NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF A DIGITAL COMMUNICATION SYSTEM USING QPSK MODULATION

by

Dilip B. Ghate

December, 1995

Thesis Advisor: Murali Tummala

Approved for public release; distribution is unlimited.

19960315 042

DTIC QUALITY INSPECTED 1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1995	Master's Thesis

4. TITLE AND SUBTITLE
IMPLEMENTATION OF A DIGITAL COMMUNICATION SYSTEM USING QPSK MODULATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Ghate, Dilip, B.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

With the advances in high speed, programmable digital signal processing (DSP) chips, modern communications links are using a combination of DSP techniques and digital communications methods to realize faster, reconfigurable, and modular systems. This thesis details the software implementation of a modern digital communication system combining various DSP functions, channel Forward Error Correcting (FEC) algorithms, and digital modulation methods. The digital modulation schemes considered here include both baseband and Quadrature Phase Shift Keying (QPSK) techniques. The proposed communication system will serve as a practical tool useful for simulating the transmission of any digital data. The various modules of the system include source encoders/decoders, data compression functions, channel encoders/decoders, and modulators/demodulators. Implementation consists of coding the various link functions in C and integrating them as a complete system. The results show the viability of a QPSK modulated digital communications link and point the direction of future research towards software radio.

14. SUBJECT TERMS	15. NUMBER OF PAGES
CELP, Convolutional codes, Viterbi decoding, baseband modulation, BSC, QPSK modulation, QPSK demodulation, AWGN channel,	99
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited.

IMPLEMENTATION OF A DIGITAL COMMUNICATION SYSTEM USING QPSK MODULATION

Dilip B. Ghate
Lieutenant, United States Navy
B.S., Rensselaer Polytechnic Institute, 1988

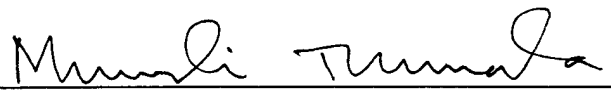
Submitted in partial fulfillment of the
requirements for the degree of

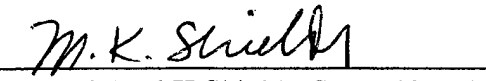
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

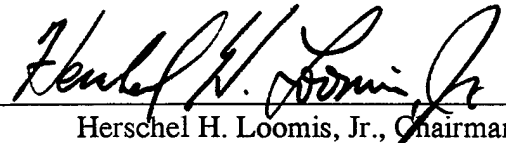
from the

NAVAL POSTGRADUATE SCHOOL
December 1995

Author: 
Dilip B. Ghate

Approved By: 
Murali Tummala, Thesis Advisor


Michael K.Shields, Second Reader


Herschel H. Loomis, Jr., Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

With the advances in high speed, programmable digital signal processing (DSP) chips, modern communications links are using a combination of DSP techniques and digital communications methods to realize faster, reconfigurable, and modular systems. This thesis details the software implementation of a modern digital communication system combining various DSP functions, channel Forward Error Correcting (FEC) algorithms, and digital modulation methods. The digital modulation schemes considered here include both baseband and Quadrature Phase Shift Keying (QPSK) techniques. The proposed communication system will serve as a practical tool useful for simulating the transmission of any digital data. The various modules of the system include source encoders/decoders, data compression functions, channel encoders/decoders, and modulators/demodulators. Implementation consists of coding the various link functions in C and integrating them as a complete system. The results show the viability of a QPSK modulated digital communications link and point the direction of future research towards software radio.

TABLE OF CONTENTS

I. INTRODUCTION

This thesis details the implementation in software of a modern digital communication link, combining various Digital Signal Procesing (DSP) functions, Forward Error Correction (FEC) channel coding algorithms, and digital modulation methods, mainly Quadrature Phase Shift Keying (QPSK). Software implementation is performed in the C++ and C programming languages and consists of separately coding and interfacing the various functions of the system. By modularizing the various functions which are performed on the data from source to destination, it becomes convenient to change individual sections of the link and model the effects of different transmission conditions on data as it is passed through the channel. The modules of the link include source encoders/decoders, channel encoders/decoders, modulators/demodulators, and channel effects.

The possible configurations of the link are numerous, so focus was maintained on a typical system that might be used in satellite communications. For this reason convolutional channel coders and QPSK modulation were chosen. Additionally, channel effects were primarily modeled as the combination of bandlimited additive white gaussian noise (AWGN) and signal attenuation. The results of this link on speech transmission show the various gains and tradeoffs that are realized as data passes through the entire system; the quantitative performance is measured by the probability of bit error, P_b, and the effect various signal-to-noise ratios have on this probability.

The bulk of the actual implementation was performed in C and C++ while most of the algorithm testing and system design was accomplished using MATLAB. This required writing MATLAB scripts and converting them to C. While the bulk of the operational routines and algorithms are written using C constructs, all input/output and file accesses are accomplished with C++ notation; therefore, all of the code has been compiled with a C++ compiler. Full conversion to C is possible with appropriate changes to the file operations.

The thesis is organized in the following manner: Chapter II details the digital communication model used in this work. Chapter III discusses the DSP methods involved, specifically source encoding/compression standards used for speech. Chapter IV outlines the channel coding methods involving FEC and Viterbi algorithms while Chapter V presents the digital communications techniques used including the baseband and QPSK modulation schemes. Overall implementation and results are reported in Chapter VI. Conclusions and areas for further development are contained in Chapter VII. Appendices A through E include supporting code and documentation.

II. DIGITAL COMMUNICATION MODEL

This chapter discusses the basic digital communication model which is the backbone of the work presented in the rest of this thesis. The model is broken into its constituent functions or modules, and each of these is in turn described in terms of its affects on the data and the system. Since this model comprises the entire system, both the source coding and communications techniques involved are briefly described. In chapters III through VI, these two areas will be covered in detail, and the specific algorithms and methods used in the software implementation will be addressed in detail.

A. SOURCE ENCODING AND DECODING

The basic digital communication model is depicted in Figure 2.1 below. The first three blocks of the diagram (source encoder, channel encoder, and modulator) together

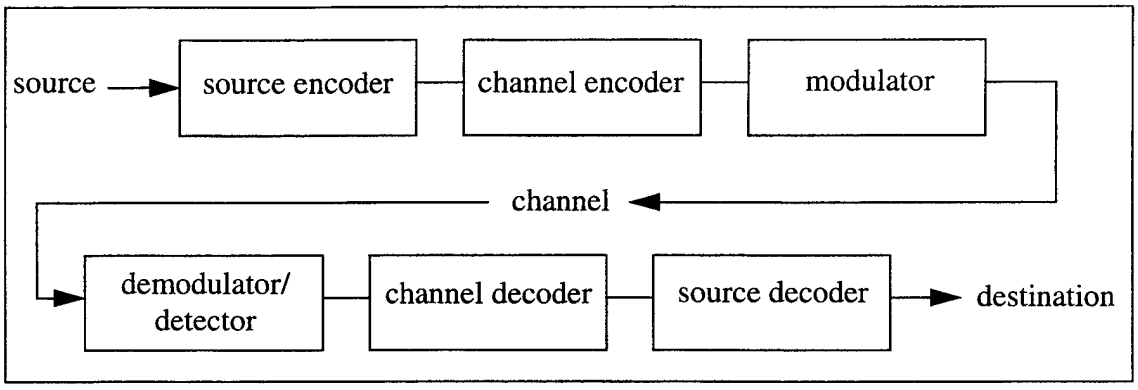


Figure 2.1: Digital Communication Model

comprise the transmitter. The source represents the message to be transmitted which includes speech, video, image, or text data among others. If the information has been acquired in analog form, it must be digitized prior to subsequent manipulation. This analog to digital conversion (ADC) is accomplished in the source encoder block.

The last three blocks consisting of detector/demodulator, channel decoder, and source decoder form the receiver. The destination represents the client waiting for the information. This might include a human or a storage device or another processing station. In any case, the source decoder's responsibility is to recover the information from the

channel decoder and to transform it into a form suitable for the destination. This transformation includes digital to analog conversion (DAC) if the destination is a human waiting to hear or view the infomation or if it is an analog storage device. If the destination is a digital storage device, the infomation will be kept in its digital state without DAC. [Ref. 1]

B. COMPRESSION

The source encoder block has the additional task of compressing the data. The essence of digital data compression is to represent a given set of data in a minimum number of bits such that the original signal can be retrieved from this compact representation with minimal loss of information. This is achieved by eliminating redundancies that exist in the original digital signal. By representing the data with fewer bits, the burdens on storage devices are reduced and transmission requirements are lessened thereby freeing storage space and bandwidth for other data. The effectiveness of a compression algorithm is measured by its compression ratio, C, defined as

$$C = \frac{b_i}{b_o}, \tag{1.1}$$

where b_i is the number of bits used before compression, and b_o is the number of bits used after compression.

There are two types of compression algorithms: lossless and lossy. As the name implies, lossless compression methods will produce a representation from which the original signal can be recovered exactly with no loss of data; however, the corresponding compression ratios are not very high. Lossy compression methods achieve higher compression ratios but permanently lose some of the information in the original signal; therefore, the recreated signal is not an identical replica of the original. If the signal is destined directly for human consumption, this is not harmful. For many speech and image signals, the human ear and eye either compensate for the discrepancies or cannot identify them; however, lossy methods are generally not acceptable if, after recovery, the signal

needs to undergo further processing. The source decoder block performs signal reconstruction from the compressed data after it is received from the channel decoder.

C. CHANNEL ENCODING AND DECODING

One of the advantages of digital communications over analog communications is its robustness during transmission. Due to the two-state nature of binary data (i.e,. either a 1 or a 0), it is not as susceptible to noise or distortion as analog data. While even the slightest noise will corrupt an analog signal, small amounts of noise will generally not be enough to change the state of a digital signal from 1 to 0 or vice versa and will in fact be 'ignored' at the receiver while the correct information is accurately recovered.

Nevertheless, larger amounts of noise and interference can cause a signal to be demodulated incorrectly resulting in a bitstream with errors at the destination. Unlike an analog system, a digital system can reduce the effect of noise by employing an error control mechanism which is used prior to modulation. The channel encoder performs this error control by systematically introducing redundancy into the information bitstream after it has been source encoded but prior to its transmission. This redundancy can then be used by the receiver to resolve errors that might occur during transmission due to noise or interference.

The channel decoder performs the task of decoding the received coded bitstream by means of a decoding algorithm tailored for the encoding scheme. Error control of this variety that allows a receiver to resolve errors in a bitstream by decoding redundant information introduced at the transmitter is known as Forward Error Correction (FEC). The price paid for employing FEC is the increased bit rate and complexity of the transmitter and receiver. The specifics of the FEC encoding and decoding algorithms used will be expanded upon in Chapter IV. [Ref. 2]

D. MODULATION

The digital modulator serves as an interface between the transmitter and the channel. It serves the purpose of mapping the binary digital information it receives into waveforms compatible with the channel. In baseband modulation, the output waveforms are simple voltage pulses which take predefined values correponding to a 1 or 0. However,

many channels, such as a satellite channel, are not suited for baseband communication and require the incoming data to be modulated to a higher frequency, referred to as the carrier frequency, so it can be converted to an electromagnetic wave that will propogate through space to its destination (e.g., a satellite or a ground station). This type of modulation, known as bandpass modulation, varies one of the following three parameters of the carrier frequency based on the the incoming digital bitstream: amplitude, frequency or phase. These modulation types are commonly known as Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK) and Phase Shift Keying (PSK), respectively. It is the last of these which is of primary interest and will be examined in more detail in Chapter V.

The digital detector/demodulator reverses the process and extracts the binary baseband information from the received modulated signal which has been subjected to noise, interference, loss, and other distortions. The demodulator produces a sequence of binary values which are estimates of the transmitted data and passes it on to the channel decoder. [Refs. 2 and 3]

E. CHANNEL EFFECTS

During transmission, the signal undergoes various degrading and distortion effects as it passes through the medium from the transmitter to the receiver. This medium is commonly referred to as the channel. Channel effects include, but are not limited to, noise, interference, linear and non-linear distortion and attenuation. These effects are contributed by a wide variety of sources including solar radiation, weather and signals from adjacent channels. But many of the prominent effects originate from the components in the receiver. While many of the effects can be greatly reduced by good system design, careful choice of filter parameters, and coordination of frequency spectrum usage with other users, noise and attenuation generally cannot be avoided and are the largest contributors to signal distortion.

In digital communication systems, a common quantity used to determinine whether a signal will be detected correctly is the ratio of energy per bit to spectral noise power density, E_b/N_o, measured at the detector. The higher the E_b, the lower the resulting bit error rate (BER), or the probability of bit error, P_b. Unfortunately, a high E_b demands greater

power consumption at the transmitter; in some cases, it may be unfeasible to obtain a high E_b due to transmitter size or power limitations as in the case of satellite transmission. Noise effects and further study of the parameter E_b/N_o will be covered in Chapters V and VI. [Ref. 2 and 3]

The digital communication system described consists of an ordered grouping of various modules which operate on an input data sequence. In practice, these modules or resources are not dedicated to a single source/destination, but they are shared by multiple sources and their destinations to achieve optimum utilization. In a digital system, the transmission bit rate is an important system resource. A given information source of bandwidth B, sampled at $2B$ samples/second using q bits per sample results in a data rate, R, of $2Bq$ bits per second. With a compression ratio C, the data rate from the source encoder is $R_s = R/C$ bits per second. Channel coding by a factor n leads to a coded data rate of $R_c = R_s n$ bits per second; R_c is the system transmission bit rate. These bits are then used by the modulator to form the transmission waveforms which have to be accomodated within the available bandwidth. At the receiver these steps are performed in the reverse order to recover the information sequence.

III. SOURCE CODING METHODS

In the digital communication system model described previously, the *source encoder* is responsible for producing the digital information which will be manipulated by the remainder of the system. After the digital signal is acquired from the analog information, the *source encoder* subjects it to a wide range of processing functions, the goals of which are to compactly represent the information. Speech, image, and textual information each have their own unique characteristics that require different source encoding techniques. Depending on the information source, different digital signal processing functions are implemented to remove the redundancies inherent in the given signal. The specifics of the speech compression techniques used in this thesis are detailed below.

A. SPEECH COMPRESSION

Since the frequency content of spoken language is confined to frequencies under 4000 Hz, it is reasonable to use a sampling frequency of 8000 Hz [Ref. 4]. Using 16 bit linear pulse code modulation (PCM) as the quantization method results in a bit rate of 128 kbps. Subsequent analysis, coding, and compression of speech are performed on segments or frames of 20 to 30 ms duration.

There are two broad categories of speech coding/compression. Both categories are concerned with representing the speech with the minimum number of applicable parameters while also allowing the speech to be intelligibly reproduced; both are lossy in nature. The first category deals with waveform coders which manipulate quantities in the speech signal's frequency representation. Typical analysis tools of waveform coders are the discrete fourier transform (DFT) and the discrete wavelet transform (DWT), both of which transform the time signal to its frequency domain representation. In this case, compression might potentially be achieved by retaining the frequency components with the largest magnitudes.

The second category of speech compression deals with voice coders, or vocoders for short. Vocoders attempt to represent speech as the output of a linear system driven by

either periodic or random excitation sequences as shown in Figure 3.1.

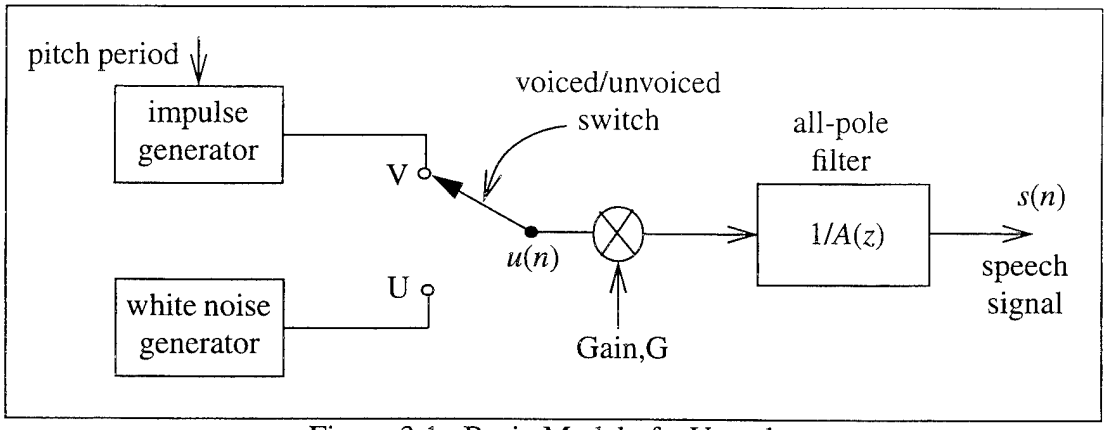


Figure 3.1 Basic Model of a Vocoder

A periodic impulse train or a white noise sequence, representing voiced or unvoiced speech, drives an all-pole digital filter to produce the speech output. The all-pole filter digital filter models the vocal tract. [Ref. 4]

Additionally, estimates of the pitch period and gain parameters are necessary for accurate reproduction of the speech. Due to the slowly changing shape of the vocal tract over time, vocoders successfully reproduce speech by modeling the vocal tract independently for each frame of speech and driving it by an estimate of a separate input excitation sequence for that frame. Most vocoders differ in performance principally based on their methods of estimating the excitation sequences.

