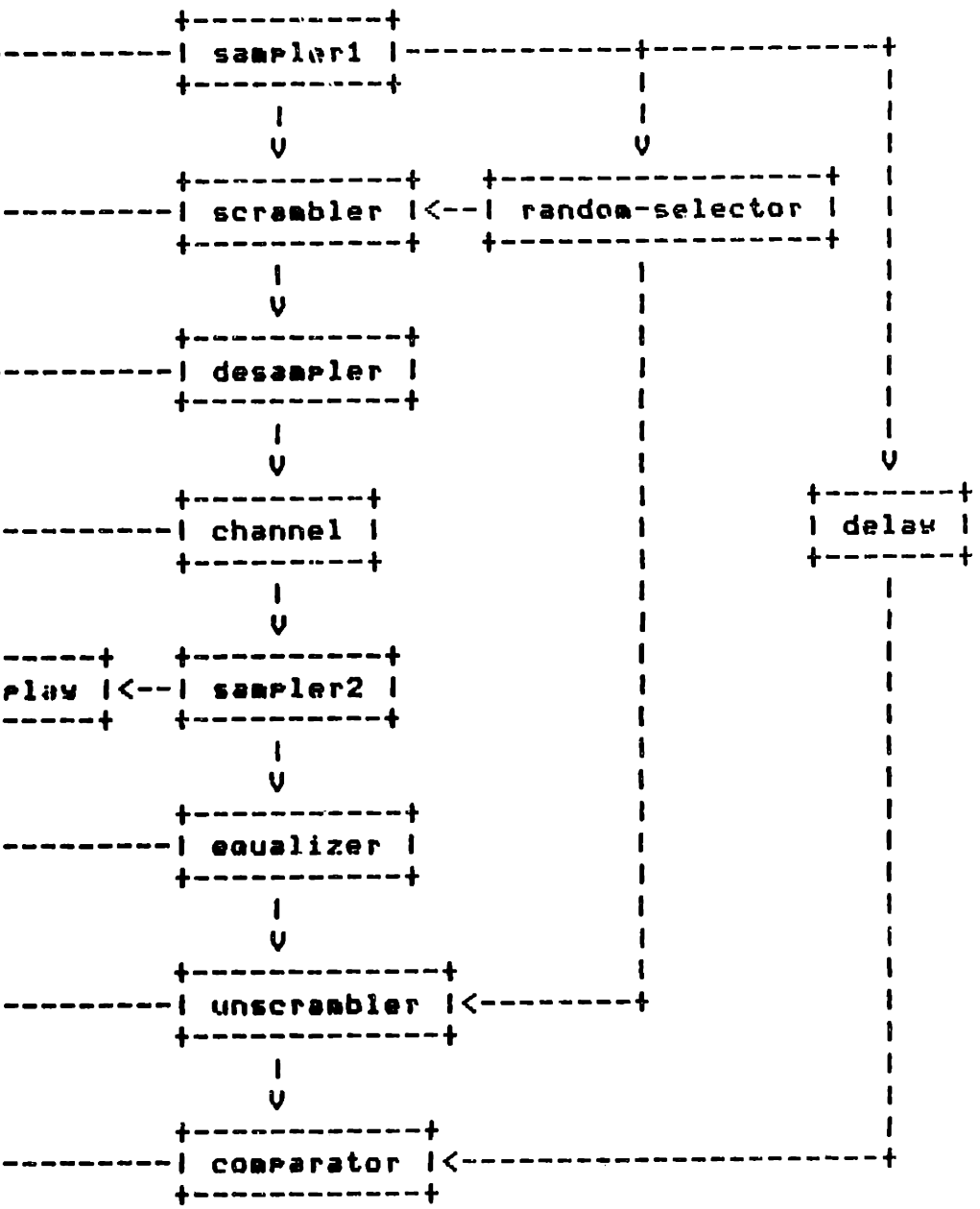
```
            (&optional (fun 'abs-complex)
             (window terminal-io))
  (inverse-fft)
  (display-signal freq-values 0.0 (// (* time-increment sample-size))
                  sample-size window "Frequency (Hz)" "Amplitude" fun))

(defun display-signal (points origin inc size window x-label y-label fun)
  (let ((max-y (max-fun-complex points fun)))
    (draw-axes window origin inc size max-y x-label y-label)
    (draw-points window size max-y points fun)))

(defun draw-axes (window origin inc size max-y x-label y-label)
  (multiple-value-bind (iw ih) (send window ':inside-size)
    (let* ((font (send window ':current-font))
           (fw (tv:font-char-width font))
           (fh (tv:sheet-line-height window))
           (sw (- iw (* 22 fw)))
           (sh (- ih (* 8 fh))))
      (send window ':clear-screen)

      ; Draw frame around graph, and dashed x-axis

      (send window ':draw-lines (tv:sheet-char-aluf window)
            (* 10 fw) (* 3 fh)
            (- iw (* 10 fw)) (* 3 fh)
            (- iw (* 10 fw)) (- ih (* 3 fh))
            (* 10 fw) (- ih (* 3 fh))
            (* 10 fw) (* 3 fh))
      (send window ':draw-dashed-line (* 10 fw) (// ih 2)
            (- iw (* 10 fw)) (// ih 2))

      ; Print x-axis label, centered, at top and bottom of screen

      (send window ':set-cursorpos
            (// (- iw (* (string-length x-label) fw)) 2) (- ih fh))
      (send window ':string-out x-label)
      (send window ':set-cursorpos
            (// (- iw (* (string-length x-label) fw)) 2) fh)
      (send window ':string-out x-label)

      ; Print y-axis label, centered, along left and right margins

      (do ((y (// (- ih (* (string-length y-label) fh)) 2) (+ y fh))
           (i 0 (1+ i)))
          ((= i (string-length y-label)) T)
        (send window ':set-cursorpos 0 y)
        (send window ':tyo (find-char y-label i))
        (send window ':set-cursorpos (- iw fw) y)
        (send window ':tyo (find-char y-label i)))

      ; Print x-axis coordinates to divide axis into ten parts at top,
      ; bottom of screen

      (do ((x (* 8.5 fw) (+ x (* .1 sw)))
           (i 0 (1+ i)))
          ((= i 10.) T)
        (send window ':set-cursorpos (fixr x) (* 2 fh))
        (format window "~2E" (+ origin (* .1 i inc (1- size))))
        (send window ':set-cursorpos (fixr x) (- ih (* 2 fh)))
        (format window "~2E" (+ origin (* .1 i inc (1- size))))))
```

```
        ; Print y-axis coordinates to divide axis into ten parts at
        ; left, right margins

        (do ((y (+ (// ih 2.0) (* .4 sh) (// fh -2.0)) (- y (* .1 sh)))
             (i -4 (1+ i)))
            ((= i 5) T)
          (send window ':set-cursorpos (* 2 fw) (fixr y))
          (format window "~2E" (* max-y (// i 5.0)))
          (send window ':set-cursorpos (- iw (* 7 fw)) (fixr y))
          (format window "~2E" (* max-y (// i 5.0))))))))

(defun draw-points (window size max-y points fun)
  (multiple-value-bind (iw ih) (send window ':inside-size)
    (let* ((font (send window ':current-font))
           (fw (tv:font-char-width font))
           (fh (tv:sheet-line-height window))
           (sw (- iw (* 22 fw)))
           (sh (- ih (* 8 fh))))

      (do ((x (1- (* 11 fw)) (+ x (// sw (float (1- size)))))
           (i 0 (1+ i)))
          ((= i size) T)
        (let* ((val (float (apply fun (list (aref points i)))))
               (y (- (// ih 2.0) (* (// sh 2.0) (// val max-y)))))
          (if (< val 0.0)
              (send window ':draw-rectangle 1 (fixr (- y (// ih 2)))
                    (1+ (fixr x)) (// ih 2))
            (send window ':draw-rectangle 1 (fixr (- (// ih 2) y))
                  (1+ (fixr x)) (fixr y)))
          (send window ':draw-circle (1+ (fixr x)) (fixr y) 3)))))
  (send window ':home-down))


; ------------------------ Fast Fourier Transform ------------------------
;
; The fast Fourier transform is adapted from class notes for 6.046
; (Algorithms). It is iterative and uses a preliminary bit reversal of
; its input. The FFT is done in place. The output is scaled for
; inverse transforms. Only sizes which are powers of 2 are supported.
; If the size is not a power of 2, it uses the slower, recursive
; algorithm below.
;
; (fast-fourier-transform          computes the FFT of array a of size n,
;   a n iexp)                      where iexp is the direction (1 =
;                                  forward)
; (fft-reorder a n)                reorders by bit reversal n elements of
;                                  a;
; (swap i j)                       exchanges elements a[i] and a[j];
; (rev i n)                        reverses the n lowest bits of i;
;
; Typical run-time of fast algorithm is 1.5 ms x n log n; slow
; algorithm takes twice as long for n a power of 2, and much longer
; for other n.

(defun fast-fourier-transform (a n iexp)
  (if ((not (= (^ 2 (1- (haulong n))) n))
      (slow-fourier-transform a n iexp)
      (fft-reorder a n)
      (do ((s 1 (+ s s))
```

```
            (omega (- iexp) (make-polar (// (* iexp 3.141592) 2. s))))
           ((= s n) T)
         (do ((omega-j 1.0 (mult-complex omega omega-j))
              (j 0 (1+ j)))
             ((= j s) T)
           (do ((i j (+ i s s)))
               ((>= i n) T)
             (let ((tt (mult-complex omega-j (aref a (+ i s))))
                   (u (aref a i)))
               (aset (add-complex u tt) a i)
               (aset (sub-complex u tt) a (+ i s))))))
       (cond ((= iexp -1)
              (do ((j 0 (1+ j)))
                  ((= j n) T)
                (aset (div-complex (aref a j) n) a j))))))

(defun fft-reorder (a n)
  (do ((i 0 (1+ i)))
      ((= i n) nil)
    (let ((r (rev i n)))
      (cond ((< i r) (swap a i r))))))

(defun swap (a i j)
  (let ((temp (aref a i)))
    (aset (aref a j) a i)
