RESIDUAL-EXCITED LINEAR PREDICTIVE (RELP) VOCODER SYSTEM WITH TMS320C6711 DSK AND VOWEL CHARACTERIZATION

A Thesis Submitted
to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Electrical Engineering
University of Saskatchewan

by

Akihiro Taguchi

Saskatoon, Saskatchewan, Canada

PERMISSION TO USE

The author has agreed that the Libraries of this University may make this thesis freely available for inspection. Moreover, the author has agreed that permission for copying of this thesis, in any manner, in whole or in part, for scholarly purposes may be granted by the professors who supervised this thesis work or, in their absence, by the Head of the Department of Electrical Engineering or the Dean of the College of Graduate Studies and Research at the University of Saskatchewan. It is understood that due recognition will be given to the author and the University of Saskatchewan in any use of the material in this thesis. Copying, publication, or use of this thesis or parts thereof for financial gain shall not be allowed without the author's written permission.

Request for permission to copy or to make any other use of material in this thesis in whole or in part should be addressed to:

Head of the Department of Electrical Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

ACKNOWLEDGMENTS

I wish to acknowledge the guidance and assistance of my supervisor, Dr. Kunio Takaya. First of all, I am sincerely grateful that he gave me the opportunity to work on this research. I also deeply appreciate his support, patience and encouragement throughout the research.

Financial assistance was provided by The Telecommunications Research Laboratories (TRLabs). TRLabs also provided equipment and facilities for this research. This assistance is gratefully acknowledged. I would like to thank faculty, staff and students of TRLabs Saskatoon for their support. I also thank faculty, staff and fellow students of the Department of Electrical Engineering who have helped me.

I wish to thank the very special people in my life. I could not have accomplished this work without the endless support and love of my wife, Elaine. Thank you to my parents in Japan. I can not express with words my appreciation of them.

UNIVERSITY OF SASKATCHEWAN

Electrical Engineering Abstract

RESIDUAL-EXCITED LINEAR PREDICTIVE (RELP) VOCODER SYSTEM WITH TMS320C6711 DSK AND VOWEL CHARACTERIZATION

Student: Akihiro Taguchi

Supervisor: Prof. K. Takaya

M.Sc. Thesis Submitted to the

College of Graduate Studies and Research

December 2003

ABSTRACT

The area of speech recognition by machine is one of the most popular and complicated subjects in the current multimedia field. Linear predictive coding (LPC) is a useful technique for voice coding in speech analysis and synthesis. The first objective of this research was to establish a prototype of the residual-excited linear predictive (RELP) vocoder system in a real-time environment. Although its transmission rate is higher, the quality of synthesized speech of the RELP vocoder is superior to that of other vocoders. As well, it is rather simple and robust to implement. The RELP vocoder uses residual signals as excitation rather than periodic pulse or white noise. The RELP vocoder was implemented with Texas Instruments TMS320C6711 DSP starter kit (DSK) using C.

Identifying vowel sounds is an important element in recognizing speech contents. The second objective of research was to explore a method of characterizing vowels by means of parameters extracted by the RELP vocoder, which was not known to have been used in speech recognition, previously. Five English vowels were chosen

for the experimental sample. Utterances of individual vowel sounds and of the vowel sounds in one-syllable-words were recorded and saved as WAVE files. A large sample of 20-ms vowel segments was obtained from these utterances. The presented method utilized 20 samples of a segment's frequency response, taken equally in logarithmic scale, as a LPC frequency response vector. The average of each vowel's vectors was calculated. The Euclidian distances between the average vectors of the five vowels and an unknown vector were compared to classify the unknown vector into a certain vowel group.

The results indicate that, when a vowel is uttered alone, the distance to its average vector is smaller than to the other vowels' average vectors. By examining a given vowel frequency response against all known vowels' average vectors, individually, one can determine to which vowel group the given vowel belongs. When a vowel is uttered with consonants, however, variances and covariances increase. In some cases, distinct differences may not be recognized among the distances to a vowel's own average vector and the distances to the other vowels' average vectors. Overall, the results of vowel characterization did indicate an ability of the RELP vocoder to identify and classify single vowel sounds.