Linear predictive coding (LPC) or autoregressive (AR) modeling of speech is used to obtain the parameters of the all-pole model for a given frame of speech:

$$s(n) = Gu(n) - a_1 s(n-1) - a_2 s(n-2) - \ldots - a_P s(n-P) \qquad (3.1)$$

where a_i (i = 1, 2,..., P) are the coefficients of the the all pole filter, and P is the order of the filter. A widely used notation used to describe this AR model is given by LPC-P with P = 10 being a common choice as in the Federal Standard FS-1015 vocoder scheme. Below, a specific refinement of the LPC technique known as code-excited linear prediction (CELP) is detailed.

B. CODE-EXCITED LINEAR PREDICTION (CELP)

Although the data rate of plain LPC coders is low, the speech reproduction, while generally intelligible, has a metallic quality, and the vocoder artifacts are readily apparent in the unnatural characteristics of the sound. The reason for this is because this algorithm does not attempt to encode the excitation of the source with a high degree of accuracy. The CELP algorithm attempts to resolve this issue while still maintaining a low data rate.

Speech frames in CELP are 30 ms in duration, corresponding to 240 samples per frame using a sampling frequency of 8000 Hz. They are further partitioned into four 7.5 ms subframes of 60 samples each. The bulk of the speech analysis/synthesis is performed over each subframe.

The CELP algorithm uses two indexed codebooks and three lookup tables to access excitation sequences, gain parameters, and filter parameters. The two excitation sequences are scaled and summed to form the input excitation to a digital filter created from the LPC filter parameters. The codebooks consist of sequences which are each 60 samples long, corresponding to the length of a subframe.

Figure 3.2 shows a schematic diagram of the CELP analyzer/coder. The stochastic codebook is fixed containing 512 zero-mean Gaussian sequences. The adaptive codebook has 256 sequences formed from the input sequences to the digital filter and updated every two subframes. A code from the stochastic codebook is scaled and summed with a gain-scaled code from the adaptive codebook. The result is used as the input excitation sequence to an LPC synthesis filter. The output of the filter is compared to the actual speech signal, and the weighted error between the two is compared to the weighted errors produced by using all of the other codewords in the two codebooks. The codebook indices of the two codewords (one each from the stochastic and adaptive codebooks), along with their respective gains, which minimize the error are then coded for transmission along with the synthesis filter (LPC) parameters. Because the coder passes each of the adaptive and

stochastic codewords through the synthesis filter before selecting the optimal codewords, CELP is referred to as an analysis-by-synthesis technique. [Ref. 4 and 5]

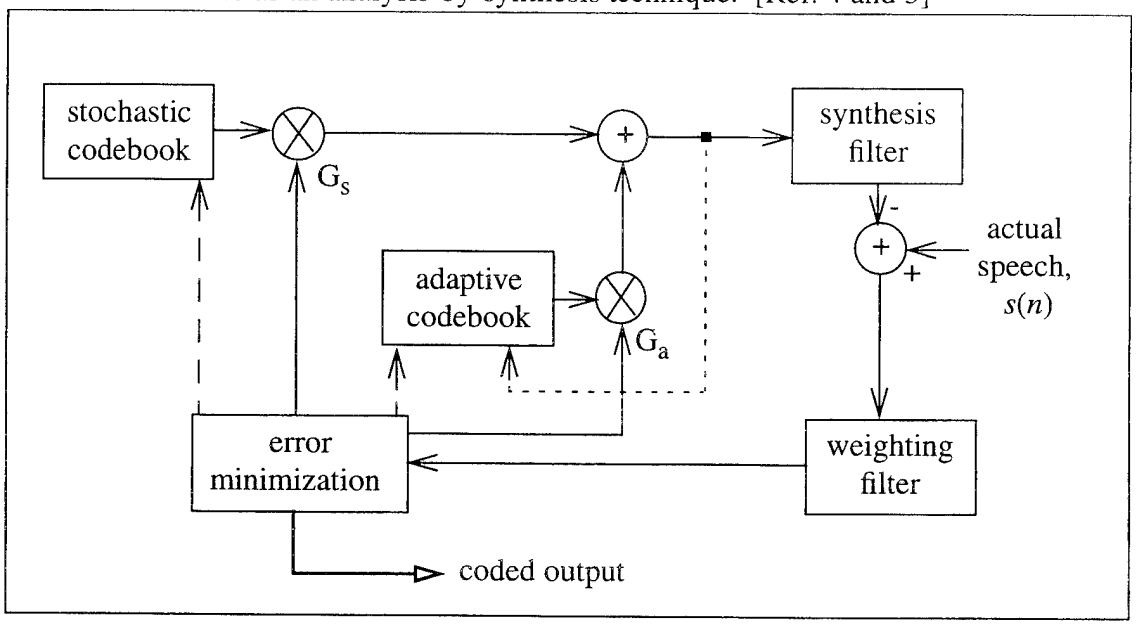


Figure 3.2 CELP Analyzer

1. Speech Synthesis

Figure 3.3 provides a more detailed view of the CELP synthesizer. Indices i_s and i_a identify the codewords used to form the input excitation sequence, and the gain parameters G_s and G_a scale the codewords before their sum is passed to the linear prediction filter

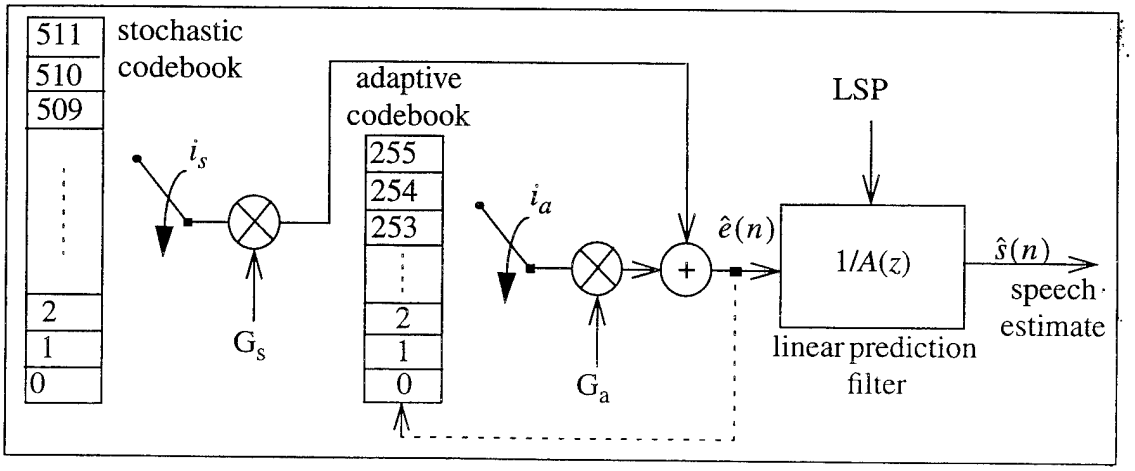


Figure 3.3 CELP Synthesizer. After [Ref. 6]

which is formed with line spectrum pairs (LSPs). The coded CELP frame includes the indices of the codebook entries, their gain parameters, and the LSP parameters.

The adaptive codebook entries are 60 samples long and are used to estimate the pitch period. The codebook consists of a 147 stage shift register. On odd subframes, the input excitation sequence used by the prediction filter is fed into the shift register, and the values on the other side are shifted out and discarded 60 samples at a time. The shift register values are used to form the codebook for use on the odd and the next even subframes; therefore, the synthesis filter excitation sequences used on even subframes are not sent to the register and are discarded. Of the total 256 sequences in the codebook, 128 represent integer delay which can be accessed directly from the shift register, and the remaining 128 represent non-integer delay which are formed by interpolation from the integer delay sequences. [Ref. 5]

As implied in the above description, synthesis of the speech is carried out with a different filter excitation sequence at every subframe. A coded CELP frame contains four sets of indices (one per subframe) corresponding to the codebook and gain values for each of its four subframes. During synthesis at the destination, the indices of the optimal codewords are used to reference the correct code sequences. Similarly, the indices of their respective gains are used to access the scaling parameters prior to use by the prediction filter. Both gain parameters are found in two separate fixed indexed codebooks of 32 entries each. [Ref. 5]

2. Linear Prediction Filter

The LPC prediction filter is a tenth-order filter derived using standard linear predictive techniques. A 30 ms Hamming window is used to window the data and the filter coefficients are found by the autocorrelation method [Ref 7]. Instead of transmitting the filter coefficients or the pole locations of the prediction filter, the corresponding line spectrum pairs (LSP) of each of the poles are encoded and transmitted. The reason for this is that the LSPs provide efficient channel error resilient coding and can also be transmitted using fewer bits than the corresponding poles or filter coefficients [Ref. 5]. The description on LSPs roughly follows the presentation given in Deller and Frerking [Refs. 4 and 5].

The poles of the prediction filter, $1/A(z)$, are the roots of the polynomial

$$A(z) = 1 - \sum_{i=1}^{N} a_i z^{-i} \tag{3.2}$$

where a_i are the LPC parameters. The idea behind LSPs is to use $A(z)$ to construct two functions

$$P(z) = A(z) - z^{-(N+1)} A(z^{-1}) \tag{3.3}$$

and

$$Q(z) = A(z) + z^{-(N+1)} A(z^{-1}) \tag{3.4}$$

where the orders of $P(z)$ and $Q(z)$ are N+1. Then $A(z)$ can be recovered from $P(z)$ and $Q(z)$ by the relation

$$A(z) = \frac{1}{2}[P(z) + Q(z)]. \tag{3.5}$$

Polynomials $P(z)$ and $Q(z)$ can be expressed as

$$P(z) = (1 - z^{-1}) \prod_{k=1}^{N/2} \left(1 - e^{j2\pi \frac{f_{kp}}{f_s}} z^{-1}\right)\left(1 - e^{-j2\pi \frac{f_{kp}}{f_s}} z^{-1}\right) \tag{3.6}$$

and

$$Q(z) = (1 + z^{-1}) \prod_{k=1}^{N/2} \left(1 - e^{j2\pi \frac{f_{kq}}{f_s}} z^{-1}\right)\left(1 - e^{-j2\pi \frac{f_{kq}}{f_s}} z^{-1}\right), \tag{3.7}$$

where f_s is the sampling frequency (8000 Hz). Other than the real zeros at $z = 1$ for $P(z)$ and at $z = -1$ for $Q(z)$, the remaining N zeros each of $P(z)$ and $Q(z)$ are complex conjugate pairs which lie on the unit circle and can be represented by the real frequencies f_{kp} and f_{kq}, respectively, in equations (3.6) and (3.7). Furthermore, these complex conjugate pairs are

interleaved as shown in Figure 3.4, so each zero of $A(z)$ corresponds to a pair of zeros, one

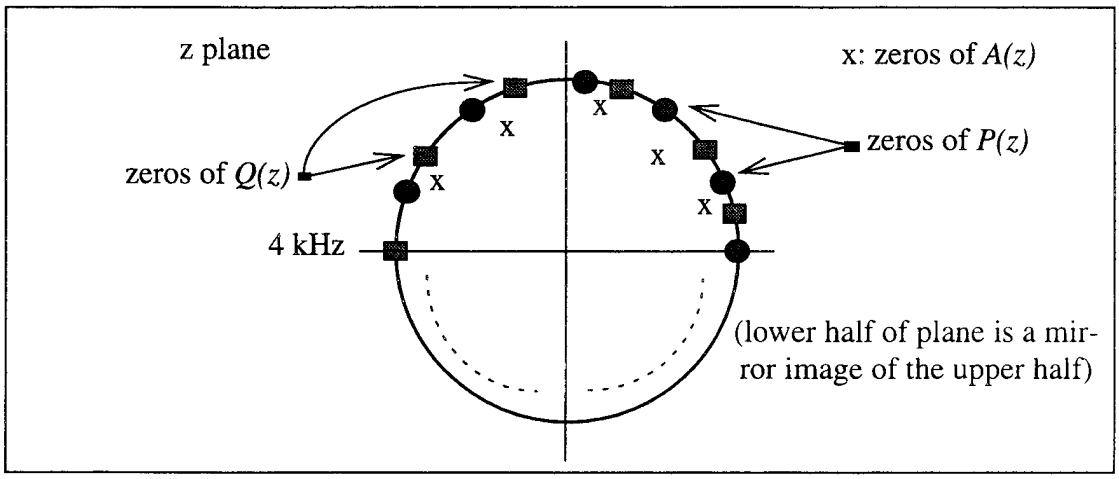


Figure 3.4 Relation Between the Zeros of $A(z)$ and its LSPs

each from $P(z)$ and $Q(z)$. The closer a pair of zeros of $P(z)$ and $Q(z)$ are to each other, the closer the corresponding zero of $A(z)$ is to the unit circle.

To more accurately reflect the time changing nature of the vocal tract, the 30 ms frame over which the LPC parameters are computed is displaced from the 30 ms speech analysis frame which determines the input excitation parameters as shown in Figure 3.5.

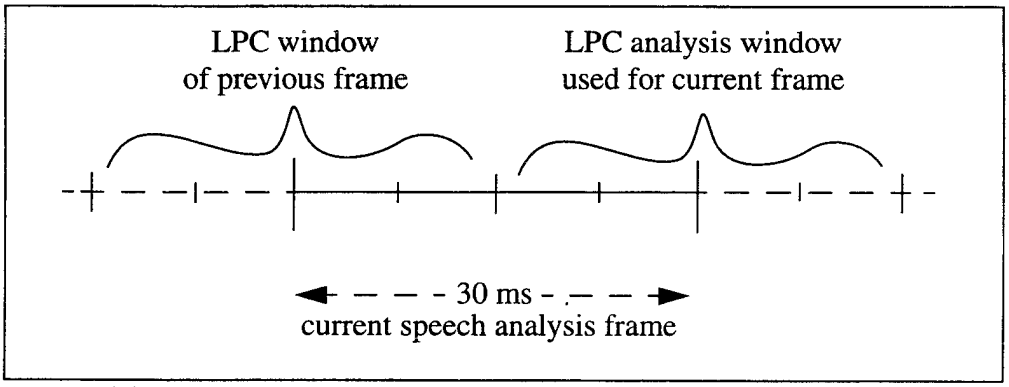


Figure 3.5 CELP Speech Analysis and LPC Analysis Frames

Specifically, the current LPC frame straddles the last two subframes (15 ms) of the current analysis window and the first two subframes (15 ms) of the next analysis window.

In the CELP algorithm, ten fixed sets of LSPs (corresponding to ten different prediction filters) are available in a lookup table during analysis and synthesis (see [Ref. 5]

for the lookup table values), and the closest set is chosen for a given solution of the prediction filter. This is done for coding efficiency; however, once the values of the LSPs have been obtained from the lookup table, they are adjusted throughout the CELP analysis and synthesis routines. Just as the input excitation sequence, gain, pitch period, and pitch gain are adjusted every subframe, the LSPs are interpolated each subframe by different weighting factors provided in Table 3.1. For example, if LSP1 of the previous frame is 420 Hz and LSP1 of the current frame is 280 Hz, then LSP1 for subframe 1 (of the current frame) is: 7/8(420 Hz) + 1/8(280 Hz) = 402.5 Hz. LSP interpolation every subframe causes the corresponding prediction filter to change every 7.5 ms thereby producing a smoother representation of the changing filter characteristics of the vocal tract compared to the relatively abrupt characteristics produced by non-overlapped, uninterpolated 30 ms windows.

TABLE 3.1: Interpolation Factors for LSPs

Subframe	Previous Frame	Current Frame
1	7/8	1/8
2	5/8	3/8
3	3/8	5/8
4	1/8	7/8

3. Speech Analysis

The analysis stage of CELP comprises the synthesizer described above along with the weighting filter and error minimization function as shown previously in Figure 3.2. After a pair of adaptive and stochastic codewords have been scaled, summed, and filtered with the prediction filter, the result is compared to the actual speech. The difference between the two represents the error and is sent to the error weighting filter, also know as the perceptual weighting filter. This filter modifies the error signal by emphasizing the areas of the spectrum to which the human ear is most sensitive. After perceptual weighting

of the error, the filter output is squared and summed to yield the error energy for the given stochastic and adaptive codebook pair.

The general procedure is to first pass every adaptive codeword (pitch period) and gain (pitch gain) through the synthesis filter to find the best scaled adaptive codeword. Once this has been found, each entry in the fixed codebook is filtered while carrying out a simultaneous search for the corresponding gain. [Ref. 5]

4. Compression Results

The CELP coder produces a 144 bit word for each 30 ms analysis frame of speech. This results in a total bit rate of 4800 bps or a compression ratio of approximately 27:1. The exact bit allocation per frame can be found in the federal standard [Ref. 6].

Although CELP has a bit rate twice that of the LPC-10 algorithm, it provides much better intelligibility. The improvement in performance is due to the dynamic use of the gain-scaled stochastic and adaptive codebooks in determining the input excitation sequences to the linear prediction filter. However, this increased performance is at the cost of increased computational load. The exhaustive search of the two codebooks and gain value lookup tables, and the interpolation routines for the adaptive codebook and LSPs combine to produce approximately an order of magnitude increase in computation over LPC-10 [Ref. 4]. The Federal Standard [Ref. 6] allows for the adjustment of the codebook sizes and of the interpolation routines used for the adaptive codebooks to reduce computational overhead, but this comes at the cost of reduced accuracy in the reproduced speech.

C. CELP VARIATIONS

There are two important variations of the CELP method, namely, the low delay CELP (LD-CELP) and the vector sum excited linear prediction (VSELP).