    (aset temp a j)))

(defun rev (i n)
  (do ((j i (// j 2))
       (m (// n 2) (// m 2))
       (r 0 (if (oddp j) (+ r r 1) (+ r r))))
      ((< m 1) r)
    nil))

(defun slow-fourier-transform (a n iexp)
  (let ((n2 (// n 2))
        (w (make-polar (* iexp 6.2831852 (// 1.0 n)))))

    ; Three cases -- if n is 1, DFT is a;
    ;                if n is even, DFT is composition of smaller
    ;                 DFTs of even and odd elements of a;
    ;                if n is otherwise, DFT is naive calculation

    (cond ((= n 1) a)

          ((= n (* n2 2))
           (let ((l (make-array n2))
                 (h (make-array n2)))
             (do ((i 0 (+ i 2))
                  (i2 0 (1+ i2)))
                 ((= i2 n2) T)
               (aset (aref a i) l i2)
               (aset (aref a (1+ i)) h i2))
             (slow-fourier-transform l n2 iexp)
             (slow-fourier-transform h n2 iexp)
             (do ((i 0 (1+ i))
                  (wl 1.0 (mult-complex wl w))
                  (wh -1.0 (mult-complex wh w)))
                 ((= i n2) T)
               (aset (add-complex (aref l i)
```

```
                                (mult-complex wl (aref h i))) a i)
        (aset (add-complex (aref l i)
                                (mult-complex wh (aref h i))) a (+ i n2))
    (if (= iexp -1)
        (do ((i 0 (1+ i)))
            ((= i n) T)
          (aset (div-complex (aref a i) 2.0) a i))))
  a)

 (T (let ((r (make-array n ':initial-value 0.0)))
      (do ((i 0 (1+ i))
           (wi 1.0 (mult-complex wi w)))
          ((= i n) T)
        (do ((j 0 (1+ j))
             (wij 1.0 (mult-complex wij wi)))
            ((= j n) T)
          (aset (add-complex (aref r i)
                             (mult-complex wij (aref a j)))
                r i)))
      (do ((i 0 (1+ i)))
          ((= i n) T)
        (aset (aref r i) a i))
      (if (= iexp -1)
          (do ((i 0 (1+ i)))
              ((= i n) T)
            (aset (div-complex (aref a i) n) a i))))
    a))))
```

```
; -*- Mode: LISP; Base: 10.; Fonts: CPTFONT -*-

; ---------------------------- Scrambler ----------------------------
;
; Test-scrambler uses the block diagram below to simulate the complete
; scrambling algorithm. It includes a channel which adds noise. The
; signal at each stage is displayed, and the input and output signals
; (with a delay of 3 signals) are compared. The input is from a speech
; source file at 16 kHz. A simple downsampling to the desired rate is
; assumed.
;
```




```
(defun test-scrambler ()
  (with-open-file (input "mc:users;ikalisk input" ':direction ':input)
    (let ((sampler1 (make-instance 'sampler))
          (scrambler (make-instance 'scrambler))
          (desampler (make-instance 'desampler))
          (channel (make-instance 'channel))
          (sampler2 (make-instance 'sampler))
          (random-selector (make-instance 'random-selector))
          (unscrambler (make-instance 'unscrambler))
          (equalizer (make-instance 'equalizer))
```

```
            (delay (make-instance 'delay))
            (comparator (make-instance 'comparator))
            (display (make-instance 'display))
            (voice-n (fixr (* *n* (// *discrete-sample-period*
                                     *voice-sample-period*)))))
        (send sampler1 ':connect scrambler)
        (send sampler1 ':connect random-selector)
        (send sampler1 ':connect delay)
        (send sampler1 ':connect display)
        (send random-selector ':connect unscrambler)
        (send random-selector ':connect scrambler)
        (send scrambler ':connect desampler)
        (send scrambler ':connect display)
        (send desampler ':connect channel)
        (send desampler ':connect display)
        (send channel ':connect sampler2)
        (send channel ':connect display)
        (send sampler2 ':connect equalizer)
        (send sampler2 ':connect display)
        (send equalizer ':connect unscrambler)
        (send equalizer ':connect display)
        (send unscrambler ':connect comparator)
        (send unscrambler ':connect display)
        (send delay ':connect comparator)
        (do ((i 0 (1+ i)))
            ((= i 146.) T)
        (send sampler1 ':process
              (set-voice-sample
                input voice-n *f1* *f2* *voice-sample-period*)))))

; Fast-scrambler approximates the scrambling algorithm. It puts the
; desampler, channel, and equalizer in one stage. Noise is not added.
; The output at each stage is displayed; this could easily be changed
; to dump transmitted and received output to two files for later
; playback.
;
;                +----------+
;    +-----------| sampler1 |------------+
;    |           +----------+            |
;    |                |                  |
;    |                V                  V
;    |           +-----------+   +-----------------+
;    +-----------| scrambler |<--| random-selector |
;    |           +-----------+   +-----------------+
;    |                |                  |
;    |                V                  |
; +----------+   +--------------+         |
; | display |<--| transmission |         |
; +----------+   +--------------+         |
;    ^                |                  |
;    |                V                  |
;    |           +-------------+         |
;    +-----------| unscrambler |<--------+
;                +-------------+
;
;
;
;
;
```

```
(defun fast-scrambler ()
  (with-open-file (input "mc:users1;kalisk input")
    (let ((sampler1 (make-instance 'sampler))
          (scrambler (make-instance 'scrambler))
          (random-selector (make-instance 'random-selector))
          (transmission (make-instance 'transmission))
          (unscrambler (make-instance 'unscrambler))
          (display (make-instance 'display))
          (voice-n (fixr (* *n* (// *discrete-sample-period*