Table of Contents

List of Figures

List of Tables

ABBREVIATIONS AND ACRONYMS

ADC	Analog-to-Digital Conversion
ADM	Adaptive Delta Modulation
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
AR	Autoregressive
CCS	Code Composer Studio
CELP	Code Excited Linear Prediction
DAC	Digital-to-Analog Conversion
dB	decibel
DMA	Direct Memory Access
DoD	U.S. Department of Defense
DSK	DSP starter kit
DSP	Digital Signal Processor
EMIF	External Memory InterFace
FIR	Finite Impulse Response
GIE	Global Interrupt Enable bit
HPI	Host Port Interface
Hz	Hertz (1 cycle/second)
IDE	Integrated Development Environment
IER	Interrupt Enable Register
IFR	Interrupt Flag Register
IIR	Infinite Impulse Response
IMH	Interrupt Multiplexer High register
IML	Interrupt Multiplexer Low register
ISR	Interrupt Service Routine
kB	kilobytes

kbps	kilobits per second
LPC	Linear Predictive Coding
MB	Megabytes
McBSP	Multichannel Buffered Serial Port
MELP	Mixed Excitation Linear Prediction
MFLOPS	Million Floating-point Operations Per Second
MIPS	Million Instructions Per Second
NMIE	Non-Maskable Interrupt Enable bit
RELP	Residual Excited Linear Prediction/Predivtive
PARCOR	Partial Correlation
RAM	Random-Access Memory
ROM	Read-Only Memory
SDRAM	Synchronous Dynamic RAM
TI	Texas Instruments
VLIW	Very-Long-Instruction-Word
Vocoder	Voice Coder

1. INTRODUCTION

1.1 Background of Research

In speech analysis and in speech synthesis, Linear Predictive Coding (LPC) is one of the most powerful techniques and one of the most useful methods for voice coder (vocoder) systems. Vocoder systems encode and decode good quality speech signals at a low bit rate.

Human speech can be divided into two categories: voiced sounds and unvoiced sounds. Voiced sounds occur when air is forced from the lungs, through the vocal cords, and out of the mouth or nose. Unvoiced sounds are generated by forming a constriction at some point in the vocal tract, such as the teeth or lips, and forcing air through the constriction to produce turbulence.

The basic premise of the LPC method is that a speech sample can be approximated as a linear combination of previous speech samples in a short time segment. The vocoder then models the vocal tract as a time-invariant, all-pole digital filter. It also utilizes a voiced/unvoiced decision switch and pitch detection to produce the excitation signals.

The basic LPC vocoder produces intelligible synthesized speech signals at a low bit rate of 2.4 kilobits/second (kbps), but the speech signals are not natural, they have a buzzer-like quality.

Many researchers have presented different types of LPC-based vocoder systems such as Code Excited Linear Prediction (CELP) and Mixed Excitation Linear Prediction (MELP). The intent of these systems is to obtain a high compression rate while maintaining the quality of synthesized speech. It is common knowledge that this accomplishment is a difficult task. Voice over IP (Internet telephone) uses CELP and its

variations that provide moderate telephonic quality speech. However, due to today's rapidly evolved broadband communication environment and emerging sophisticated data compression techniques, it is also a fact that the importance of compressing the speech signal is decreasing to some extent.

The Residual Excited Linear Predictive (RELP) vocoder is one of the LPC-based vocoders. Although the compression rate is moderate, because a sequence of residual signals is needed to excite speech signals, the quality of the synthesized speech is far better than the basic LPC, CELP and MELP vocoders. The system is robust [26], since there is no need to analyze whether the sound is voiced or unvoiced nor to analyze the pitch period.