1. LD-CELP

As the name implies, LD-CELP attempts to reduce the delay incurred by the heavy computational load of the standard CELP routine. The main differences of this method with the standard CELP are the addition of a tenth-order, backward-adaptive linear gain

predictor which scales the excitation sequence and the elimination of the pitch estimation process, thereby removing the adaptive codebook/gain functions. Without the adaptive codebook, LD-CELP uses a 50th order prediction filter to compensate for the loss of pitch information. Additional differences include more frequent updates of the LPC coefficients (every 2.5 ms), significantly shorter excitation sequences (5 samples versus 60), and smaller frame/subframe lengths (10 ms/2.5 ms). The result of these modifications is an output data rate of 16000 bps with one-way delay of approximately 2 ms compared to other non-CELP based vocoders operating at the same data rate but with delays on the order of 20-40 ms [Ref. 4]. For full details on the parameters of LD-CELP and its implementation, the reader is referred to the ITU-T recommendation G.728.

2. VSELP

VSELP is an important variation of CELP because it is now the standard for North American digital cellular phone systems as defined in the IS-54 standard. The encoder uses the sum of three scale-adjusted sequences to form the input to a tenth-order synthesis filter. Each of the three codebooks contains 128 sequences that are 40 samples long. An analysis frame length of 20 ms is used, and the corresponding subframes are 5 ms each. Two of the codebooks are formed from two separate sets of seven basis codewords, and together they form an excitation sequence similar to that produced by the stochastic codebook in CELP; however, unlike CELP, the seven basis codewords of each codebook are optimized over a training database to minimize the perceptually weighted error [Ref. 4]. The third codebook plays the role of the adaptive codebook in CELP and is formed from the summed sequence input to the synthesis filter, and during every subframe its entries are correlated to the actual speech signal to yield the best pitch estimate. The output data rate of VSELP is 8000 bps. [Ref. 4]

D. IMPLEMENTATION

The standard CELP routine used in this thesis is based on FS-1016, and the code was developed by AT&T but available free as public domain software. It was modified by the author to compile and run on Sun SPARC workstations. The complete source code

listings and a compiled routine are now stored in the /tools_res directory of the ECE Department computer system.

The compiled encoding routine accepts only 16-bit linear PCM speech sampled at f_s = 8000 Hz and outputs the 144 bit long words in packed hexadecimal format (36 hex characters per frame). Since SPARC workstations process speech using the Sun '.au' format which is 8 bit μ-law compressed data, all speech files recorded on SPARCstations were converted to 16 bit linear PCM prior to usage by means of public domain conversion routines released by Sun Microsystems, Inc. Similarly, the decoding routine output was converted back to 8 bit μ-law data.

All intermediate processing from source encoder output to source decoder input was performed in binary, so the hexadecimal output from the CELP coder was converted to unpacked unipolar binary format (0 or 1) with the routine **hex2bin.C,** and the reverse process was accomplished with **bin2hex.C**. Both routines are included in Appendix A.

The CELP algorithm provides an effective means of speech reproduction while maintaining a high compression ratio with a corresponding low bit rate. Due to the growing use of CELP based coding methods in commercial systems, it is useful to study these schemes. The CELP code is available as public domain software which enables a study of CELP and its performance in conjunction with a digital communication system.

IV. CHANNEL CODING

Unlike source coding which seeks to remove redundant information in a signal, the goal of channel coding is the opposite in the sense that it adds redundancy to a bit stream which allows the detector at the receiver to detect and/or correct errors which might have been introduced during transmission. As mentioned in Chapter II, we are concerned specifically with FEC codes.

A parameter used to define a FEC code is its code rate, r, where

$$r = \frac{k}{n} \tag{4.1}$$

where k represents the number of bits into the channel encoder, n is the number of bits out of the encoder, and $k < n$. The code rate, equivalently represented using the notation (n, k), is a measure of the information contained per codebit. Large values of n relative to k are better for error-free coding, but the cost is in increased transmission bandwidth requirements. The two most common types of FEC codes encountered in communications systems are block codes and convolutional codes. [Ref. 1]

For block codes, the encoder partitions a given input information sequence into k-bit words and represents each input word by an n-bit codeword, where the codeword is often the input word appended with $n - k$ bits. If an input message sequence is partitioned into words that are k bits long, there are a total of 2^k possible information words and 2^k corresponding codewords. These codewords are designed to meet some predetermined error-correcting requirements and are fixed after they have been computed. Encoding is just a mapping of an information word to its respective codeword. At the decoder, the received codeword, which may have been altered due to noise, is mapped to the codeword with which it has the least difference followed by recovery of the corresponding information word. For more detailed information on block codes see Gibson [Ref. 1] and Roden [Ref. 8].

Convolutional codes form the second major category of FEC codes but are encoded and decoded in a very different manner than block codes. Today they are used in a variety of communication links, such as modern satellite communication systems. For this reason, the remainder of this chapter will deal exclusively with the encoding, decoding, and implementation of convolutional codes. The development will be limited to rate $1/n$ codes with specific attention focused on $r = 1/2$ codes because they are a good representation of convolutional codes.

A. CONVOLUTIONAL CODES

For $(n,1)$ convolutional codes, each bit of the information sequence into the encoder results in an output of n bits. However, unlike block codes, the relationship between information bits and output bits is not a simple one-to-one mapping. In fact, each input information bit is 'convolved' with K-1 other information bits to form the output n-bit sequence. The value K is known as the constraint length of the code and is directly related to its encoding and decoding complexity as described below in a brief explanation of the encoding process.

For each time step, an incoming bit is stored in a K stage shift register, and bits at predetermined locations in the register are passed to n modulo-2 adders to yield the n output bits. Each input bit enters the first stage of the register, and the K bits already in the register are each shifted over one stage with the last bit being discarded from the last stage. The n output bits produced by the entry of each input bit have a dependency on the preceding K-1 bits. Similarly, since it is involved in the encoding of K-1 input bits in addition to itself, each input bit is encoded in nK output bits. It is in this relationship that convolutional coding derives its power. For larger values of K, the dependencies among the bits increase, and the ability to correct more errors rises correspondingly. But the complexity of the encoder and especially of the decoder also becomes greater.

Shown in Figure 4.1 is the schematic for a (2,1) encoder with constraint length K = 3 which will serve as the model for the remainder of the development of convolutional coding. In the coder shown, the n = 2 output bits are formed by modulo-2 addition of the bits in stages one and three and the addition of bits in stages one, two, and three of the shift

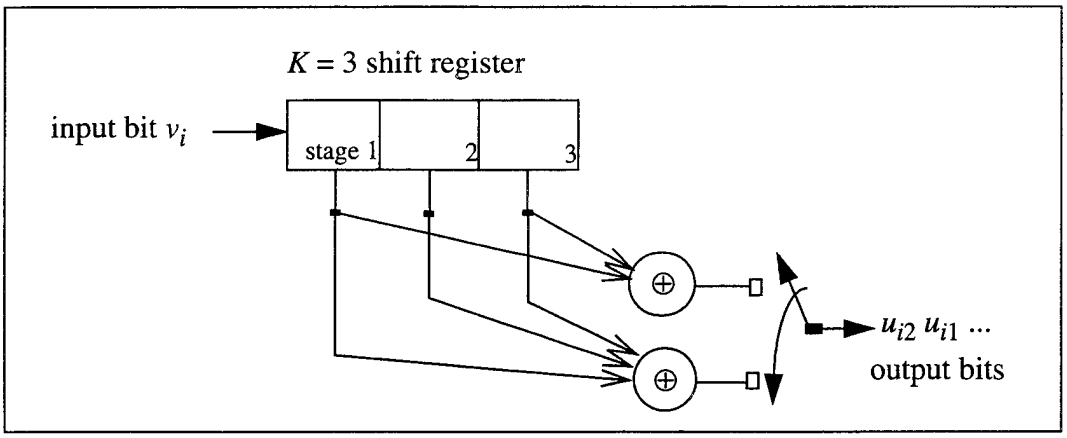


Figure 4.1 $K = 3$, $r = 1/2$ Convolutional Code Encoder

register. For example, consider the input sequence given by

$$\mathbf{v} = 11011\ldots.$$

If the left-most bit is the earliest in time and enters the shift register first, then assuming the register is empty (i.e., filled with zeros) at the beginning of the encoding process, the first two output bits, u_{11} and u_{12}, produced by the coder by input bit $v_1 = 1$ would be

$$u_{11} = 1 \oplus 0 = 1$$

$$u_{12} = 1 \oplus 0 \oplus 0 = 1.$$

The $\oplus$ symbol represents modulo-2 addition. The rest of the output sequence is:

$$\mathbf{u} = 10\ 10\ 00\ 10\ \ldots$$

Each pair of output bits, $u_{i1}\ u_{i2}$, can be more conveniently written as codesymbol $\mathbf{u}_i$.

It is also common to identify the coder by its code vectors. These length K vectors identify the bit locations in the shift regitster which are used by the n modulo-2 adders; therefore, there are n code vectors which represent the coding scheme. For the example

case, the code vectors are $\mathbf{c}_1$ = [1 0 1] representing stages one and three, and $\mathbf{c}_2$ = [1 1 1] for all three stages.

1. State Diagram

To assist in analyzing the code and determining its output for a given input sequence, there are various representations of convolutional codes which are helpful. These include polynomial representations, state tables, code tree, state diagrams, and trellis diagrams. All of these contain the same information but they each provide different insights into the workings of the code. For detailed information on all of the descriptions and their mathematical foundations, see Sklar [Ref. 2]. We will briefly focus on state and trellis diagrams.

Figure 4.2 shows the state diagram for the coder introduced in the previous section.

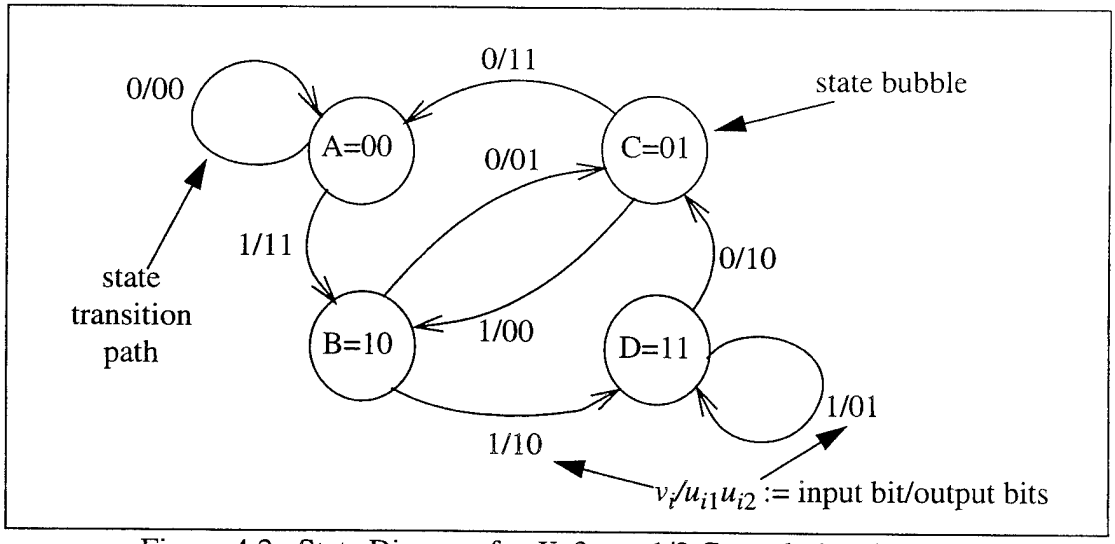


Figure 4.2 State Diagram for K=3, r = 1/2 Convolutional Coder

There are four states associated with the coder and represented by the state bubbles lettered A through D. Each state is uniquely identified by the two bits in the first two stages of the shift register: since the input data is binary, there are 2^2 states. In general, for an $(n,1)$ convolutional coder of constraint length K, there are 2^{K-1} states. The state transition paths show the state changes that occur for a given input bit, v_i, along with the correponding output bits,

$u_{i1}u_{i2}$, produced during the transition. Given the initial state of the coder, the coded output for an input data sequence can be easily determined by moving from state bubble to state bubble along the appropriate state transition path for each input bit (either 0 or 1) and reading the associated output bits. The state diagram representation can also be put in tabular form which is more manageable as K gets larger.

2. Trellis Diagram

Another insightful representation of convolutional codes is the trellis diagram. The trellis diagram in Figure 4.3 can be quickly obtained from the state diagram shown above.

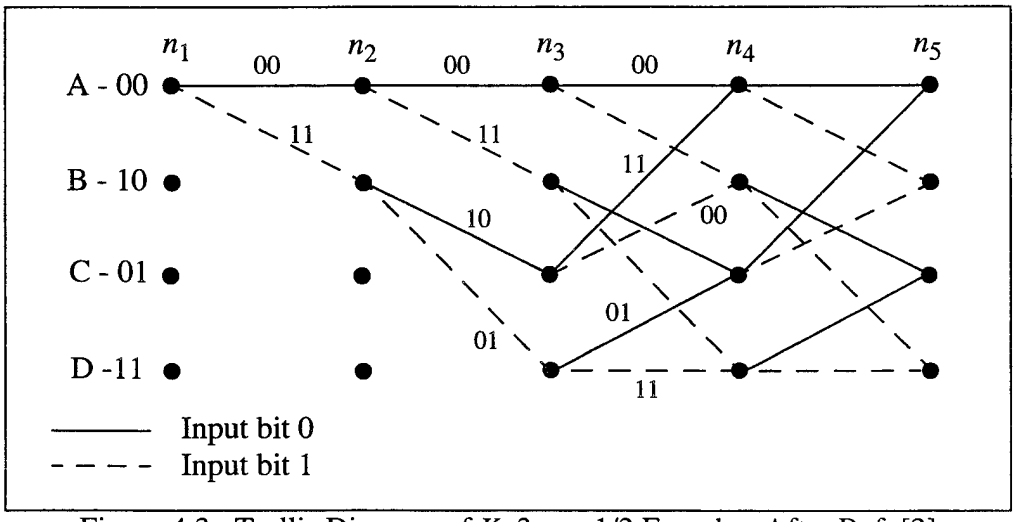


Figure 4.3 Trellis Diagram of K=3, r = 1/2 Encoder. After Ref. [2]

In the trellis representation, states A through D are represented by four nodes and all of the possible state transitions are shown beginning at some initial state (which is at A = 00 in this case). As a new bit is input to the coder at each time sample, n_i, the trellis depicts the possible progression of the codeword from state to state where solid lines represent an input bit of 1 and dashed lines indicate a 0 input bit. In other words, given the initial state of the coder, Figure 4.3 shows all of the possible paths through the trellis corresponding to all of the possible input sequences. It is clear that a repetetive pattern emerges a few time steps from the initial condition, and it continues forever. The pattern is merely the 'unwrapped' state transition diagram of Figure 4.2 where each state transition path between state bubbles

in Figure 4.1 is replaced by a trellis branch between the corresponding nodes. The trellis representation is a very important tool when implementing the Viterbi decoding algorithm to be described later.

3. Systematic vs. Nonsystematic Codes

A code is systematic if the encoder forms the output codeword by appending $n - k$ bits to the input information word. In the case of block codes, any non-systematic code can be reduced to a systematic form and still keep its designed error correcting abilities. The performance of convolutional codes depends on its *free distance,* d_f. The d_f is a measure of the code's error correcting capability such that a code with a larger d_f will be able to correct more errors. The number of errors a code can correct is given by

$$t = \left\lfloor \frac{d_f - 1}{2} \right\rfloor \tag{4.2}$$

where the $\lfloor x \rfloor$ operation indicates the integer closest to but not greater than the argument x [Ref. 2]. For convolutional codes, the non-systematic version will generally have a larger d_f than its systematic version and will be able to correct more errors. In the case of the $K = 3$, $r = 1/2$ coder, the codevectors $\mathbf{c}_1$ and $\mathbf{c}_2$ listed earlier are the non-systematic codevectors which yield a d_f of 5 and $t = 2$. More detailed information concerning systematic codes and calculations of free distance can be found in Sklar [Ref. 2] and Riccharia [Ref. 9].

B. VITERBI ALGORITHM

In general, there can be errors in the received data stream. For the correct code sequence to be recovered, these errors need to be resolved. Channel decoding is performed at the receiver after demodulation and prior to source decoding to attempt to resolve these errors. Convolutional decoding involves uncovering the transmitted message sequence by using the inherent coupling of information between bits introduced at the encoder to remove the errors. Because the size of the transmitted code sequence is unknown to the receiver, it must allocate enough memory resources to store and decode large sequences. If the initial state of the coder is not known, the decoder must also keep track of all of the

possible code sequences in order to perform an accurate decoding operation. The best decoding schemes attempt to quickly locate and update the most likely sequences while quickly discarding the least likely ones. Two of the more commonly used decoding algorithms are the sequential decoding and Viterbi decoding algorithms of which only the Viterbi method will be discussed here [Refs. 2 and 7].