                                   *voice-sample-period*)))))
      (send sampler1 ':connect scrambler)
      (send sampler1 ':connect random-selector)
      (send sampler1 ':connect display)
      (send random-selector ':connect unscrambler)
      (send random-selector ':connect scrambler)
      (send scrambler ':connect transmission)
      (send scrambler ':connect display)
      (send transmission ':connect unscrambler)
      (send transmission ':connect display)
      (send unscrambler ':connect display)))
      (do ((i 0 (1+ i)))
          ((= i 146.) T)
        (send sampler1 ':process
              (get-voice-sample
                input voice-n *f1* *f2* *voice-sample-period*)))))

; Send-signal is used to move a signal from one module to those which
; follow it in the block diagram. Send-random-matrix is used by
; random-selector to set the next random matrix in scrambler and
; unscrambler.

(defun send-signal (connect-list signal1)
  signal1
  (mapcar '(lambda (x) (send x ':process signal1)) connect-list))

(defun send-random-matrix (connect-list matrix)
  matrix
  (mapcar '(lambda (x) (send x ':set-random-matrix matrix)) connect-list))

; ---------------------------- Basic module ----------------------------
;
; Basic-module is the common component of all module flavors which can
; be connected to other modules.  Connect-list is the list of modules
; in the block diagram which follow a module. The operation connect
; adds a new module to that list.

(defflavor basic-module
           ((connect-list nil))
           ()
  :initable-instance-variables
  :special-instance-variables)

(defmethod (basic-module :connect)
           (module2)
  (setq connect-list (cons module2 connect-list)))

; ---------------------------- Down-sampler ----------------------------
;
; The down-sampler is used to convert a signal at some sample rate to
; one at another sample rate when the two rates are not related by a
```

```
; constant factor. Input-period and output-period are those rates. The
; process operation desamples its input signal to some large rate
; divisible by both rates, then samples the result to send to its
; successors.

(defflavor down-sampler
           ((input-period *voice-sample-period*)
            (output-period *discrete-sample-period*))
           (basic-module)
  :initable-instance-variables)

(defmethod (down-sampler :process)
           (signal1)
  (let ((factor (fixr (// (* output-period
                              (gcd (fixr (// input-period))
                                   (fixr (// output-period))))))))
    (send signal1 ':desample factor)
    (send signal1 ':bandlimit 0.0 (// output-period))
    (send signal1 ':sample output-period)
    (send signal1 ':scale factor)
    (send-signal connect-list signal1)))

; ------------------------------ Sampler ------------------------------
;
; The sampler converts a signal from one period to another which is a
; multiple. It sends the result to its successors.

(defflavor sampler
           ((period *discrete-sample-period*))
           (basic-module)
  :initable-instance-variables)

(defmethod (sampler :process)
           (signal1)
  (send signal1 ':sample period)
  (send-signal connect-list signal1))

; ------------------------- Random selector ----------------------
;
; The random selector chooses a scrambling matrix from the set
; matrices, and passes that matrix to its successors.

(defflavor random-selector
           ((matrices *random-orthogonal-matrices*))
           (basic-module)
  :initable-instance-variables)

(defmethod (random-selector :process)
           (signal1)
  signal1
  (send-random-matrix connect-list
                      (choose-random-orthogonal-matrix matrices)))

; ---------------------------- Scrambler ---------------------------
;
; The scrambler performs the forward, or input to transmission,
; scrambling of a signal. It uses basis1 to compute the weights to be
; scrambled, rotates those weights by random-matrix, and recomposes
; them using basis2 to make a signal to be transmitted to its
; successors.
```

```
(defflavor scrambler
           ((basis1 *prolate-spheroidal-basis*)
            (basis2 *prolate-spheroidal-basis^*)
            random-matrix)
           (basic-module)
  (:initable-instance-variables basis1 basis2)
  (:settable-instance-variables random-matrix))

(defmethod (scrambler :process)
           (signal1)
  (let ((output (send (send (send (send
                                    signal1 ':sequence)
                                  ':multiply-transposed basis1)
                            ':multiply random-matrix)
                    ':multiply basis2)))
    (let ((output-signal
           (make-discrete-signal (send output ':column 0)
                                 (send signal1 ':time-base)
                                 (send signal1 ':time-increment))))
      (send-signal connect-list output-signal))))

; ------------------------------ Unscrambler ------------------------------
;
; The unscrambler performs the inverse, or transmission to output,
; scrambling of a signal.  It uses basis1 to compute the weights to be
; scrambled, rotates those weights by random-matrix, and recomposes
; them using basis2 to make a signal to be transmitted to successors.

(defflavor unscrambler
           ((basis1 *prolate-spheroidal-basis^*)
            (basis2 *prolate-spheroidal-basis*)
            random-matrix)
           (basic-module)
  (:initable-instance-variables basis1 basis2)
  (:settable-instance-variables random-matrix))

(defmethod (unscrambler :process)
           (signal1)
  (let ((output (send (send (send (send
                                    signal1 ':sequence)
                                  ':multiply-transposed basis1)
                            ':multiply-transposed random-matrix)
                    ':multiply basis2)))