In the multimedia environment, content-based speech response applications, such as speech-to-text conversion and voice response systems, presently attract a great deal of attention. When one tries to use the RELP vocoder for content sensitive applications, it lacks information about whether parameters associated with the RELP vocoder are capable of determining what word was spoken. However, the frequency response of the LPC filter obtained by the RELP encoding process is a low order approximation of the signal spectrum of speech. This indicates that parameters extracted by the RELP vocoder do have a potential ability to recognize human speech.

1.2 Objectives of Research

The first objective of this research was to establish a prototype of the Residual Excited Linear Predictive (RELP) vocoder system with the recent digital signal processor (DSP) development system, Texas Instruments' TMS320C6711 DSP starter kit (DSK), in real-time.

The system should extract all fundamental parameters, such as linear prediction coefficients, reflection coefficients, and prediction errors, from analog speech signals in real-time at the encoding process. As well, it should be able to synthesize speech signals with the obtained parameters at the decoding process.

Identifying vowel sounds is one of the most important elements in recognizing speech contents. The second objective was to explore a method of characterizing the vowels in human speech by means of parameters extracted by the RELP vocoder system. Five English vowels were chosen for the experimental sample. Utterances of individual vowel sounds and of the vowel sounds in one-syllable-words by a same speaker were used. The method utilized Euclidian distance defined by the frequency response of the LPC filter as a criterion to classify an unknown vowel segment into one of the five vowel groups.

1.3 Structure of Thesis

This thesis is organized into seven chapters.

Chapter 1 introduces a brief background and the objectives of this research.

Chapter 2 discusses the background of this research. Included are a brief discussion of speech signals and the structure of the Linear Predictive Coding (LPC) vocoder, and an outline of the Residual Exited Linear Predictive (RELP) vocoder. Some variations of the generic LPC-based vocoder systems are also discussed. Further, methods applicable to the vowel classification are introduced.

Chapter 3 describes the algorithm of linear prediction related to the linear predictive coding (LPC) including autoregressive (AR) model, the Levinson recursion, lattice filter structure, and spectrum estimation based on linear models. As well, an algorithm of the RELP vocoder is discussed.

In Chapter 4, the detailed implementation of the RELP vocoder is illustrated. An overview is presented of the Digital Signal Processor (DSP) development system, which is used to implement the RELP vocoder.

Chapter 5 presents the observed characteristics of the vowel sounds used for study in this research. Also presented are procedures to classify vowels recognized by the RELP vocoder.

The performance of the implemented RELP vocoder system and the results of vowel classification are discussed in Chapter 6.

Research conclusions and suggestions for further study are found in Chapter 7.

Appendix A describes the mathematical framework of linear prediction. The following are discussed: the basic principle of orthogonality, which is essential to the problem of linear prediction; the AR model; backward linear prediction; and the Levinson recursion.

Appendix B provides a list of all C programs that are used to implement the RELP vocoder with TMS320C6711 DSK.

2. BACKGROUND

Linear Predictive Coding (LPC) is one of the most popular techniques to analyze and synthesize human speech signals. The characteristics of this method are the accurate estimation of speech and the relatively small computational resource requirements [18]. This chapter opens with a discussion of human speech signals, then introduces the structure of the LPC vocoder, some of the existing LPC-based vocoder systems, and the Residual-Excited Linear Predictive (RELP) vocoder.

2.1 Speech Signals

In digital signal processing, there are applications which deal with human speech such as speech synthesis, speech coding, and speech recognition. For most people, speech is the primary method for the communication of language. Speech carries message information of the speaker in acoustic waveform. The information in the speaker's mind is transformed into neural signals which control articulators such as the glottis, vocal cords, lips and tongue in performing a series of physical movements. This action produces acoustic waves, of which a pressure waveform represents the initial information that the speaker wishes to express.

Most languages can be described in terms of a set of distinctive sounds, or phonemes, generally between 30 and 50 in number. There are about 42 phonemes in English, for instance. These phonemes can be classified as either voiced or unvoiced.

As stated in Chapter One, voiced sounds, such as vowels and certain consonants, e.g. /v/, /b/ and /z/, occur when air is forced from the lungs, through the vocal cords, and out of the mouth or nose. In response to varying muscle tension, the vocal cords vibrate at frequencies between 50 Hz and 1 kHz, resulting in periodic pulses of air which excite the vocal tract.