1. Maximum Likelihood Decoding

If every output data sequence produced by the channel encoder is equally likely and if the probability of bit error, P_b, is the same and independent for each transmitted 0 or 1 in the sequence, then a maximum likelihood decoding procedure can be used by the channel decoder [Ref. 2]. The decoding procedure involves finding the channel encoder output sequence, $\boldsymbol{U}^{m_0}$, which satisfies the following relationship:

$$P\left(\mathbf{u_r}\middle|\boldsymbol{U}^{m_0}\right) = max\left\{P(\mathbf{u_r}|\boldsymbol{U}^{m})\right\}, \quad \text{for all } \boldsymbol{U}^{m} \tag{4.3}$$

where $\mathbf{u_r}$ is the received code sequence, $\boldsymbol{U}^{m}$ is one of the possible message sequences, and $P(\mathbf{u_r}|\boldsymbol{U}^{m})$ represents the conditional probability of receiving the code sequence $\mathbf{u_r}$ given the sequence $\boldsymbol{U}^{m}$. Since the solution involves a complete search over all possible message code sequences, the answer represents the most likely transmitted sequence. The conditional probability for each message sequence can also be calculated based on the conditional probability of occurence of each bit in $\mathbf{u_r}$ given the bit for that time step in the correponding message sequence,

$$P(\mathbf{u_r}|\boldsymbol{U}^{m}) = \prod_{i=1}^{\infty} P(\mathrm{u}_{ri}|U_i^{m}) \tag{4.4}$$

where u_{ri} and U_i are the ith bits in the received sequence and the reference message sequence, respectively. Sometimes it is useful to take the logarithm of this result in order to simplify the calculation to a series of addition operations. See Therrien [Ref. 7] for more details.

2. Viterbi Decoding Algorithm

The biggest practical constraint imposed by the maximum likelihood decoding method is that for long message sequences produced by large constraint coders, the amount of memory and time required to store and calculate all of the conditional probabilities for all of the sequences becomes prohibitive and in practice it is unrealizable. The Viterbi algorithm uses the method of maximum likelihood decoding with a few modifications to reduce this workload.

a. Preliminaries

Before explaining the Viterbi procedure, two quantities need to be defined: the Hamming weight and the Hamming distance. The Hamming weight, w_H, of a binary sequence is equal to the number of 1s in the sequence. For example, the codeword $\mathbf{u_1}$ = [1 0 0 1 1 0] has $w_H(\mathbf{u_1}) = 3$. A related quantity is Hamming distance, d_H, which is the number of bit by bit differences between any two binary words of the same length. For the codewords $\mathbf{u_1}$ above and $\mathbf{u_2}$ = [1 1 0 1 0 0], $d_H(\mathbf{u_1}, \mathbf{u_2}) = 2$ since they differ in a total of two bit locations (at positions 2 and 5). The Hamming distance and Hamming weight are related as follows:

$$w_H(\mathbf{u_1} \oplus \mathbf{u_2}) = d_H(\mathbf{u_1}, \mathbf{u_2}) \tag{4.5}$$

where the $\oplus$ symbol once again indicates modulo-2 addition. Using the two codewords above,

$$\mathbf{u_1} \oplus \mathbf{u_2} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix},$$

and using equation (4.5),

$$w_H(\mathbf{u_1} \oplus \mathbf{u_2}) = 2 = d_H(\mathbf{u_1}, \mathbf{u_2}).$$

b. Decoding Procedure

Instead of using the conditional probabilities of the received code sequence as the decision criterion, the Viterbi algorithm uses the metric defined by the Hamming distance to determine the most likely transmitted code sequence. Specifically, the code

sequence with the smallest total Hamming distance from the received sequence is the most likely transmitted message.

The Viterbi procedure can be more easily understood by referring to the trellis representation of Figure 4.4, with specific attention on the repetetive pattern which emerges at step n_3. The important feature to notice is that each node has two entry branches and that these remain fixed. These branches are denoted as $\boldsymbol{i}_{j1}$ and $\boldsymbol{i}_{j2}$ (where $(\boldsymbol{i}, j1, j2) \in \{A, B, C, D\}$) where $j1$ and $j2$ are the source nodes which lead to node $\boldsymbol{i}$. Each branch is associated with the output code symbol resulting with a transition from state j to state i. For example, $\mathbf{A}_A$ and $\mathbf{A}_C$ are the two branches which originate from nodes A and C and lead to node A with $\mathbf{A}_A$=00 and $\mathbf{A}_C$=11. As the received code sequence $\mathbf{u}_\mathbf{r}$ arrives, it is parsed into code symbols of length two bits, $\mathbf{u}_{\mathbf{r}l}$ (l = 0, 1, 2,...), and these code symbols are fed to the decoder. At iteration step l, the Hamming distance between code symbol $\mathbf{u}_{\mathbf{r}l}$ and each of the 8 branches is calculated resulting in two Hamming distances per node. For node A, these are $d_H(\mathbf{u}_{\mathbf{r}l},\mathbf{A}_A)$ and $d_H(\mathbf{u}_{\mathbf{r}l},\mathbf{A}_C)$, for example. Each node also has an associated total path metric, $p_l(\boldsymbol{i})$. This is the Hamming distance between the entire path in the trellis which leads to node i at step l, $\boldsymbol{i}_{path}(l)$, and the received code sequence $\mathbf{u}_\mathbf{r}(l)$ as of step l,

$$p_l(\boldsymbol{i}) = d_H(\mathbf{u}_\mathbf{r}(l),\boldsymbol{i}_{path}(l)) \qquad (4.6)$$

computed at the end of step l. The two Hamming distances per node are summed with the respective total path metrics of the source nodes calculated during the previous iteration:

$$d_H(\mathbf{u}_{\mathbf{r}l},\boldsymbol{i}_{j1}) + p_{l-1}(\boldsymbol{j1})$$

and

$$d_H(\mathbf{u}_{\mathbf{r}l},\boldsymbol{i}_{j2}) + p_{l-1}(\boldsymbol{j2}).$$

The total path metric for node $\boldsymbol{i}$ at step l is the smaller of the two values,

$$p_l(\boldsymbol{i}) = \min\{d_H(\mathbf{u}_{\mathbf{r}l},\boldsymbol{i}_{j1}) + p_{l-1}(\boldsymbol{j1}),\ d_H(\mathbf{u}_{\mathbf{r}l},\boldsymbol{i}_{j2}) + p_{l-1}(\boldsymbol{j2})\} \qquad (4.7)$$

In this way, the path which corresponds to the lesser total distance is kept while the other is discarded. After a number of codesymbols of $\mathbf{u}_\mathbf{r}$ have been processed, the nodes in all of

the i_{path} sequences some y steps behind the current iteration will converge to the same path for all of the states such that at step l,

$$p_{l-y}(\mathbf{A}) = p_{l-y}(\mathbf{B}) = p_{l-y}(\mathbf{C}) = p_{l-y}(\mathbf{D}). \qquad (4.8)$$

This common path is part of the desired sequence and can be easily decoded to the original bit sequence. The process continues until only one path is left through the entire trellis and the entire sequence has been decoded. This bit pattern is the most likely transmitted information sequence and is subsequently fed to the source decoder for final processing.

c. Hard vs. Soft Decisions

So far in this section, the assumption is that the input to the decoder is either a 0 or a 1 bit, correponding to some defined two-level voltage assignment. This method is called hard decision coding because the detector preceding the channel decoder determines whether or not the received bit from the channel is a 0 or a 1. There is another implementation in which the detector outputs a multilevel voltage and the channel decoder bases its output on these inputs. This soft decision decoding procedure has been shown to provide a 2 - 3 dB coding gain over the hard decision process but comes with the price of increased receiver and channel decoder complexity [Ref. 10]. In this thesis only hard decision decoding was used.

3. Memory Requirements and Computational Load

It is apparent that the repetetive Hamming distance calculations and trellis path comparisons can lead to a loss in speed of operation of the algorithm. Furthermore, although the number of trellis branches that need to be stored decreases whenever a new minimized path is found, this does not necessarily happen at every iteration. There are a few factors which must be weighed against each other in order to optimize performance. These are the frequency of path metric calculations and branch comparisons, the desired speed of operation, the amount of branch history that needs to be saved, and memory limitations.

While frequent calculations will reduce memory requirements and yield output data at a more constant rate, the time taken to perform the calculations and update the path

histories will slow down the overall execution time. For larger constraint length coders, since more bits are used to encode the information, it is not necessarily productive to perform frequent calculations since the decoder will require more received bits to resolve any errors through the trellis. In other words, the decoder might end up doing all of the calculations yet not produce any output.

For rate $1/n$, constraint length K coders, there are 2^{K-1} paths that must be stored. The amount of path history that is retained for these paths is also critical to the decoder's performance. If too much path history is kept, then large amounts of memory are used. Coupled with infrequent calculations and trellis updates, the computational overhead becomes massive during each step when calculations are performed. On the other hand, the probability of resolving that part of the sequence correctly increases because there is more information with which to work. Obviously, there needs to be a compromise in order to minimize demands on memory and computational load while maximizing decoding efficiency.

It has been shown that the total amount of storage required by the decoder to maintain all of the state histories, so the input can be corrrectly recovered is

$$s = h2^{K-1}, \qquad (4.9)$$

where s is the total trellis branch storage requirement, h is the length of the information path for each state, and 2^{K-1} are the number of states. The value of h is generally betweeen 4 to 5 times the constraint length [Ref. 2].

C. IMPLEMENTATION

The source code for the channel encoders and decoders used in this thesis are given in Appendix B. The details of the codes used and their implementation are given below.

1. Encoders

In today's communication systems, there are a variety of convolutional coders in use. Two commonly used varieties are the $K = 5$ and the $K = 7$, $r = 1/2$ coders. The parameters of these codes are given in Table 4.1. The code vectors are each for the non-

systematic versions yielding the listed free distances. Coding was straightforward and the code for the more involve $K = 7$ coder is provided as file **K7_enc.C** in Appendix B. Both the input and output data are in unpacked unipolar binary format. The encoder will take any binary data in unipolar format and output the coded sequence. Additionally, after the last bit enters the coder, it is flushed through the shift register with six zeros to fully involve it in the encoding process.

TABLE 4.1: Convolutional Code Parameters

K	code vectors	free distance	t
5	10111 11001	7	3
7	1001111 1101101	10	4

2. Viterbi Decoder

The following information is specific to the $K = 7$ decoder. Similar results hold for the less complicated $K = 5$ decoder. For a $K = 7$ code, there are $2^{K-1} = 64$ states. The states and their associated Hamming weights were found using MATLAB. Both the states and Hamming weights are coded in a lookup table included in header files. The path history, h, for each state is limited to 40 which is slightly larger than suggested in the preceding section, but it is used to ensure near optimal performance.

When the decoding process begins, the algorithm proceeds without any knowledge of the initial state of the encoder. This makes decoding a more generalized procedure in the event the shift register of the coder was not flushed with zeros prior to encoding or if multiple data streams are encoded back to back. After every two bits enter the decoder, the 128 Hamming distances (two per node) appropriate for that input pair are accessed from a lookup table and summed with the total path metrics of the corresponding source nodes from the previous iteration. This yields 128 paths through the trellis of which only half are kept. This add, compare, and select routine is performed at every iteration.

All 64 paths are checked against each other every 15 iterations to determine whether their $i_{path}(l)$ sequences are the same. This corresponds to 30 input bits. When all the common nodes are found, they are decoded and the outputs are written to a file. These nodes are then flushed from memory, the path histories are all reset with new starting points at the most recent common source node. This way, data is routinely being output, and the memory requirements are minimized.

There is a chance that, at the tail end of the incoming message sequence, there might not be enough coded bits to fully resolve the message sequence. This results in a minimal bit loss on the order of 15 - 20 bits. When compared to the total bit length of the message sequence, this is minor; however, this problem can be averted by appending more encoded bits to the sequence (i.e., flushing more zeros through the coder) thereby providing the decoder with more information.

The decoder routine is under the filename **Viterbi_7.C** in Appendix B. Additionally, the header files **vit7_consts.h**, **vit7hamwts.h**, and **vit_functions.h** contain lookup tables and function routines used to compare, select, update, and decode paths through the trellis. The code for the $K = 5$ decoder is similar to that given for the $K = 7$ coder with some simplifications and changes in decoding parameters.

FEC codes provide a method of error correction in a transmitted data sequence. Although they increase the data rate in a system, the benefits of FECs under noisy conditions outweigh this inherent characteristic. Convolutional codes and Viterbi decoding are widely used methods of implementing FECs. The computer code contained in Appendix B is an implementation of a widely used convolutional code and suited for channel simulations.

V. DIGITAL MODULATION

In digital transmission systems, the data sequence from the channel encoder is partitioned into L bit words, and each word is mapped to one of M corresponding waveforms according to some predetermined rule, where $M = 2^L$. For example, in the case of baseband modulation, the words are $L = 1$ bits each where each bit correponds to one of two different voltage levels. As we shall see later, in a QPSK modulation system, the incoming sequence is separated into words of $L = 2$ bits each and mapped to $M = 2^2 = 4$ different waveforms. During transmission, the channel causes attenuation and introduces noise to the signal. The net result is the formation of a version of the original signal which may not be detected correctly by the receiver. If the errors are too numerous, the channel decoder may not be able to resolve the information correctly. Baseband modulation using the simple binary symmetric channel model is briefly discussed, and the details of QPSK modulation are then presented.

A. BASEBAND MODULATION

Baseband modulation is the process of transmitting each bit in a sequence by means of a voltage pulse. The bit information is either contained in an assigned voltage level or in the transition between two different voltage level. In this thesis, the two baseband representations used are the unipolar and the bipolar formats. Unipolar format assigns a high voltage level for T_b seconds to one of the bits and a zero voltage level of the same duration to the other. The value T_b is the bit duration time and is the reciprocal of the data rate, R. For the bipolar format, the bits are assigned voltages of the same magnitude but of opposite sign. Usually, the 1 bit is assigned the higher voltage and the 0 bit is assigned the lower value. Further information on baseband representation is available in [Ref. 11].

1. Binary Symmetric Channel

The binary symmetric channel (BSC) is the simplest model for a channel, and it is a special case of the discrete memoryless channel (DMC). The inputs and outputs of a DMC come from finite discrete alphabets where each output depends on a set of

conditional probabilities given each input. In the case of a BSC, the input and output alphabets are both the same, namely {0, 1}, and the ouputs are described by the conditional probabilities P(0|0), P(0|1), P(1|0), and P(1|1). Because the channel is symmetric, the following relationships hold:

$$P(0|1) = P(1|0) = P_b \tag{5.1}$$

and

$$P(0|0) = P(1|1) = 1 - P_b. \tag{5.2}$$

A conceptual picture of a BSC is shown in Figure 5.1. Although it is easy to associate the BSC with a baseband modulation scheme, the waveform representation of the data sequence is irrelevant because the BSC models the channel effects on the underlying binary data sequence regardless of the modulation method. The BSC is very useful for strictly modeling the effects of P_b on the fidelity of the information sequence after it has been source decoded. Similarly, it can be used to test the effectiveness of the channel encoding and decoding routines. Both of these tasks were accomplished by coding a BSC and sending data across it. For additional reading on DMC and BSC, see Couch [Ref. 11].

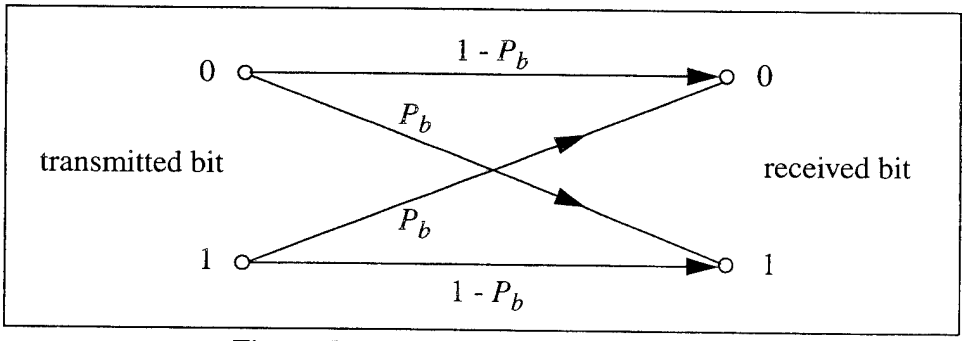


Figure 5.1 Binary Symmetric Channel

The coded BSC model using baseband modulation is contained in Appendix C as **baseband.C**. Upon execution, a user supplied P_b is applied to a data sequence, and each bit is changed based on that probability, thereby simulating a bit error.

B. QPSK MODULATION

In this section the topics of QPSK modulation of digital signals including their transmission, demodulation, and detection, are developed. The material in this section and the related coding of this system are both based on transmission using an AWGN channel model which is covered at the end of this section. Some of the techniques discussed below are specifically designed for robustness under these conditions.

Because this is a digital implementation of a digital system, it is important to note that the only places where analog quantities occur are after the DAC, prior to the actual transmission of the signal, and before the ADC at the receiver. All signal values between the source encoder input and modulator output are purely digital. This also holds for all quantities between the demodulator and the source decoder.

1. Background

QPSK modulation is a specific example of the more general M-ary PSK. For M-ary PSK, M different binary words of length $L = \log_2 M$ bits are assigned to M different waveforms. The waveforms are at the same frequency but separated by multiples of $\varphi = 2\pi/M$ in phase from each other and can be represented as follows:

$$s_i(n) = \cos\left(2\pi\frac{f_c}{f_s}n + \varphi_i\right), \quad \varphi_i = \frac{2\pi i}{M} \tag{5.3}$$

with $i = 1, 2, \ldots M$. The carrier frequency and sampling frequency are denoted by f_c and f_s, respectively.

Since an M-ary PSK system uses L bits to generate a waveform for transmission, its symbol or baud rate is $1/L$ times its bit rate. For QPSK, there are $M = 4$ waveforms separated by multiples of $\varphi = \pi/2$ radians and assigned to four binary words of length $L = 2$ bits. Because QPSK requires two incoming bits before it can generate a waveform, its symbol or baud rate, D, is one-half of its bit rate, R.