    (let ((output-signal
           (make-discrete-signal (send output ':column 0)
                                 (send signal1 ':time-base)
                                 (send signal1 ':time-increment))))
      (send-signal connect-list output-signal))))

; ---------------------------- Comparator ----------------------------
;
; The comparator determines the error energy between two signals and
; displays it, along with the energies of the signal, on window.  The
; process operation uses pairs of signals.  It saves the first signal
; it receives, and performs the comparison when the second signal is
; received.  It has no successors.

(defflavor comparator
           ((window terminal-io)
```

```
            (signal1 nil))
           ()
  :initable-instance-variables
  (:special-instance-variables signal1))

(defmethod (comparator :process)
           (signal2)
  (cond ((not signal1) (setq signal1 signal2))
        (t (send window ':select)
           (send window ':home-cursor)
           (princ "Energy of first signal = " window)
           (princ (send signal1 ':energy) window)
           (princ "; energy of second signal = " window)
           (princ (send signal2 ':energy) window)
           (princ "; error = " window)
           (princ (send signal1 ':error signal2) window)
           (princ ".")
           (twi window)
           (setq signal1 nil))))

; ----------------------------- Display -----------------------------
;
; The display plots on window the time and frequency representations
; of a signal, and the energy of that signal. It has no successors.

(defflavor display
           ((window terminal-io))
           ()
  :initable-instance-variables)

(defmethod (display :process) (signal1)
  (send window ':select)
  (send window ':home-cursor)
  (send window ':fresh-line)
  (princ "Energy of signal = " window)
  (princ (send signal1 ':energy) window)
  (princ ".")
  (twi window)
  (send signal1 ':display-time 'real-part window)
  (twi window)
  (send signal1 ':display-freq 'abs-complex window)
  (twi window))

; ----------------------------- Desampler -----------------------------
;
; The desampler converts signals from a discrete representation to a
; continuous one, which has a smaller sampling period. It applies an
; impulse response to two most recent input signals to produce an
; output signal. Buffer holds the recent input signals; buffer2 is
; used for applying the impulse-response. The process operation sends
; a continous signal to its successors.

(defflavor desampler
        ((buffer (make-discrete-signal
                   (make-array (* *n* 16.)
                               ':fill-pointer (* *n* 16.)
                               ':initial-value 0.0)
                   (- (* *n* *discrete-sample-period*))
                   (// *discrete-sample-period* 8.)))
         (buffer2 (make-discrete-signal
```

```
                    (make-array (* *n* 16.)
                                ':fill-pointer (* *n* 16.)
                                ':initial-value 0.0)
                    (- (* *n* *discrete-sample-period*))
                    (// *discrete-sample-period* 8.)))
          (impulse-response *desampler-impulse-response*))
        (basic-module)
    :initable-instance-variables)

(defmethod (desampler :process) (signal1)
  (let ((signal1 (send signal1 ':copy)))
    (send signal1 ':desample)
    (send buffer ':append signal1)
    (send buffer ':copy buffer2)
    (let ((sample-size (send buffer ':sample-size)))
      (send buffer2 ':filter impulse-response)
      (send-signal connect-list
                   (send buffer2 ':extract (// sample-size 2)
                         (1- sample-size))))))

; ---------------------------- Equalizer ----------------------------
;
; The equalizer applies impulse-response to its eight most recent
; input signals to produce an equalized output signal. Buffer holds
; the recent signals, and buffer2 is used for applying the impulse
; response. The output is sent to successors.

(defflavor equalizer
        ((buffer (make-discrete-signal
                   (make-array (* *n* 8.)
                               ':fill-pointer (* *n* 8.)
                               ':initial-value 0.0)
                   (- (* *n* *discrete-sample-period*))
                   *discrete-sample-period*))
         (buffer2 (make-discrete-signal
                    (make-array (* *n* 8.)
                                ':fill-pointer (* *n* 8.)
                                ':initial-value 0.0)
                    (- (* *n* *discrete-sample-period*))
                    *discrete-sample-period*))
         (impulse-response *tapped-delay-line*))
        (basic-module)
  :initable-instance-variables)

(defmethod (equalizer :process)
           (signal1)
  (send buffer ':append signal1)
  (send buffer ':copy buffer2)
  (let ((sample-size (send buffer ':sample-size)))
    (send buffer2 ':filter impulse-response)
    (send-signal connect-list
                 (send buffer2 ':extract (// (* sample-size 7) 8.)
                       (1- sample-size)))))

; ---------------------------- Channel ----------------------------
;
; The channel maps continous input signals to continuous output
; signals, using an impulse-response on the two most recent signals.
; Buffer holds the recent signals, and buffer2 is used to apply the
; impulse response.  Noise determines the level of random noise added
```

```
; after applying the impulse response. The output signal is sent to
; successors.

(defflavor channel
        ((buffer (make-discrete-signal
                   (make-array (* *n* 16.)
                               ':fill-pointer (* *n* 16.)
                               ':initial-value 0.0)
                   (- (* *n* *discrete-sample-period*))
                   (// *discrete-sample-period* 8.)))