2. Transmitter

Figure 5.2 illustrates the method of QPSK generation. The first step in the formation of a QPSK signal is the separation of the incoming binary data sequence, $\boldsymbol{b}$, into an in-phase bitstream, $\boldsymbol{b_I}$, and a quadrature phase bitstream, $\boldsymbol{b_Q}$, as follows. If the incoming data is given by $\boldsymbol{b} = b_0, b_1, b_2, b_3, b_4, \ldots$ where b_i are the individual bits in the sequence, then $\boldsymbol{b_I} = b_0, b_2, b_4, \ldots$ (even bits of $\boldsymbol{b}$) and $\boldsymbol{b_Q} = b_1, b_3, b_5, \ldots$ (odd bits). The digital QPSK signal is created by summing a cosine function modulated with the $\boldsymbol{b_I}$ stream and a sine function modulated by the $\boldsymbol{b_Q}$ stream. Both sinusoids osciallate at the same digital frequency, $\omega_o = 2\pi f_c / f_s$ radians. The QPSK signal is subsequently filtered by a bandpass filter, which will be described later, and sent to a DAC before it is finally transmitted by a power amplifier. Details on the DAC and subsequent transmission operations can be found in Frerking [Ref. 5] and Freeman [Ref. 10].

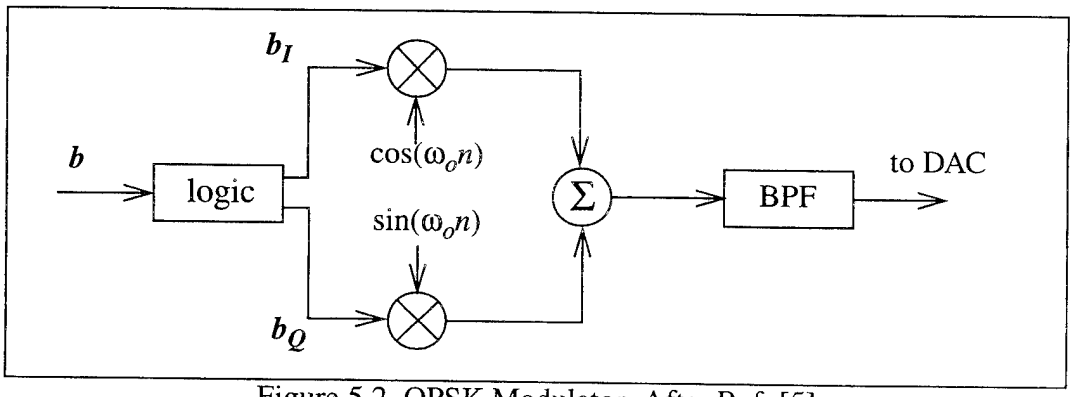


Figure 5.2 QPSK Modulator. After Ref. [5]

a. Signal Constellation

It is often helpful to represent the modulation technique with its signal space representation in the *I-Q* plane as shown in Figure 5.3. The two axes, *I* and *Q*, represent the two orthogonal sinusoidal components, cosine and sine, respectively, which are added together to form the QPSK signal as shown in Figure 5.2. The four points in the plane represent the four possible QPSK waveforms and are separated by multiples of $\pi/2$ radians from each other. By each signal point is located the input bit pair which produces the

respective waveform. The actual I and Q coordinates of each bit pair are the contributions of the respective sinusoid to the waveform. For example, the input bits (0,1) in the second quadrant correspond to the (I,Q) coordinates, (-1,1). This yields the output waveform $-I+Q = -\cos(\omega_o n) + \sin(\omega_o n)$. Because all of the waveforms of a QPSK have the same ampilitude, all four points are equidistant from the origin. Although the two basis sinusoids shown in Figure 5.2 are given by $\cos(\omega_0 n)$ and $\sin(\omega_0 n)$, the sinusoids can be any two functions that are orthogonal.

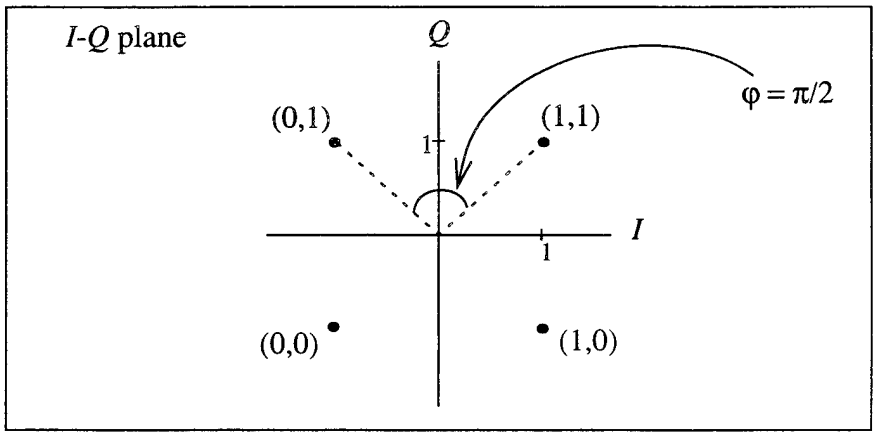


Figure 5.3 Signal Constellation of QPSK

b. Filtering

The QPSK signal created by the addition of the two sinusoids has significant energy in frequencies above and below the carrier frequency. This is due to the frequency contributions incurred during transitions between symbols which are either 90 degrees or 180 degrees out of phase with each other. It is common to limit the out of band power by using a digital bandpass filter (BPF) centered at ω_0. The filter has a flat passband and a bandwidth which is 1.2 to 2 times the symbol rate [Ref. 5].

3. Receiver

The receiver's function consists of two steps: demodulation and detection. Demodulation entails separating the received signal into its constituent components. For a QPSK signal, these are the cosine and sine waveforms carrying the bit information.

Detection is the process of determining the sequence of ones and zeros those sinusoids represent.

a. Demodulator

The demodulation procedure is illustrated below in Figure 5.4. The first step is to multiply the incoming signal by locally generated sinusoids. Since the incoming

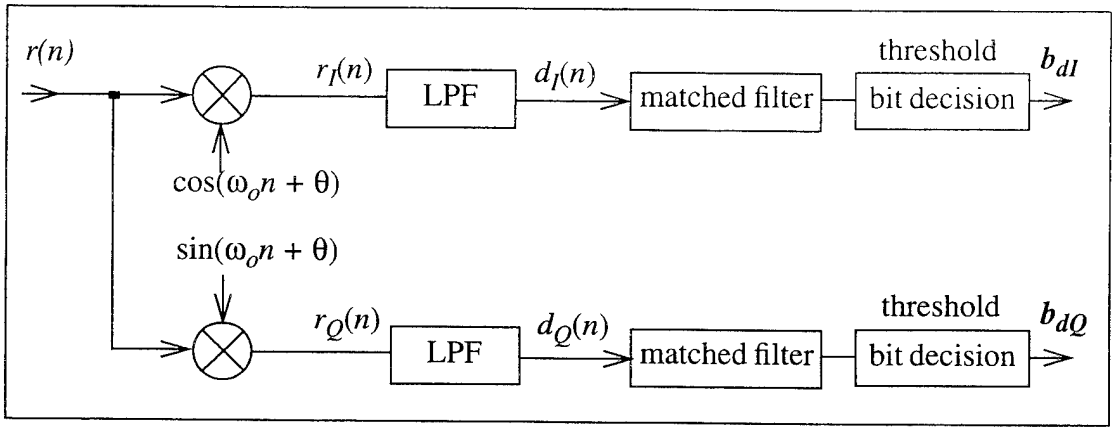


Figure 5.4 QPSK Demodulator and Detector

signal is a sum of sinusoids, and the receiver is a linear system, the processing of the signal can be treated individually for both components and summed upon completion. Assuming the received signal is of the form

$$r(n) = A_I \cos(\omega_o n) + A_Q \sin(\omega_o n) , \qquad (5.4)$$

where A_I and A_Q are scaled versions of the b_I and b_Q bitstreams used to modulate the signal at the transmitter. The contributions through the upper and lower arms of the demodulator due to the $\cos(\omega_o n)$ input alone are

$$r_{cI}(n) = A_I \cos(\omega_o n)\cos(\omega_o n + \theta) \qquad (5.5)$$

and

$$r_{cQ}(n) = A_I \cos(\omega_o n)\sin(\omega_o n + \theta) \qquad (5.6)$$

where θ is the phase difference between the incoming signal and locally generated sinusoids. These equations can be expanded using trigonometric identities to yield

$$r_{cI}(n) = 0.5A_I[\cos(2\omega_o n + \theta) + \cos(-\theta)] \quad (5.7)$$

and

$$r_{cQ}(n) = 0.5A_I[\sin(2\omega_o n + \theta) - \sin(-\theta)]. \quad (5.8)$$

The LPFs remove both high frequency terms leaving only the terms ,

$$d_{cI}(n) = 0.5A_I\cos(-\theta) = 0.5A_I\cos(\theta) \quad (5.9)$$

and

$$d_{cQ}(n) = (-0.5)A_I\sin(-\theta) = 0.5A_I\sin(\theta). \quad (5.10)$$

These terms are proportional to the phase difference between the incoming signal and the locally generated sinusoid. Using a similar development for the $\sin(\omega_o n)$ portion of the input $r(n)$, the outputs of the upper and lower arms of the demodulator are

$$d_{sI}(n) = (-0.5)A_Q\sin(\theta) \quad (5.11)$$

$$d_{sQ}(n) = 0.5A_Q\cos(\theta) \quad (5.12)$$

The demodulated bitstreams $d_I(n)$ and $d_Q(n)$ are found by summing the signal components in (5.9) and (5.11),

$$d_I(n) = 0.5A_I\cos(\theta) + (-0.5)A_Q\sin(\theta), \quad (5.13)$$

and (5.10) and (5.12),

$$d_Q(n) = 0.5A_I\sin(\theta) + 0.5A_Q\cos(\theta). \quad (5.14)$$

For $\theta = 0$, equation (5.13) produces a scaled version of the in-phase bitstream while equation (5.14) is the quadrature phase component. The goal of driving θ to zero is realized with a phase locked loop (PLL).

b. Synchronization

For the received data to be detected and interpreted correctly, there needs to be coordination between the receiver and the transmitter. Since they are not physically connected, the receiver has no direct means of knowing the 'state' of the transmitter. This state includes the both the phase argument of the modulator and the bit timing of the

transmitted data sequence. The receiver must therefore extract the desired information from the received digital signal to achieve synchronization.

A common means of accomplishing synchronization is with a PLL as shown in Figure 5.5. The inputs to the PLL are the outputs from the two LPFs in Figure 5.4. As shown in equations (5.13) and (5.14), the outputs of both LPFs are a function of the phase difference, θ, between $r(n)$ and the sinusoids. The two demodulated data streams

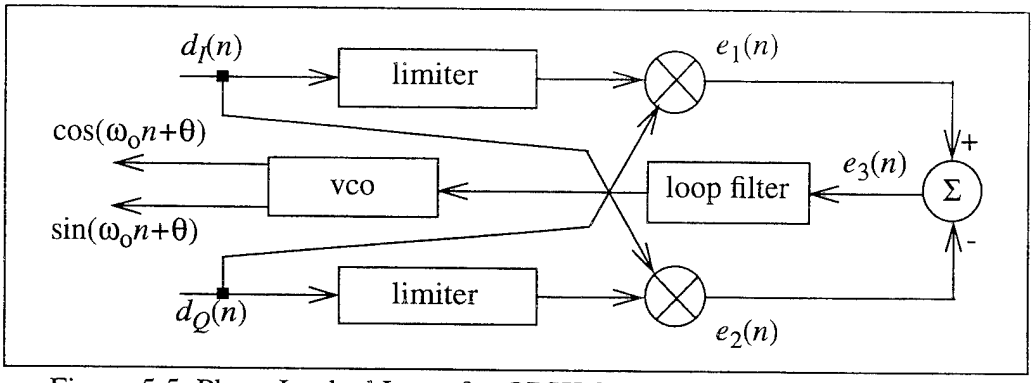


Figure 5.5 Phase Locked Loop for QPSK Synchronization After Ref. [5]

are each multiplied by the sign of the other stream, using the limiters,

$$e_1(n) = sign(d_Q(n))d_I(n) \tag{5.15}$$

$$e_2(n) = sign(d_I(n))Q(n), \tag{5.16}$$

and subtracted to yield the error signal input, $e_3(n)$, to the loop filter:

$$e_3(n) = sign(d_Q(n))d_I(n) - sign(d_I(n))Q(n). \tag{5.17}$$

The loop filter is an FIR low pass filter given by [Ref. 5]

$$y(n) = \sum_{i=0}^{N=9} (0.8)^i x(n-i) \tag{5.18}$$

where the output $y(n)$ depends on a weighted average of its previous inputs, $x(n)$. The output is sent as the phase argument to the voltage controlled oscillator (vco) which produces the two sinusoid outputs used to demodulate the received signal, $r(n)$.

c. Detection

After the signal $r(n)$ has been demodulated into the bitstreams $d_I(n)$ and $d_Q(n)$, the corresponding bit information must be recovered. The commonly used technique is to use a matched filter at the output of each LPF as shown in Figure 5.4. The matched filter is an optimum receiver under AWGN channel conditions and is designed to produce a maximum output when the input signal is a mirror image of the impulse response of the filter. The outputs of the two matched filters are the detected bitstreams $\boldsymbol{b}_{dI}$ and $\boldsymbol{b}_{dQ}$, and they are recombined to form the received data bitstream. The development of the matched filter and its statistical properties as an optimum receiver under AWGN conditions can be found in various texts [Refs. 3 and 7].

4. AWGN Channel

The previously introduced BSC channel modeled all of the channel effects with one parameter, namely the BER; however, this model is not very useful when attempting to more accurately model a communication system's behavior. The biggest drawback is the lack of emphasis given to the noise which significantly corrupts all systems.

The most commonly used channel model to deal with this noise is the additive white Gaussian noise (AWGN) channel model. The name results because the noise is simply added to the signal while the term 'white' is used because the frequency content is equal across the entire spectrum. In reality, this type of noise does not exist and is confined to a finite spectrum, but it is sufficiently useful for systems whose bandwidths are small when compared to the noise power spectrum. When modeling a system across an AWGN

channel, the noise must first be filtered to the channel bandwidth prior to addition. An illustration is shown in Figure 5.6.

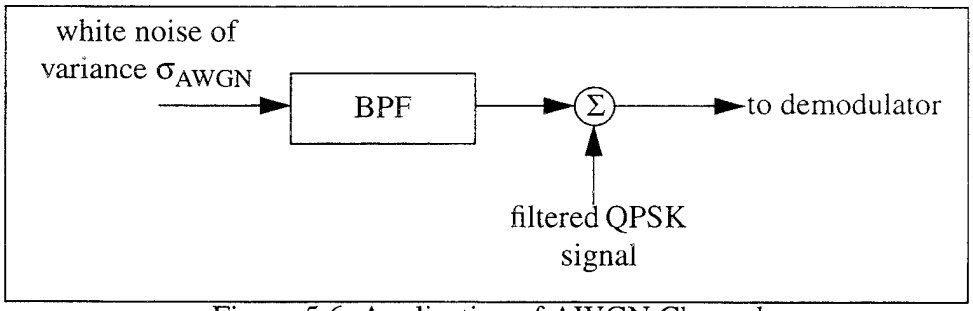


Figure 5.6 Application of AWGN Channel

C. IMPLEMENTATION

All code for this chapter is contained in Appendix C. There are a variety of LPFs and BPFs used in the routines, and the coefficients of these filters are documented in the code. All filters were designed using the MATLAB functions *firl*, *butter*, and *filter* in the Signal Processing Toolbox [Ref. 12]. All of the repeatedly used values such as cos, sin, and filter coefficients are retrieved from lookup tables to reduce computational load. The entire QPSK system is coded as one function to reduce execution time. Specifically, even though the inputs to the modulator and the outputs from the detectors are binary, all of the intermediate values are in floating point. By combining all of the functions, the large amounts of time used in processing the floating point values and the associated file input/ outputs are reduced. The files used are **qpsk.C**, **qpskmod.h**, **demod.h**, **bpf.h**, and **noise.h** which are included in Appendix C.

1. Modulator

The modulator uses the output from the channel encoder to create the QPSK signal. Because the output of the channel encoder is in unipolar form, it is first converted to bipolar form with all 0 bits changed to the value -1. The modulated frequency is 455 kHz which is commonly used as an intermediate frequency in superheterodyne receivers. All calculations were performed at this frequency, and the modulated frequency will be

denoted as f_c even though it is not the carrier frequency. The sampling frequency, f_s , = $8f_c$ = 3640 kilosamples/second, and the sample time is, $T_s = 1/f_s$.

Using CELP coded speech, the output bit rate of the convolutional coder is $R = 9600$ bps. Using QPSK modulation which needs $L = 2$ bits to generate a waveform results in a symbol rate, D, of 4800 baud; therefore, the number of samples per baud, N_b, is given by

$$N_b = round\left(\frac{f_s}{D}\right). \tag{5.19}$$

Given the input bitstream $\boldsymbol{b}$, every T_s seconds, the bits b_i (in-phase bit) and b_{i+1} (quadrature phase bit) are multiplied by a cosine and a sine function, respectively, and summed. After N_b samples, the bits b_{i+2} and b_{i+3} are used and so on until the entire bitstream is used. For the given f_s and D, $N_b = 758$ samples/symbol.

Both sinusoids are obtained from a lookup table of 8 values (because $f_s = 8f_c$) to reduce computations and minimize the program run time. After modulation, the digital signal is filtered with a 6th order digital Butterworth bandpass filter with the digital passband frequencies given by

$$\omega_{c1} = \frac{(f_c - 0.3R)}{f_s} \tag{5.20}$$

$$\omega_{c2} = \frac{(f_c + 0.3R)}{f_s}. \tag{5.21}$$

These correspond to the real frequencies 452.12 kHz and 457.88 kHz, respectively. All data is output to a file with floating point precision. The filter coefficients for both the BPF and the interpolation filter and other details are documented in the code in Appendix. For more details on the filtering operations and routines see Kraus [Ref. 12] and Embree [Ref. 13].

2. Demodulator/Detector

For a given f_c, f_s, and associated filter parameters, the delay of the modulated signal through the system is fixed. Using this delay, the received signal can be demodulated

correctly. The additional delay through the demodulator LPFs is similarly fixed and can be used to correctly set the bit timing for bit retrieval from the matched filter detectors. In this system, with f_c = 455 kHz and f_s = 3640 kilosamples/s, the full system delay from modulator to matched filter is 452 samples, so the first 452 samples arriving at the matched filter are discarded since these do not represent any of the data. Since there are N_b = 758 samples per bit, the matched filters produce a maximum output value every 758 samples. Therefore every 758 input values to each matched filter are stored in a buffer, and the dot product of this buffer with the 758 matched filter coefficients is used to determine the output bit value. The dot product is the FIR filtering operation of the matched filter. A threshold is used such that a positive result corresponds to an output bit of 1 and a 0 otherwise (see Figure 5.4).

3. AWGN Channel

Gaussian noise is generated using the Box-Muller random number generator algorithm [Ref. 13]. The noise variance, σ^2_{AWGN}, is user defined. Prior to adding the noise to the QPSK signal, it is filtered with the same BPF used to filter the digital QPSK signal. The filtered noise is added to the filtered QPSK signal prior to the demodulation routine.

Baseband modulation over a BSC provides a simple means of analyzing system performance and is beneficial when the details of the modulation scheme are not critical. QPSK is a widely used modulation method used in a variety of links, such as satellite communications. A coded QPSK system using an AWGN channel model provides a realistic, albeit rudimentary, method for analyzing the performance of a digital system.

VI. RESULTS

The preceding chapter developed the details of a digital communication system and the implementation method of each of the system functions. Now, the goal is to test these functions and to successfully integrate them, allowing data to pass through the entire system. Specifically, the performance of the FEC channel coders, the QPSK modulation system, and the CELP coder are of interest. The following two sections contain the results of these tests. In each case, the primary data sets were short segments (3 - 4 seconds) of speech data obtained on the internet.

A. BINARY SYMMETRIC CHANNEL

1. Channel Coder

As mentioned in Chapter V, for the purposes of testing the CELP coder and measuring the effectiveness of the channel coders, baseband simulation using a binary symmetric channel (BSC) model is the best approach. Using a random data stream of 50,000 bits, the channel encoder produced 100,000 bits which were sent over the BSC with different P_b values. These P_b values were obtained via a user input interface during execution of the baseband channel program and were used to determine whether an input bit should be corrupted. The received data stream is decoded and the resulting total error is the number of bit errors between the decoded and the original bits. Ideally, the effective bit error rate (BER) after the channel decoder should be equal to zero.

Table 6.1 shows the performance of both coders under the various bit error probabilities. As the table shows, for a channel with $P_b \leq 0.03$, both $K = 5$ and $K = 7$ convolutional coders fully resolve the data. For $0.03 < P_b < 0.1$, both coders improve the bit error rate but do not reduce it to zero; the $K = 7$ coder performs better than the $K = 5$ coder. For $P_b \geq 0.1$, both coders exhibit performance deterioration.

TABLE 6.1: Convolutional code effect on P_b

channel P_b	coder constraint length, K	effective P_b after decoder
0.001	5	0.0
	7	0.0
0.01	5	0.0
	7	0.0
0.03	5	0.0
	7	0.0
0.05	5	0.003
	7	0.002
0.07	5	0.018
	7	0.011
0.1	5	0.24
	7	0.08

The positive effects of channel coding can best be seen with image data. Figure 6.1(a) shows the original. The image dimensions are 480 x 640 pixels with each pixel represented by an eight-bit grey scale, resulting in a total of approximately 2.5 Megabits. The image was first sent across the BSC with a $P_b = 0.05$ but without any channel encoding, and the result is shown in Figure 6.1(b). Finally, Figure 6.1(c) is the result of using a $K = 5$ channel encoder and is clearly much improved over the previous rendition; however, this performance was obtained with the transmission of twice as many bits, nearly 5 Megabits, across the BSC. Nevertheless, the purpose of the illustration is to demonstrate the effectiveness of the channel encoding scheme ($K = 5$) detailed in Chapter IV.

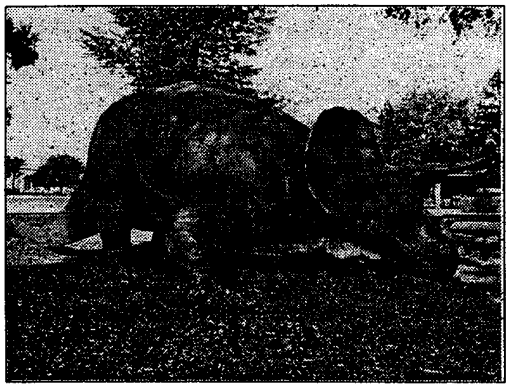
Original

(a)

BER = 0.05

(b)

K=5 FEC



(c)

Figure 6.1 Effect of using a Channel Coder

B. QPSK MODULATION

1. System Performance

Performing the QPSK simulation using the CELP coder involved the use of a number of executables including both the main system functions and interface 'modules'. These modules convert the outputs of the main system functions into the appropriate data formats necessary at the inputs of the succeeding system functions. Appendix D contains the sequence of executables necessary to perform a QPSK simulation.

The performance of the QPSK system is measured by its channel bit error rate (P_b), before any channel decoding, for a given E_b/N_o (in dB). E_b and N_o are calculated at the inputs of each matched filter. The measured value of E_b is 0.00002 Joules/bit. To determine the noise power spectral density, the variance of the filtered AWGN is divided by the system bandwidth, B = 5760 Hz. The MATLAB script **ebno.m** in Appendix E is used to calculate the E_b/N_o values for different input standard deviation, σ_{AWGN}, values. The actual BER was determined by comparing the actual bit patterns before modulation with those obtained after detection (the bit differences in the files are counted using MATLAB) but prior to channel decoding.

Figure 6.2 shows a plot of the probability of bit errors as a function of the signal-to-noise ratio (E_b/N_o). The solid line indicates theoretical performance, obtainable from Sklar [Ref. 2] and Freeman [Ref. 10], and the measured performance is shown by the dotted line; the dashed portion of the line indicates projected values. The theoretical curve is obtained from the following relation:

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_o}}\right) \qquad (6.1)$$

where $Q(\cdot)$ is the error function. A complete derivation of equation (6.1) for a QPSK system is provided in Sklar [Ref. 2]. Although the system performance falls short of the ex-

pected theoretical values, the measured points follow the same shape of the curve. The measured values are on average displaced from the theoretical values by about 2 to 3 dB, which

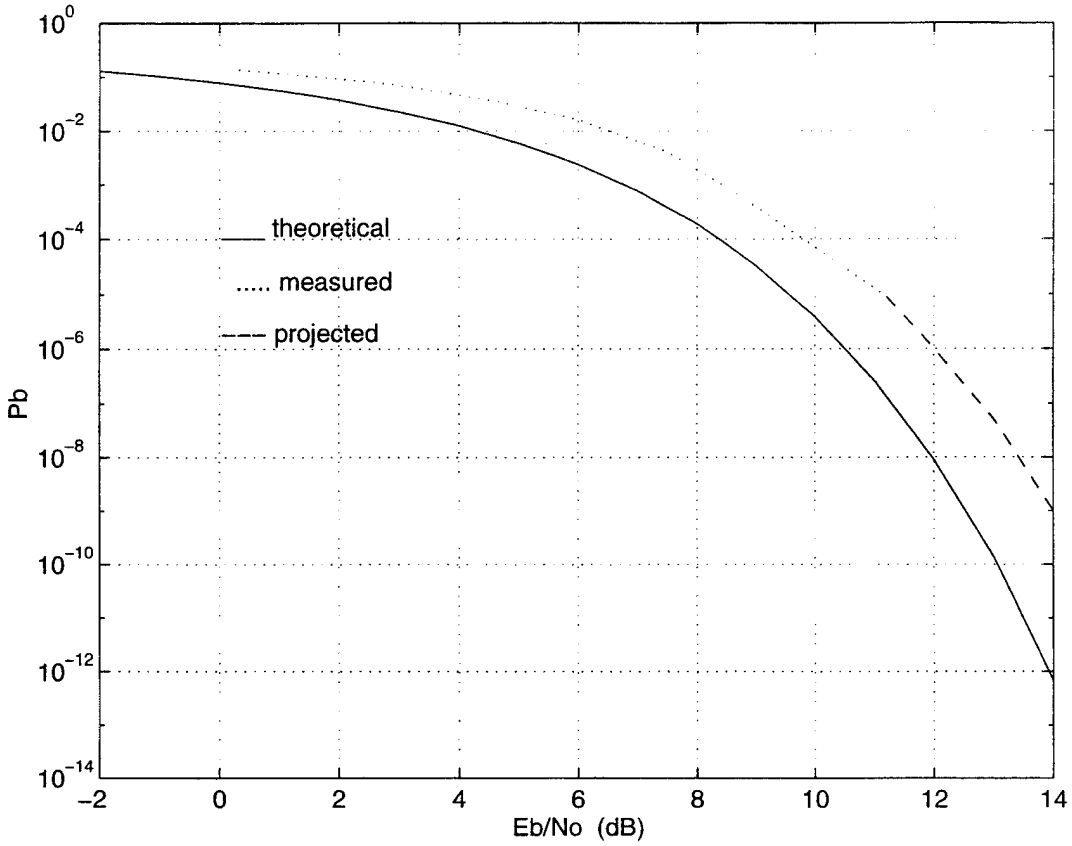


Figure 6.2 P_b vs E_b/N_o (Theoretical and Measured Values)

can be accounted for implementation loss resulting from the non-ideal frequency response characteristics of the channel filter.

Table 6.2 shows the effective BER at the source decoder for several E_b/N_o values when using a channel coder (K = 7). For $E_b/N_o > 5$ dB, the channel decoder corrected all

of the errors while for $E_b/N_o < 5$ dB, the resultant bitstream contained various errors which the channel decoder could not fully resolve.

TABLE 6.2: BER Results using $K = 7$ Channel Coder for Several E_b/N_o

E_b/N_o(dB)	P_b (from Figure 6.2)	effective system P_b
11.2	0.00001	0
4.8	0.06	0.003
1.6	0.1	0.08

C. CELP RESULTS

CELP is a widely used speech compression/coding scheme, and its performance in terms of speech intelligibility is considered good for cases where there are little or no bit errors. The question is how well it performs under higher levels of bit error. To analyze the performance of the CELP algorithm, the coded speech was channel coded ($K = 7$ coder) and passed across the entire QPSK system using different E_b/N_o values. Following the Viterbi decoder, the data stream with bit errors was passed to the CELP decoder for final processing and speech reproduction.

With any speech reproduction system, the results are subjective, but they can be expressed using the mean opinion score (MOS) tables [Ref. 4]. Table 6.3 contains the MOS values for CELP coded speech samples with different BER.For $P_b \leq 0.001$, the decoded CELP output was intelligible and had good fidelity as expected; for $0.001 < P_b < 0.05$ the

TABLE 6.3: MOS Values of CELP Coded Speech for different BER

Pb	MOS
0.001	4.2
0.010	3.5
0.030	2.7
0.050	2.1

CELP decoder output was intelligible, but there was some loss in fidelity. Finally, for $P_b \geq 0.05$, the audio output rapidly degraded to being unintelligible with little fidelity. Because CELP provides a compression of 27:1, each bit in a coded CELP frame carries more information than a bit in the corresponding uncoded linear PCM bit stream. This means that higher BERs result in more severe distortion in the CELP decoded output than in the uncompressed speech

In summary, the results clearly show the benefits of using channel coding and the successful implementation of the QPSK system in software. Specifically, the channel coders effectively reduced the BER to zero when channel bit errors are below 0.03. Poor performance at higher BERs is expected when implementing systems with memory constraints and fast speed requirements. The QPSK system performance closely paralleled theoretical results. The CELP algorithm provides an effective, low bit-rate speech representation. For moderate amounts of errors, $P_b < 0.03$, the decoded speech is intelligible with good waveform fidelity; however, for higher error rates, the decoder performance is poor and unusable.

VII. CONCLUSIONS

A. CONCLUSIONS

In this thesis the details of a specific digital communication system implementation have been discussed. Specifically, the mechanisms necessary to use QPSK modulation and convolutional coding to transmit compressed speech have been examined. CELP compression is an efficient speech compression technique which produces low data rate coding while maintaining the intelligibility of the processed speech. Convolutional coding and Viterbi decoding are established channel coding schemes which allow safe transmission of data under noisy, error producing conditions. Lastly, QPSK is an efficient modulation scheme currently used by modern satellite communication links.

The full implementation shows the viability of coding a digital link in software and its use as a simulation tool for real data. The modular nature allows for interchangeability of components as was shown with the use of different constraint length channel coders. However, as currently coded, the system is not flexible in accomodating different carrier frequencies or sampling frequencies. This is due to the lack of a PLL which would resolve the synchronization issues and allow for the use of different simulation frequencies.

B. FUTURE WORK

For the purposes of testing the link and simulating the transmission of data, this software implementation provides a reasonable platform for future work. Additionally, the benefits of channel coding and effectiveness of various modulation schemes can be realistically represented. As it stands, the system is rudimentary and not optimized for speed. Future work may consist of:

- optimizing the C code and porting it to a DSP chip,
- using active filters and components to simulate channel conditions,
- incorporating the link as a subsystem of a much larger network,
- using the link to transmit image data, and

- enhancing the synchronization functions of the link (PLL).

The above improvements and enhancements would eventually result in a complex but flexible functional implementation of a digital communication system. By further modularizing the functions, one of the big advantages of programmable DSP could be exploited, namely, the ability to reconfigure parts of the link quickly without the difficulties involved in the manipulation of hardwired systems.

APPENDIX A. COMPUTER CODE FOR INTERFACING THE CELP CODER WITH THE COMMUNICATION SYSTEM

The CELP encoder outputs coded speech frames in hexadecimal format. These hex values need to be converted to unipolar binary format prior to use by the channel encoder. Similarly, the channel decoder output must be converted back to hex frames of 144 bits each (36 hex characters) prior to use by the CELP decoder. The following two scripts perform these conversions.

```
/*****************************************************************************
filename: hex2bin.C

   Function: converts 144 bit framed CELP encoded data from hex to binary.

   Input: packed hex frames (36 characters/frame = 144 bits).
   Output: unpacked unipolar binary data.
*****************************************************************************/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

#define  FRAMESIZE      36
#define          ASCII      55

main(int argc, char *argv[])
{
/* variables:
 *        nibble     -     single hex character from a frame
 *        displayMask -     picks off each bit in nibble
 *        frame[]    -     contains all 36 characters in one CELP frame
*/

  unsigned short nibble, displayMask = 0;
  unsigned char frame[FRAMESIZE+1]= {0};

/* file i/o.
*/

  if (argc != 3) {
    cout << "function: h2b converts celp '.chan' files from hex -> binary"
       << endl << "Usage: h2b infile outfile" << endl << endl;
    exit(1);
  }
  else {
```

```
    ifstream celpIn(argv[1], ios::in);
    if (!celpIn) {
      cerr << "Input file " << argv[1] << " could not be opened "<< endl;
      exit(1);
    }

    ofstream binOut(argv[2], ios::out);
    if (!binOut) {
      cerr << "Output file " << argv[2] << "could not be opened" << endl;
      exit(1);
    }


/* read each frame of CELP: then convert to binary*/

    while (celpIn >> frame) {

/* read in each hex character of the frame.
*/
      for (int c = 0; c <=35; c++) {
        nibble = (unsigned short)frame[c];

/* convert from ASCII representation to 4 bit binary quantity
*/
        if ((nibble >= '0')&&(nibble <= '9')) {
          nibble = nibble - (unsigned short)'0';
        }
        else {
          nibble = nibble - ASCII;
        }

/* mask 3 bits of nibble starting with mask = 1 0 0 0 to pick of individual
 * bits in the nibble and output the bits to a file.
*/
        displayMask = 1 << 3;
        for (int c1 = 1; c1 <= 4; c1++) {
          binOut << (nibble & displayMask ? '1' : '0') << " ";
          displayMask >>= 1;
        }
      }
    }
    binOut.close();
    celpIn.close();
  }
  return 0;
}
```

```
/*****************************************************************************
filename: bin2hex.C

   Function: convert binary stream into packed hex format for CELP synthesis -
             the hex nibbles are output to a file in frames of 36 hex/frame.

    Input: file of unpacked unipolar binary channel decoded data.
    Output: packed hexadecimal data - 36 hex/row (==frame).
*****************************************************************************/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>

#define   FRAMESIZE     36
#define   NIBBLESIZE     4

main(int argc, char *argv[])
{
/* variables
 *          bit          -    input bit (0 or 1)
 *          nibble       -    one hex character
 *          bitCount     -    keep track of 4 bits/hex
 *          nibbleCount  -    keep count of 36 nibbles/frame
*/

  unsigned short bit, nibble = 0, bitCount = 3, nibbleCount = 0;

/***/
  if (argc != 3) {
    cout << "function: 'bin2hex' converts bitstream to packed "
        << "hex frames for celp synthesis" << endl
        << "Usage: bin2hex infile outfile" << endl;
  }
  else {
    ifstream binIn(argv[1], ios::in);
    if (!binIn) {
      cerr << "Input file " << argv[1] << " could not be opened "<< endl;
      exit(1);
    }

    ofstream hexOut(argv[2], ios::out);
    if (!hexOut) {
      cerr << "Output file " << argv[2] << " could not be opened" << endl;
      exit(1);
    }

/***/
/* read in each bit (0 or 1).
*/
```

```
    while (binIn >> bit) {

      nibble = nibble + (bit << bitCount);

/* convert each nibble (= 4 bits) into its hex value.
*/
      if ( (bitCount % NIBBLESIZE) == 0) {
        hexOut << hex << nibble;
        nibble = 0;
        bitCount = 4;
        nibbleCount++;

          if (nibbleCount == FRAMESIZE) {
            hexOut << endl;
            nibbleCount = 0;
          }
      }
      bitCount--;
    }

/* if there weren't enough bits at the end of the file to make a nibble,
 * convert whatever is left to hex and output.
*/
    if (bitCount != 3) {
      hexOut << hex << nibble;
      nibbleCount++;
    }

/* if there were not enough nibbles (36) for a frame, output hex F to fill
 * the frame and exit.
*/
    if (nibbleCount != 0) {
      for (int c = nibbleCount; c < FRAMESIZE; c++)
        hexOut << hex << 15;
    }
    hexOut.close();
    binIn.close();
  }
  return 0;
}
```