         (buffer2 (make-discrete-signal
                    (make-array (* *n* 16.)
                                ':fill-pointer (* *n* 16.)
                                ':initial-value 0.0)
                    (- (* *n* *discrete-sample-period*))
                    (// *discrete-sample-period* 8.)))
         (impulse-response *channel-filter*)
         (noise *noise-level*))
        (basic-module)
  :initable-instance-variables)

(defmethod (channel :process)
           (signal1)
  (send buffer ':append signal1)
  (send buffer ':copy buffer2)
  (let ((sample-size (send buffer ':sample-size)))
    (send buffer2 ':filter impulse-response)
    (send buffer2 ':add-noise noise)
    (send-signal connect-list (send buffer2 ':extract (// sample-size 2)
                                      (1- sample-size)))))

; ------------------------- Transmission -------------------------
;
; The transmission provides a fast way to perform desampler, channel,
; and equalizer operations, without using continous signals. Buffer
; holds the eight most recent input signals. Buffer2 is used for
; applying the impulse response, which is computed by combining the
; impulse responses of the three modules above. The output signal is
; sent to successors.

(defflavor transmission
           ((buffer (make-discrete-signal
                      (make-array (* *n* 8.)
                                  ':fill-pointer (* *n* 8.)
                                  ':initial-value 0.0)
                      (- (* *n* *discrete-sample-period*))
                      *discrete-sample-period*))
            (buffer2 (make-discrete-signal
                       (make-array (* *n* 8.)
                                   ':fill-pointer (* *n* 8.)
                                   ':initial-value 0.0)
                       (- (* *n* *discrete-sample-period*))
                       *discrete-sample-period*))
            (impulse-response
              (let ((i-r (send *channel-filter* ':copy)))
                (send i-r ':filter *desampler-impulse-response*)
                (send i-r ':sample *discrete-sample-period*)
                (send i-r ':extend 4)
                (send i-r ':filter *tapped-delay-line*)
                i-r)))
```

```
            (basic-module)
  :initable-instance-variables)

(defmethod (transmission :process) (signal1)
  (send buffer ':append signal1)
  (send buffer ':copy buffer2)
  (let ((sample-size (send buffer ':sample-size)))
    (send buffer2 ':filter impulse-response)
    (send-signal connect-list (send buffer2 ':extract
                                    (// (* sample-size 7.) 8.)
                                    (1- sample-size))))

; ------------------------------ Delay ------------------------------
;
; The delay introduces a wait between receiving input and producing
; output.  Wait is the number of samples delayed; signals is a buffer
; of samples accumulated. The process operation outputs zero-valued
; signals until the buffer is full, then begins outputting previous
; inputs. The output signals are sent to successors. Wait is 3 to
; match the transmission line response.

(defflavor delay
           ((wait 3)
            (signals nil))
           (basic-module)
  :initable-instance-variables
  (:special-instance-variables signals))

(defmethod (delay :process)
           (signal1)
  (cond ((= wait 0) (send-signal connect-list signal1))
        (T (if (null signals)
               (let ((s (send signal1 ':copy)))
                 (send s ':scale 0.)
                 (do ((i 0 (1+ i)))
                     ((= i wait))
                   (setq signals (cons s signals)))))
           (setq signals (cons signal1 signals))
           (send-signal connect-list (nth wait signals))
           (do ((i 1 (1+ i))
                (s signals (cdr s)))
               ((= i wait) (rplacd s nil)))
           nil)))

; --------------------- Testing utility routines ---------------------

(defun dump-basis (basis file)

  ; Dumps to file a representation of basis, in time and frequency
  ; domain. Assumes size, period, etc., are equal to system constants.

  (with-open-file (s file ':direction ':output)
    (let ((nu (send basis ':columns)))
      (do ((i 0 (1+ i)))
          ((= i nu))
        (let ((d (make-discrete-signal
                   (send basis ':column i) 0.0 *discrete-sample-period*)))
          (send d ':dump-time s (string-append
                                  "#Time sequence--vector #"
                                  (format nil "~A" i)))
```

```
          (send d ':dump-freq s (string-append
                                  ";Frequency transform--vector #"
                                  (format nil "~A" i)))))))))

(defun show-voice-weights ()

  ; Reads voice samples from input file and displays their
  ; concentrations to vectors in the input basis.

  (with-open-file (input "mc:users1;kalisk input" ':direction ':input)
    (do ((i 0 (1+ i)))
        ((= i 40.) T)
      (let ((signal1
              (get-voice-sample input
                                (fixr (* *n* (// *discrete-sample-period*
                                                *voice-sample-period*)))
                                *f1* *f2* *voice-sample-period*)))
        (send signal1 ':sample *discrete-sample-period*)
        (let ((weights (send (send signal1 ':sequence)
                             ':multiply-transposed
                             *prolate-spheroidal-basis*))
              (energy (// (send signal1 ':energy)
                          (send signal1 ':time-increment))))
          (do ((i 0 (1+ i))
               (e 0.0 (+ e
                         (// (^ (real-part (send weights
                                                 ':element i 0)) 2)
                             energy))))
              ((= i *nu*) (print e))