APPENDIX B. COMPUTER CODE FOR CONVOLUTIONAL CODERS AND VITERBI DECODERS

This appendix contains the C++ code for the $K = 7$ convolutional coders along with the code for the $K = 7$ Viterbi algorithm and associated header files. The $K = 5$ convolutional coder/Viterbi coder routines are similar to the $K = 7$ version with a few slight modifications.

```
/*****************************************************************************
filename: fec7enc.C
                              K=7, r=1/2 Convolutional Coder

  Function: channel codes input binary data using K=7, r = (2,1) convolutional code.

  Input: file of unpacked unipolar binary data (0s and 1s only! with  spaces in between them).
  Output: file of unpacked unipolar binary data.
 *****************************************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#define         K       7
#define     RATE        2

void convolve(unsigned short, unsigned short[]);

main(int argc, char *argv[])
{

/* variables:
*                 bit   ->   input 0 or 1 bit
*        codeSymbol   ->   array of length 2 for the output code symbols
*/
  unsigned short bit;
  unsigned short codeSymbol[RATE] = {0};

/* read in command line parameters.
*/

  if (argc != 3) {
    cout << "function: 'fec7enc' codes binary data with K=7 (2,1) "
        << "conv. coder" << endl
        << "Usage: fec7enc infile outfile" << endl;
  }
  else {
    ifstream binIn(argv[1], ios::in);
    if (!binIn) {
      cerr << "Input file " << argv[1] << " could not be opened" << endl;
      exit(1);
    }
```

```
    ofstream fecOut(argv[2], ios::out);
    if (!fecOut) {
      cerr << "Output file " << argv[2] << " could not be opened "<< endl;
      exit(1);
    }
/***/

/* read in one bit (must be unipolar!! -> a 0 or 1) at a time and compute the
 * two output code symbols.  Output to file in unpacked form - ie. a space
 * between values.
*/
    while (binIn >> bit) {
      convolve(bit,codeSymbol);
      fecOut << codeSymbol[0] << " " << codeSymbol[1] << " ";
    }

/* flush the register with K - 1 = 6 zeros.
*/
    for (unsigned short flush = 1; flush < K; flush++) {
      convolve(flush,codeSymbol);
      fecOut << codeSymbol[0] << " " << codeSymbol[1] << " ";
    }

  fecOut.close();
  binIn.close();
  }
  return 0;
}


/* function computes the two output binary symbols using
*/

void convolve(unsigned short inBit, unsigned short u[])
{

/* variables:
 *          shift_reg  ->   stores the 7 bits in the register
 *                          and values must be kept between function calls
 *          same       ->   sum of bits in the register which are used
 *                          to compute both codesymbols
*/

  static unsigned short shift_reg[K] = {0};
  unsigned short same;

  for (int n = (K-1); n >=1; n--) {
    shift_reg[n] = shift_reg[n-1];
  }

  shift_reg[0] = inBit;

/* code vectors used to determine code bits u1, u2 using modulo 2 addition:    *
 *      (non-systematic):       v1 => 1 0 0 1 1 1 1
```

```
 *                              v2 => 1 1 0 1 1 0 1
 */

  same = shift_reg[0] + shift_reg[3] + shift_reg[4] + shift_reg[6];

/* for every input bit, 2 output bits are generated.
 */
  u[0] = (same + shift_reg[5]) % 2;
  u[1] = (same + shift_reg[1]) % 2;

}
/*************************************************************************
filename: viterbi.C
                Viterbi Decoder for K = 7, r = 1/2 convolutional codes

 Function: implements a Viterbi decoder for an  input stream of unpacked unipolar
           binary data
  Input: file of unipolar binary data - representing demodulated/detected data stream.
  Output: file of unpacked unipolar binary data - estimate of information sequence

 Associated header files:
 *************************************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#include "vit7_consts.h"

unsigned short states[NOS*2][4] = {
#include "vit7hamwts.h"
};

#include "vit_functions.h"

main(short int argc, char *argv[])
{

/* variables:
 *            c  - index arrays for appropriate 2 bit input
 *        u1, u2  - received 2 bit code symbols
 *         input  -
 */

  unsigned short c, u1, u2, input = 0;

/* read command line parameters
 */
  if (argc != 3) {
    cout << "function: 'viterbi7' decodes a K = 7, r =1/2 conv. coded "
       << "bitstream" << endl
       << "Usage: viterbi7 infile outfile" << endl;
    exit(1);
  }
  else {
```

```
    ifstream vitIn(argv[1], ios::in);
    if (!vitIn) {
      cerr << "Input file " << argv[1] << " could not be opened" << endl;
      exit(1);
    }

    ofstream vitOut(argv[2], ios::out);
    if (!vitOut) {
      cerr << "Output file " << argv[2] << " could not be opened" << endl;
      exit(1);
    }
/***/

/* initialize and allocate memory for all NOS path histories - this is a C++
 * construct (use malloc/delete for C implementation.
*/
    for (int p = 0; p <= (NOS - 1); p++) {
      pathHist[p] = new unsigned short[HISTORY];
      newPath[p] = new unsigned short[HISTORY];
    }

/* recover a pair of code symbols and determine index, c, from inputs u1, u2.
*/
    while (vitIn >> u1) {
      vitIn >> u2;
      c = 2*u1 + u2;

/* index Hamming distances for each pair of branches into each node, add
*  to respective source distance totals, and select one. (this doesn't make
*  much sense does it?)
*/
      findNewPath(c,input);
      updatePathHist(input);
      input++;

/* compare all of the paths and decode them after every CHECKPATH new branches.
*/
      if ((input % CHECKPATH) == 0) {
        input = decodeTrellis(input, vitOut);
      }
    }

/* when decoding finished, de-allocate and clear memory requirements - once
 * again, these are C++ commands.
*/
    for (p = 0; p <= (NOS -1); p++) {
      pathHist[p] = delete[];
      newPath[p] = delete[];
    }

    vitIn.close();
    vitOut.close();
    return 0;
  }
}
```

```
/***/


/* function decodes trellis path to a bit pattern after the paths have
 * converged some distance behind the current iteration.
 * arguments passed to function:
int decodeTrellis(int input, ofstream &toFile)
{
/* variables:
 *          flagA - set if only one node is common (ie. the first one) in
 *                  the paths and ensures it is not 'lost' between calls
 *        counter - tracks the number of common nodes
 *          flagB - '0' indicates paths have converged at least at one node
 *         hansel - common node used for bit identification
*/

  static short flagA = 0;
  unsigned short counter = 0, flagB = 1, hansel;

/* check for convergence in path histories.
*/
  while (flagB == 1) {

/* starting with the first node, check to see if any of the paths don't
 * begin w/ the same one.  If they don't, then the paths haven't converged
 * yet so set flagB and leave this function call.
*/
    for (unsigned short n = 0; n < (NOS - 1); n++) {

      if (*(pathHist[n] + counter) != *(pathHist[n+1] + counter)) {
        flagB = 0;
      }
    }

/* if a node is similar amongst all of the paths, then the paths have
 * converged in at least one location...check for more.
*/
    if (flagB == 1)
      counter++;
  }

/* if there are any path similarities starting from the beginning (among !all! * * the paths) then decode them.
*/

  if (counter > 0) {

    for (unsigned short p = 0; p < counter; p++) {
      hansel = *(pathHist[0] + p);

/* output the bits to output file.  Each bit is determined by the branch
 * between two nodes...if there is only one common node, a bit can't be
 * decoded (this is why flagA is set to '1' below)
*/
      if ((p > 0) || (flagA == 1)) {
        if (hansel <= (NOS/4 - 1)) {
```

```
        toFile << 0 << ' ';
      } else if ((hansel >= NOS/2) && (hansel < (3*NOS/4))) {
        toFile << 0 << ' ';
      } else {
        toFile << 1 << ' ';
      }
    }

/* set flagA to '1' to indicate only one node is in common and must be used
 * dufing the next function call to decode a bit.
*/
    else {
      flagA = 1;
    }
  }

/* update all paths: reset pointers to new starting nodes and clear previous
*  decoded history.
*/
  for (n = 0; n <= (NOS - 1); n++) {

    for (int p = counter; p < input; p++) {
      *(newPath[n] + p - counter) = *(pathHist[n] + p);
      *(pathHist[n] + p - counter) = *(newPath[n] + p - counter);
    }
  }

/* if a there was path convergence, return this...
*/
  return (input - counter);
 }

/* otherwise return this...
*/
 return input;
}


/*****************************************************************************
filename: vit_functions.h

contents: Function prototypes and definitions for Viterbi r = 1/2 decoder.
*****************************************************************************/

void findNewPath(int, int);
void updatePathHist(int);
int decodeTrellis(int, ofstream &);


/* findNewPath: finds the shorter of the two paths (using the Hamming distance)
 *          between each state and its two source states.  Then the total
 *          path weights are updated for the current time step.
*/
void findNewPath(int c, int input)
{
```

```
  unsigned short minValue, index;
  unsigned short stateTempWts[2] = {0};

  for (int n = 0; n <= (NOS - 1); n++) {

/* calculate both Hamming weights for a node during an interation and
 * eliminate the larger one.
*/
    stateTempWts[0] = states[n * 2][c] + pathWts[source[0][n]];
    stateTempWts[1] = states[(n*2)+1][c] + pathWts[source[1][n]];

    if (stateTempWts[0] <= stateTempWts[1]) {
      minValue = stateTempWts[0];
      index = 0;
    }
    else {
      minValue = stateTempWts[1];
      index = 1;
    }

/* update the path metric for the node based on the shorter path found
 * above.
*/
    newPathWts[n] = stateTempWts[index];

    for (int p = 0; p < input; p++) {
      *(newPath[n] + p) = *(pathHist[source[index][n]] + p);
    }
    *(newPath[n] + input) = source[index][n];
  }
}
/***/


/* updatePathHist: the new path of each of the states at time t is
 *              the path histories for the next time, t+1.
*/
void updatePathHist(int input)
{
  for (int n = 0; n <= (NOS - 1) ; n++) {
    for (int p = 0; p <= input; p++) {
      *(pathHist[n] + p) = *(newPath[n] + p);
    }
    pathWts[n] = newPathWts[n];
  }
}

/*****************************************************************************
filename: vit7_consts.h

contents: constants, transition state values, and global variables of
      Viterbi decoder for K = 7, r = 1/2 coder.
*****************************************************************************/

#define   NOS      64   /* number Of states - of encoder */
```

```
#define   HISTORY    40    /* retain this much path history through the
                           trellis for each state */
#define   CHECKPATH  15    /* check for similarity between paths once every
                           CHECKPATH timesteps */


/* source states for each of the 64 states - each state has only 2 possible
 * states of origin. Each column,i, represents the 2 source nodes for the
 * 'i'th node.
*/

unsigned short source[2][NOS] =
        {{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
          0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
          1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31,
          1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31},
         {32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
          32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
          33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63,
          33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63,}};

unsigned short pathWts[NOS] = {0}, newPathWts[NOS] = {0},
          *pathHist[NOS], *newPath[NOS];


/*****************************************************************************
filename: vit7hamwts.h

contents: the Hamming weights between the 4 possible input bit pairs (00, 01,
        10, and 11) and the correct bit pattern produced by the transition
        between a source node and the destination node.
*****************************************************************************/
 {0, 1, 1, 2},{2, 1, 1, 0},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {1, 0, 2, 1},{1, 2, 0, 1},
 {1, 2, 0, 1},{1, 0, 2, 1},
 {1, 2, 0, 1},{1, 0, 2, 1},
 {1, 0, 2, 1},{1, 2, 0, 1},
 {1, 0, 2, 1},{1, 2, 0, 1},
 {1, 2, 0, 1},{1, 0, 2, 1},
 {1, 2, 0, 1},{1, 0, 2, 1},
 {1, 0, 2, 1},{1, 2, 0, 1},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {2, 1, 1, 0},{0, 1, 1, 2},
 {0, 1, 1, 2},{2, 1, 1, 0},
 {0, 1, 1, 2},{2, 1, 1, 0},
```

{2, 1, 1, 0},{0, 1, 1, 2},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{2, 1, 1, 0},{0, 1, 1, 2},
{0, 1, 1, 2},{2, 1, 1, 0},
{0, 1, 1, 2},{2, 1, 1, 0},
{2, 1, 1, 0},{0, 1, 1, 2},
{2, 1, 1, 0},{0, 1, 1, 2},
{0, 1, 1, 2},{2, 1, 1, 0},
{0, 1, 1, 2},{2, 1, 1, 0},
{2, 1, 1, 0},{0, 1, 1, 2},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 2, 0, 1},{1, 0, 2, 1},
{1, 0, 2, 1},{1, 2, 0, 1},
{0, 1, 1, 2},{2, 1, 1, 0},
{2, 1, 1, 0},{0, 1, 1, 2},
{2, 1, 1, 0},{0, 1, 1, 2},
{0, 1, 1, 2},{2, 1, 1, 0},
{0, 1, 1, 2},{2, 1, 1, 0},
{2, 1, 1, 0},{0, 1, 1, 2},
{2, 1, 1, 0},{0, 1, 1, 2},
{0, 1, 1, 2},{2, 1, 1, 0}