            nil))))))

(defun show-basis (basis f1 f2)

  ; Displays concentrations of vectors in basis to desired band

  (do ((i 0 (1+ i)))
      ((= i *nu*))
    (let ((d (make-discrete-signal
               (send basis ':column i) 0.0 *discrete-sample-period*)))
      (print (// (send d ':energy-band f1 f2) (send d ':energy))))))
```

```
;;; -*- Mode: LISP; Fonts: CPTFONT; Base: 10. -*-

; ----------------------- Matrix operations -----------------------
;
;       The matrix object represents the typical mathematical
;       notion of a two-dimensional set of values. There are
;       two types of matrix objects: the matrix, which may
;       be any size and contain any elements, and the symmetric-
;       matrix, which is restricted to be real, square, and
;       symmetric.
;
;       The usual operators and selectors are supported. Let
;       A be an m x n matrix. Indices i, j of A begin at 0, 0.
;
;       row i                   returns a 1 x n matrix containing
;                               the elements of row i
;       column j                returns an m x 1 matrix containing
;                               the elemenst of column j
;       element i j             returns the element at index i, j
;       multiply                returns the product B x A; prints
;        B &optional C          an error if sizes don't match; if
;                               C is specified, puts product in it
;       multiply-transposed     returns the product B' X A, where
;        B &optional C          B' is B transposed; if C is
;                               specified, puts product in it
;
;       The symmetric-matrix object has two other operations:
;
;       eigenvectors-and-
;        eigenvalues            returns an m x m matrix in which
;                               column i is an eigenvector, and
;                               an m-element array in which element
;                               i is the corresponding eigenvalue
;       eigenvectors-
;        ordered nu             returns a nu x m matrix in which
;                               column i is an eigenvector such
;                               that corresponding eigenvalues are
;                               arranged in decreasing order
;
;       Finally, the following output operators are supported:
;
;       dump stream doc         writes to stream a representation
;                               of matrix, headed by string doc
;
;       display stream          writes to stream a rectangular
;                               representation of matrix

(defflavor matrix
           (rows
            columns
            elements)
           ()
  :initable-instance-variables
  :gettable-instance-variables)

(defflavor symmetric-matrix
           (rows
            columns
            elements)
           (matrix)
```

```
  :initable-instance-variables
  :settable-instance-variables)

(defmethod (matrix :row)
           (i)
  (let ((row (make-array columns ':fill-pointer columns)))
    (do ((j 0 (1+ j)))
        ((= j columns) row)
      (aset (aref elements i j) row j))))

(defmethod (matrix :column)
           (j)
  (let ((column (make-array rows ':fill-pointer rows)))
    (do ((i 0 (1+ i)))
        ((= i rows) column)
      (aset (aref elements i j) column i))))

(defmethod (matrix :display)
           (&optional (stream terminal-io))
  (do ((i 0 (1+ i)))
      ((= i rows) T)
    (terpri stream)
    (do ((j 0 (1+ j)))
        ((= j columns) T)
      (format stream "~4,1,8,$" (aref elements i j)))))

(defmethod (matrix :dump)
           (stream
            &optional (documentation "#Matrix"))
  (princ documentation stream)
  (print rows stream) (princ " #rows" stream)
  (print columns stream) (princ " #columns" stream)
  (do ((i 0 (1+ i)))
      ((= i rows) (terpri stream))
    (do ((j 0 (1+ j)))
        ((= j columns) T)
      (print (aref elements i j) stream)
      (princ "#" stream) (princ i stream)
      (princ " " stream) (princ j stream))))

(defmethod (matrix :element)
           (i j)
  (aref elements i j))

(defmethod (matrix :multiply)
           (matrix2 &optional matrix3)
  (let ((rows2 (send matrix2 ':rows))
        (columns2 (send matrix2 ':columns))
        (elements2 (send matrix2 ':elements)))
    (let ((product
           (if matrix3 (send matrix3 ':elements)
                       (make-array (list rows2 columns)))))
      (do ((i 0 (1+ i)))
          ((= i rows2)
           (if matrix3 matrix3 (make-matrix product)))
        (do ((j 0 (1+ j)))
            ((= j columns) T)
          (do ((k 0 (1+ k))
               (sum 0.0
                    (add-complex
```

```
                    sum
                    (mult-complex (aref elements2 i k)
                                  (aref elements k j)))))
              ((= k columns2) (aset sum product i j))
            nil))))))

(defmethod (matrix :multiply-transposed)
           (matrix2 &optional matrix3)
  (let ((rows2 (send matrix2 ':rows))
        (columns2 (send matrix2 ':columns))
        (elements2 (send matrix2 ':elements)))
    (let ((product
            (if matrix3 (send matrix3 ':elements)
                        (make-array (list columns2 columns)))))
      (do ((i 0 (1+ i)))
          ((= i columns2)
           (if matrix3 matrix3 (make-matrix product)))
        (do ((j 0 (1+ j)))
            ((= j columns) T)
          (do ((k 0 (1+ k)))
              (sum 0.0
                   (add-complex
                     sum
                     (mult-complex (aref elements2 k i)
                                   (aref elements k j)))))
            ((= k rows2) (aset sum product i j))
          nil))))))

(defmethod (symmetric-matrix :eigenvectors-and-eigenvalues)
           ()
  (let ((E (make-array (list rows columns) ':initial-value 0.0))
        (val (make-array columns))