APPENDIX C. COMPUTER CODE FOR BASEBAND MODULATOR AND QPSK SYSTEM

The first program below implements a simple baseband modulator with a BSC channel. The code following, is the main routine and header files of the full QPSK system. In the program **qpsk.C**, the header files containing the filter coefficients of the demodulator LPF and the matched filter are not included due to their length.

```
/*************************************************************************
filename: baseband.C
                        Baseband Modulator using BSC

 Function: implement a simple baseband modulator across BSC channel by
           applying user defined Pb to each bit

 Input: read in a binary file - unipolar unpacked binary
 Output: unipolar unpacked binary to a file
*************************************************************************/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>

#ifndef RANDMAX
#define RANDMAX 32767
#endif

main(int argc, char *argv[])
{
/* variables:
 *                  bit - input bit
 *               newBit - output bit
 *          errorMask - used to 'flip' a bit if an error occurs
 *               chance - calculate if input bit will undergo 'error'
 *              p_error - user input Pb
*/
  unsigned short bit, newBit
        errorMask=0;
  float chance, p_error = 0.0;

/* file i/o
*/

  if (argc != 3) {
    cout << "function: 'baseband' simulates a simple baseband channel "
       << "for any binary data" << endl
```

```
      << "Usage: baseband infile outfile" << endl << endl;
    exit(1);
  }
  else {
    ifstream channelIn(argv[1], ios::in);
    if (!channelIn) {
      cerr << "Input file " << argv[1] << "could not be opened "<< endl;
      exit(1);
    }

    ofstream bbOut(argv[2], ios::out);
    if (!bbOut) {
      cerr << "Output file " << argv[2] << "could not be opened" << endl;
      exit(1);
    }


/***/
    cout << "\nEnter the probability of bit error => ";
    cin >> p_error;

/* for each input bit, calculate a chance and cause an error if chanc <= Pb.
*/
    while (channelIn >> bit) {
      chance = (rand()/(float)RANDMAX) + p_error;
      errorMask = (unsigned short)floor(chance);

/* perform bit level XOR operation to change the bit if an errors as
 * determined above.
*/
      if (errorMask == 1) {
       bit = bit ^ errorMask;
      }

      bbOut << bit << " ";
    }

    bbOut.close();
    channelIn.close();
    return 0;
  }
}



/*************************************************************************
 filename: qpsk.C
                              Digital QPSK System

 Function: modulates, adds noise, and demodulates a data stream using  QPSK
           modulation
```

```
   Input: file (not packed) of unipolar binary data (0 or 1 only).
   Output: unpacked demodulated and detected data stream

   Associated header files:  qpskmod.h, demod.h, noise.h, bpf.h
 ************************************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>

#include "qpskmod.h"
#include "demod.h"
#include "noise.h"
#include "bpf.h"

double dlpf[N] = {
#include "dlpf.h"
};

double matched[NPB] = {
#include "mfcoeffs.h"
};


main(int argc, char *argv[])
{
/* variables
 *          Ibit, Qbit -      inphase/quadrature bits (modulate cos/sin)
 *          nPerBaud  -       number of samples per baud
 *          index     -       access sinusoid lookup table
 *          delay     -       signal delay through the system
 *          m         -       NPB (samples/baud)
 *          normalize -       scale sinusoids by 1/(sqrt2) so sum == 1
 *          inPhase/quadPhase - inphase/qphase sinusoids
 *          qpsk/qpskfilt     - original/filtered qpsk signals
 *          scale     -       std dev of AWGN
 *          noise     -       the AWGN sample (prior to filtering)
 *          rcvd      -       s[n] + noise[n] at demodulator input
 *          demodI/demodQ
 *          Imfout/Qmfout     - output from the two matched filters
 *          rI[], rQ[] - input buffers to I/Q demodulator LPFs
 *          Imfin[], Qmfin[]  - input buffers to the I/Q matched filters        */

 short Ibit, Qbit;
 unsigned short  nPerBaud,
          index, n,
          delay = DELAY, m = NPB;
 double normalize,
     inPhase, quadPhase,
     qpsk, qpskfilt,
```

```
      scale, noise,
      rcvd,
      demodI = 0.0, demodQ = 0.0,
      Imfout = 0.0, Qmfout = 0.0;
  double rI[N] = {0}, rQ[N] = {0},
      Imfin[NPB] = {0}, Qmfin[NPB] = {0};


/* Open input and output files */

  if (argc != 3) {
    cout << "function: 'qpsk' - QPSK simulation w/ awgn channel" << endl
        << "Usage: qpskmod infile outfile" << endl << endl;
    exit(1);
  }
  else {
    ifstream dataIn(argv[1], ios::in);
    if (!dataIn) {
      cerr << "Input file " << argv[1] << "could not be opened "<< endl;
      exit(1);
    }

    ofstream qpskS(argv[2], ios::out);
    if (!qpskS) {
      cerr << "Output file " << argv[2] << "could not be opened" << endl;
      exit(1);
    }

/***/

cout << "Enter the std of the noise: ";
cin >> scale;
cout << endl;


normalize = 1/sqrt(2);
index = 1;

/* input the two bits per QPSK symbol.
*/
while (dataIn >> Ibit) {
  dataIn >> Qbit;

/* convert to bipolar bit stream */
  Ibit = 2*Ibit - 1;
  Qbit = 2*Qbit - 1;

/* for NPB samples per baud, use the same Ibit and Qbit to modulate the qpsk
 * signal.
*/
  n = 1;
```

```
  while (n <= NPB) {

/* sinusoids are accessed from lookup table of 8 values (fs = 8*fc).
*/
    inPhase = Ibit * cosLookUp[(index + 1) % Nyq];
    quadPhase = Qbit * cosLookUp[(index - 1) % Nyq];
    qpsk = normalize * (inPhase + quadPhase);

/* generate an AWGN sample, add it to the QPSK value and filter both - they can
 * both be filtered separately and summed to 'more clearly' denote that the
 * noise is filtered prior to addition but this is faster (and since it is
 * a linear operation, it produces the same result.
*/
    noise = scale*gaussian();
    rcvd = filter(qpsk + noise);

/* INSERT interpolation and decimation routines here to more realistically
 * simulate an analog signal.
*/

/* demodulate the 'received' signal.
*/

/* INSERT phase locked loop routine here (as a do-while loop) to simulate
 * more realistic signal recovery.
*/
    rI[0] = rcvd * cosLookUp[(index + 1) % Nyq];
    rQ[0] = rcvd * cosLookUp[(index - 1) % Nyq];

    demodI = rI[(N-1)/2]*dlpf[(N-1)/2];
    demodQ = rQ[(N-1)/2]*dlpf[(N-1)/2];

    for (unsigned short int tap = 0; tap < (N - 1)/2 ; tap++) {
      demodI += (rI[tap] + rI[N-tap-1]) * dlpf[tap];
      demodQ += (rQ[tap] + rQ[N-tap-1]) * dlpf[tap];
    }

    for (unsigned short update = (N - 1); update > 0; update--) {
      rI[update] = rI[update - 1];
      rQ[update] = rQ[update - 1];
    }

    if (delay > 0) {
      delay--;
    }
/* used matched filters to 'detect' the I and Q bits. Interleave and output
 * the results to the output file.
*/
    else {
      m--;
      Imfout += matched[m]*demodI;
```

```
      Qmfout += matched[m]*demodQ;

      if (m == 0) {
        if (Imfout > 0) {
          qpskS << 1 << ' ';
        }
        else {
          qpskS << 0 << ' ';
        }

        if (Qmfout > 0) {
          qpskS << 1 << ' ';
        }
        else {
          qpskS << 0 << ' ';
        }

        Imfout = 0.0;
        Qmfout = 0.0;
        m = NPB;
      }
    }


    if (index == Nyq) {
      index = 0;
    }
    index++;
    n++;
  }
}

/* at the end of the data stream if there are values left in the matched filter
 * then output two zeros as the last symbol.
*/
if (m != NPB) {
  qpskS << 0 << ' ' << 0;
}

qpskS.close();
dataIn.close();
return 0;
}
}

/*************************************************************************
filename: qpskmod.h

contents: QPSK modulation/demodulation parameters
*************************************************************************/
```

```
#define   D     4800      /* symbol rate for data rate of 9600 bps */
#define     fc  455000      /* intermediate freq */
#define     fs  3640000     /* fs = 8fc */
#define   Nyq     8         /* sampling rate */

#define  DELAY    452      /* ! the drawback of not having a PLL ! */
#define    NPB    758      /* round(fs/D) */

/* digital filter parameters:
 *                 type: Butterworth BPF
 *           passband: fc +- 0.3*R ==> 452.12 kHz < pass  < 457.88 Hz
 *                order: 6th
 *          coefficients: see below (a0 at the top, a6 at the bottom)
*/

#define  TAPS     7

double a[TAPS] =
{
  1.00000000000000,
 -4.22863200584110,
  8.94059042311700,
-11.20184667623209,
  8.88152415382525,
 -4.17294333434056,
  0.98031108137184
},

b[TAPS] =
{
  0.00000012164818,
  0.0,
 -0.00000036494454,
  0.0,
  0.00000036494453,
  0.0,
 -0.00000012164818
},

cosLookUp[Nyq] =
{
  0.00000000000000,
  0.70710678118655,
  1.00000000000000,
  0.70710678118655,
  0.00000000000000,
 -0.70710678118655,
 -1.00000000000000,
 -0.70710678118655
};
```

```
/* function prototype for BPF filter function.
*/
double filter(double);


/*****************************************************************************
filename: demod.h

contents: function prototypes for noise generator and constant definition of
        demodulator LPF
*****************************************************************************/
#define     N           37   /* number of taps of demodulator LPF */

double gaussian();
double noiseFilter(double);



/*****************************************************************************
filename: bpf.h

contents: filter definition of BPF used on qpsk signal and AWGN
*****************************************************************************/

/* filter - a Butterworth BPF channel filter.  For qpsk it removes spectral
 *          sidelobes; and acts as noise filter.
*/
double filter(double input)
{
/* variables:
 *              yHist[], xHist[] - retain input/ouptut filter history
*/
  static double yHist[TAPS] = {0},
                xHist[TAPS] = {0};

  double output = 0.0;

  xHist[0] = input;
  yHist[0] = b[0]*xHist[0];

  for (unsigned short int i = 1; i < TAPS; i++) {
    yHist[0] = yHist[0] + b[i]*xHist[i] - a[i]*yHist[i];
  }

  output = yHist[0];

/* update filter histories after each step.
*/
  for (unsigned short update = (TAPS - 1); update > 0; update--) {
    xHist[update] = xHist[update - 1];
    yHist[update] = yHist[update - 1];
```

```
  }

  return output;
}

/*****************************************************************************
filename: noise.h

contents: function definitions used to generate AWGN
          NOTE: this source code obtained from Embree [Ref. 13]
*****************************************************************************/

/* gaussian - generates zero mean unit variance Gaussian random variables.
 *            Uses Box-Muller transformation of two uniform r.v.
*/

double gaussian()
{
  static int ready = 0;       /* flag to indicate stored value */
  static double gstore;       /* place to store other value */
  double v1, v2, r, fac, gaus;

  double uniform();

/* make two numbers if none stored.
*/
  if (ready == 0) {
    do {
      v1 = 2.*uniform();
      v2 = 2.*uniform();
      r = v1*v1 + v2*v2;
    } while(r > 1.0);       /* for radius less than 1 */

/* remap v1 and v2 to Gaussian numbers.
*/
    fac = sqrt(-2.*log(r)/r);
    gstore = v1*fac;        /* store one of the two */
    gaus = v2*fac;          /* return the other one */
    ready = 1;              /* set ready flag */
  }
  else {
    ready = 0;              /* reset ready flag for next pair */
    gaus = gstore;          /* return the stored one */
  }

  return gaus;
}

/* uniform - generates zero mean uniform rv from -0.5 to 0.5.  This function
 *           is called by gaussian().
*/
```

```
#ifndef RAND_MAX
#define RAND_MAX 32767
#endif

double uniform()
{
  return ((double)(rand() & RAND_MAX)/RAND_MAX - 0.5);
}
```

APPENDIX D. SEQUENCE OF COMMANDS TO RUN QPSK SIMULATION ON CELP CODED SPEECH

The following list of commands must be run from a shell tool on a SUN SPARCstation in the order presented. Although the commands are given for CELP coded data, any data that is in unipolar binary format can be run through the channel encoders/ decoders and QPSK system. Below, the input file is assumed to be an audio file in the SUN '.au' format. The shell command line arguments are in bold followed by a brief description of the function. For commands with two filename arguments, the first file is the input and the second is the output. Except as noted, the filename suffixes can be anything that clarifies the file content to the user - the ones shown are merely representative and used for clarity.

1) **format_enc -3 <*filename*.au | format_dec -3 -l >*filename*.spd** - this function converts the 8 bit μ-law audio file to 16 bit linear PCM using the CCITT G.723 24kbps ADPCM coder as the conversion interface. The '.spd' suffix on the output file is required for the following CELP encoder. Note that there are no spaces between the {<,>} symbols and the filenames.

2) **celp_encode -i *filename*.spd -o *filename2*** - when the coder executes, it generates a '.log' file, which contains detailed information on the performance of the coder during this run, and three decoded files using different types of output filters all of which have an '.spd' filename suffix - these four files can be discarded. It also produces a channel file, *filename2*.chan for intermediate processing. This is the file of interest. It is in packed hexadecimal format with 36 characters/line (frame).

3) **hex2bin *filename*.chan *filename*.bin** - converts each hex character in the input file into its 4 bit binary form and outputs the results to a file with a single space between each binary value.

4) **fec7enc *filename*.bin *filename*.K7** - applies a $K = 7$, $r = 1/2$ convolutional code to the input data and outputs the coded sequence to a file. The last bit in the input sequence is flushed through the register with 6 zeros to fully involve it in the encoding procedure. A $K = 5$ coder may be substituted her: the command is **fec5enc** followed by equivalent filename notation.

5) **qpsk *filename*.K7 *filename*.qpsk** - converts the input unipolar {0,1} data into bipolar {-1,1} data and performs QPSK simulation. The data isfirst modulated at $f_c = 455$ kHz using an $f_s = 3640$ Hz, then filtered and summed with bandlimited AWGN. The user enters the standard deviation of the AWGN, σ_{AWGN}, at the user prompt. Demodulation is performed with two 36th order FIR LPFs (one each for the I

and Q streams). Detection is by means of two matched filters of length 758 (= number of samples per baud). All filter coefficients are in header files and are valid only for the given carrier frequency and simulation frequency. All intermediate values are in the simulation use double floating point precision - output data to file is in unipolar binary format.

Instead of performing a QPSK simulation, a baseband simulation across a BSC can be performed by using the command **baseband** followed by the input and output files.

6) **viterbi7** ***filename*.qpsk** ***filename*.v7** - performs hard decision Viterbi decoding on $K = 7$, $r = 1/2$ convolutional codes. If a $K = 5$ coder is used, **viterbi5** must be used to decode the sequence.

7) **bin2hex** ***filename*.v7** ***filename*2.chan** - converts unpacked unipolar data back to packed hex format for the CELP decoder. The data is rearranged into lines of 36 characters per line (corresponding to one coded CELP frame).

8) **celp_decode -c** ***filename*.chan -o** ***filename2*** - decodes the input channel file into three output files. Two of these are filtered versions and will have the suffixes *filename2*npf.spd and *filename2*hpf.spd - these can be discarded. The third file *filename2.spd* is the file of interest and it is in 16 bit linear PCM format.

9) **format_enc -3 -l <*filename*.spd | format_dec -3 >*filename*.r.au** - converts the 16 bit linear PCM file to 8 bit μ-law compressed data (without a header). The '.r' in the filename is used to indicate that it represents received information. The user should used different filenames to avoid overwriting the original file.

10) **raw2audio** ***filename*.r.au** - appends SUN audio file header so it can be recognized and played by a SPARCstation.

Any data in unipolar binary format can be input to the system at step 4 and accessed again after step 6 but the data requires source coding/decoding software and interface software as necessary to make the information suitable for the simulation.

APPENDIX E. CALCULATION OF E_b/N_o

The following MATLAB script computes the E_b/N_o parameter in dB and corresponding P_b for a given AWGN noise variance. The measurements are taken at the input to the matched filter detectors.

```
%
% filename: ebno.m
% function: calculates Eb/No for different noise variances -
%           change 'scale_noise' value to measure for different noise levels.
%

clear

fc = 455000;                    % carrier freq.
fs = 8 * fc;                    % sampling frequency
BR = 9600; % bit rate into modulator (= 4800 baud)

W = ([-0.3*BR 0.3*BR] + fc)/(fs/2);     % channel passband is fc +- 0.3xBitrate
[b1,a1] = butter(3,W);                  % design Butterworth filter (6th order)

points = 200000;            % test points
scale_noise = 7;            % variance = scale_noise ^ 2 (AWGN power)
noise = scale_noise*randn(1,points);

channelNoise = filter(b1,a1,noise);     % bandpass filter the AWGN

b = fir1(36,fc/(fs/2));                 % demodulator LPF
detectedNoise = filter(b,1,channelNoise);  % the output of this filter is
                                           %  noise value at the matched filter
                                           %  input.

B = 5760;                       % bandwidth of channel
No = std(detectedNoise)^2/B;    % average power of noise at detector

Eb = 0.00002;                   % energy per bit (Joules) calculated over
                                % 500000 points on I and Q channels

EbNo = 10*log10(Eb/No);         % dB value of Eb/No
Pb = q(sqrt(2*Eb/No));          % Pb = Q(sqrt(2*Eb/No))
```

LIST OF REFERENCES

1. Gibson, Jerry D., *Principle of Digital and Analog Communications*, Macmillan Publishing Company, New York, NY, 1993.

2. Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, PTR Prentice-Hall, Englewood Cliffs, NJ, 1988.

3. Tranter, W. H. and Ziemer, R. E., *Principles of Communications: Systems, Modulation, and Noise*, John Wiley and Sons, New York, NY, 1995.

4. Deller, John R., Jr., et. al., *Discrete-Time Processing of Speech Signals*, Macmillan Publishing Company, New York, NY, 1993.

5. Frerking, Marvin E., *Digital Signal Processing in Communication Systems*, Van Nostrand Reinhold, New York, NY, 1994.

6. Federal Standard 1016, "Telecommunications: Analog To Digital Conversion of Radio Voice by 4800 Bit/Second Code Excited Linear Prediction (CELP)," February 14, 1991.

7. Therrien, Charles W. *Discrete Random Signals and Statistical Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1992.

8. Roden, Martin S., *Analog and Digital Communication Systems*, Prentice Hall, Upper Saddle River, NJ, 1996.

9. Richharia, M. *Satellite Communication Systems: Design Principles*, Mc-Graw Hill, Inc., New York, NY, 1995.

10. Freeman, Roger L, *Telecommunication Transmission Handbook*, John Wiley & Sons, New York, NY, 1991.

11. Couch, Leon W., III, *Digital and Communication Systems*, Macmillan Publishing Company, New York, NY, 1993.

12. Krauss, Thomas P., et. al., *Signal Processing Toolbox User's Guide*, The MathWorks, Inc., Natick, MA, 1994.

13. Embree, Paul M. and Kimble, Bruce, *C Language Algorithms for Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1991.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library, Code 013 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Prof. Murali Tummala, Code EC/Tu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	4
5.	LCDR Michael K. Shields, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
6.	Prof. Herschel H. Loomis, Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
7.	Commander K. Webb, Code SPAWAR 72 Space and Naval Warfare Systems Command Crystal Park #5, 2451 Crystal Dr. Arlington, VA 22202-5100	2
8.	CDR D. Gear, Officer in Charge NISE EAST Detachment Washington 3801 Nebrask Ave N.W. Washington, DC 20393	2

9. LCDR Gregory H. Skinner 1
Advanced Maritime Projects Office
Building 659, Box 51
Naval Air Station
Jacksonville, FL 32212

10. LT Bruce E. Watkins 1
NCCOSC RDT&E DIV
53560 Hull St.
San Diego, CA 92152-5001

11. LT Dilip B. Ghate 2
3105 Wyndham Ave.
Richardson, TX 75082