        (A (make-array (list rows columns))))

    ; Matrix A is reduced into eigenvalues; identity matrix E
    ; accumulates eigenvectors.

    (copy-array-contents elements A)
    (do ((j 0 (1+ j)))
        ((= j columns) T)
      (aset 1.0 E j j))
    (Jacobi columns 50. A 1.0e-10 1.0e-10 1.0e-5 E)

    ; Normalize columns of E and copy diagonal elements of A into val.

    (do ((j 0 (1+ j)))
        ((= j columns) (values (make-matrix E) val))
      (aset (aref A j j) val j)
      (let ((length
              (do ((i 0 (1+ i)))
                  (length 0.0 (+ length (^ (aref E i j) 2.))))
                  ((= i rows) (sqrt length)))))
        (do ((i 0 (1+ i)))
            ((= i rows) T)
          (aset (// (aref E i j) length) E i j))))))

(defmethod (symmetric-matrix :eigenvectors-ordered)
           (nu)
  (let ((index (make-array rows))
        (e-vecs (make-array (list rows nu)))
```

```
        (o-vals (make-array rows)))

    ; Index is used for sorting eigenvalues. O-vecs and o-vals hold
    ; the sorted versions of eigenvectors and eigenvalues.

    (do ((j 0 (1+ j)))
        ((= j rows) T)
      (aset j index j))
    (multiple-value-bind (vecs vals)
        (send self ':eigenvectors-and-eigenvalues)

      ; Sorting algorithm is essentially a bubble sort. Eigenvalues
      ; are accessed indirectly through index; elements are swapped
      ; by exchanging indices, to put them in increasing order.

      (do ((i (1- rows) (1- i)))
          ((= i 0) T)
        (do ((j 0 (1+ j)))
            ((= j i) T)
          (cond ((< (aref vals (aref index j))
                    (aref vals (aref index (1+ j))))
                 (swap index j (1+ j))))))

      ; O-vals and o-vecs are created from elements of vals and
      ; vecs accessed by indices.

      (do ((i 0 (1+ i)))
          ((= i rows))
        (aset (aref vals (aref index i)) o-vals i))
      (let ((elements-v (send vecs ':elements)))
        (do ((i 0 (1+ i)))
            ((= i rows) (values (make-matrix o-vecs) o-vals))
          (do ((j 0 (1+ j)))
              ((= j nu))
            (aset (aref elements-v i (aref index j)) o-vecs i j)))))))

; ------------------------ Jacobi algorithm -----------------------

; The Jacobi algorithm is used for computation of eigenvectors of a
; real symmetric matrix of elements of A. Diagonal elements of A
; become eigenvalues; corresponding columns of E become eigenvectors.
; A is an n x n matrix.
;
; The Jacobi algorithm is iterative, and during each pass rotations
; are applied to A to remove off-diagonal elements.  Kmax is the
; maximum number of iterations. E1 indicates the largest value whose
; arctangent can be approximated with 0.  E2 is the largest value of
; an off-diagonal element which should not be eliminated. The
; algorithm terminates when the sum of the squares of diagonal
; elements of A differs by less than the fraction e3.

(defun Jacobi (n kmax A e1 e2 e3 E)

  ; Terminate after kmax iterations, or when diagonal sums change
  ; by small amount.

  (do ((k 0 (1+ k))
       (sigma1 (diagonal-square-sum A n) (diagonal-square-sum A n))
       (sigma2 0.0 sigma1))
      ((or (= k kmax) (< (abs (- 1.0 (// sigma2 sigma1))) e3)) T)
```

```
; Move over elements below diagonal (same as those above, since
; matrix remains symmetric).

(do ((i 0 (1+ i)))
    ((= i (1- n)) T)
  (do ((j (1+ i) (1+ j)))
      ((= j n) T)

    ; Ignore element if already near 0.

    (cond ((> (abs (aref A i j)) e2)

           ; Compute, in a precise fashion, the cosine and sine of
           ; the required angle of rotation. The rotation uses as
           ; pivots the diagonal elements in the same row or
           ; column as the current element.

           (let* ((q (abs (- (aref A i i) (aref A j j))))
                  (p (if (< q e1) 0.0
                         (* 2.0 (aref A i j) q
                            (// (- (aref A i i) (aref A j j))))))
                  (cos-a (if (< q e1) 0.70710677
                             (sqrt
                               (* 0.5
                                  (+ 1.0
                                     (// q (sqrt (+ (^ p 2)
                                                    (^ q 2))))))))))
                  (sin-a (if (< q e1) 0.70710677
                             (// p (* 2.0 cos-a
                                      (sqrt (+ (^ p 2)
                                               (^ q 2)))))))))

             ; Apply rotation to E and A, but only in those
             ; columns and rows affected (that is, those
             ; containing the eliminated and pivot elements).

             (do ((l 0 (1+ l)))
                 ((= l n) T)
               (let ((eli (+ (* (aref E l i) cos-a)
                             (* (aref E l j) sin-a)))
                     (elj (- (* (aref E l i) sin-a)
                             (* (aref E l j) cos-a)))
                     (ali (+ (* (aref A l i) cos-a)
                             (* (aref A l j) sin-a)))
                     (alj (- (* (aref A l i) sin-a)
                             (* (aref A l j) cos-a))))
                 (aset eli E l i)
                 (aset elj E l j)
                 (aset ali A l i)
                 (aset alj A l j)))

             ; Apply rotation for element above diagonal in A.

             (do ((l 0 (1+ l)))
                 ((= l n) T)
               (let ((ail (+ (* (aref A i l) cos-a)
                             (* (aref A j l) sin-a)))
                     (ajl (- (* (aref A i l) sin-a)
                             (* (aref A j l) cos-a))))
```

```
                    (aset ail A i l)
                    (aset ajl A j l))))))))))

(defun diagonal-square-sum (A n)

; Effect: Returns sum of squares of n diagonal elements of matrix A.

  (do ((i 0 (1+ i))
       (sum 0.0 (+ sum (^ (aref A i i) 2))))
      ((= i n) sum)
    nil))
```

REFERENCES

Bell Telephone Laboratories (1971), Transmission Systems for Communication, rev. 4th ed.

Goldstine, H.H., F.J. Murray, J. von Neumann (1959), "The Jacobi method for real symmetric matrices," Journal of the ACM, 6, p. 59.

Holsinger, J.L. (1964), "Digital communication over fixed time-continuous channels with memory: with special application to telephone channels," Lincoln Laboratories Technical Report 366.

Landau, H.J., and H.O. Pollak (1962), "Prolate spheroidal wave functions, Fourier analysis, and uncertainty -- III," Bell System Tech. Journal, 41, no. 4.

Meixner, J., F.W. Schafke, G. Wolf (1980), "Mathieu functions and spheroidal functions and their mathematical foundations -- further studies," Lecture Notes in Mathematics, 837, Springer-Verlag, Berlin.

Nelson, R.E. (1976), "A guide to voice scramblers for law enforcement agencies," National Bureau of Standards Special Publication 480-8.

Rabiner, L.R., and R.W. Schafer (1978), Digital Processing of Speech Signals, Prentice-Hall Inc., Englewood Cliffs NJ.

Sedgewick, R. (1983), Algorithms, Addison-Wesley, Reading MA.

Slepian, D. (1978), "Prolate spheroidal wave functions, Fourier analysis, and uncertainty -- V: the discrete case," Bell System Tech. Journal, 57, no. 5.

Sloane, N.J.A. (1983), "Encrypting by random rotations," in "Cryptography," Lecture Notes in Computer Science, 149, Springer-Verlag, Berlin.

Strang, G. (1980), Linear Algebra and its Applications, Academic Press, New York.

van der Steen, A.J. (1983), "Timing of fast DFT algorithms," Signal Processing, 5, no. 6.

Wyner, A.D. (1979), "An analog scrambling scheme which does not expand bandwidth, Part I: discrete time," IEEE Trans. Inform. Theory, 25.

Wyner, A.D. (1979), "An analog scrambling scheme which does not expand bandwidth, Part II: continuous time," ibid.

Wyner, A.D. (1984), private communication.