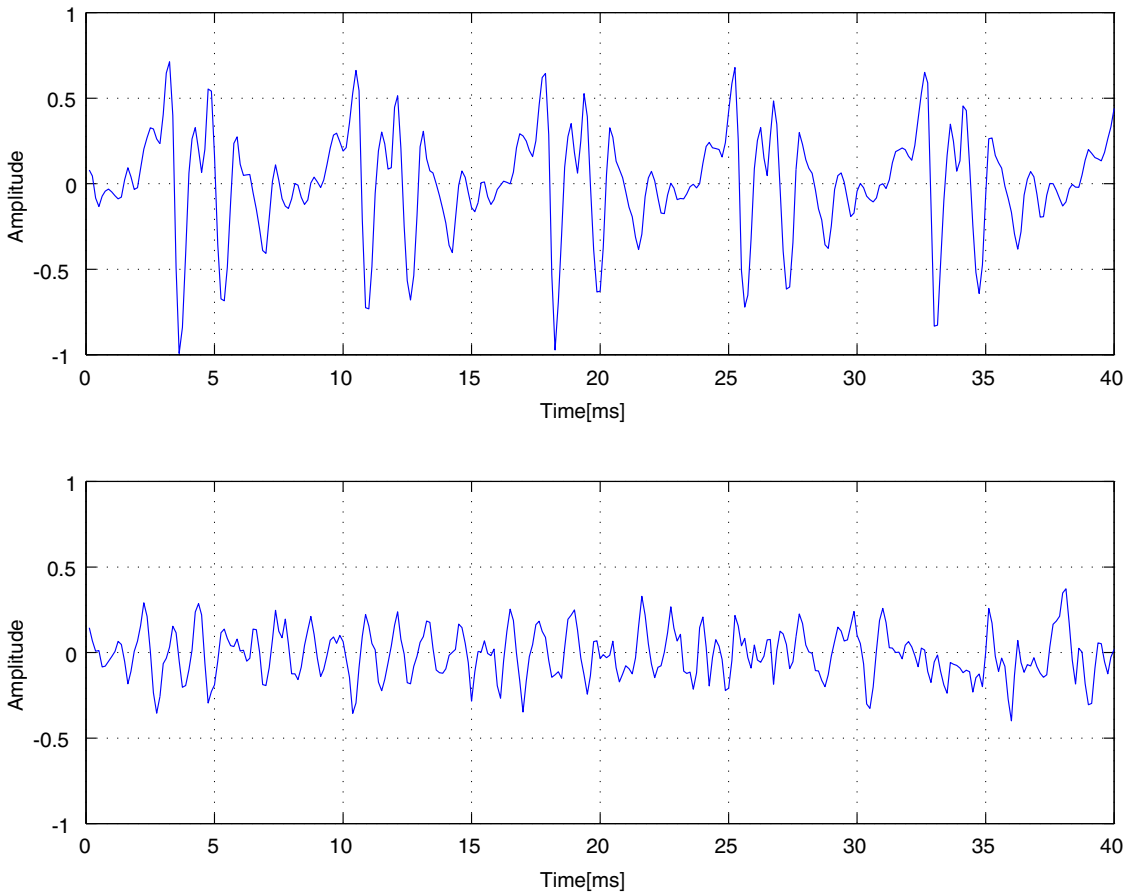


Figure 2.1 Analog representation of voiced [ʌ] as in 'run' (top) and unvoiced [s] as in 'sit' (bottom)

Unvoiced sounds, like /f/, /s/, /p/, /th/ as in thin and /k/, are generated by forming a constriction at some point in the vocal tract, such as the teeth or lips, and forcing air through the constriction to produce turbulence. This is regarded as a broad-spectrum noise source to excite the vocal tract. In other words, the sounds which use vibration of the vocal cords creating periodic pulses are called voiced and the sounds without vibration of the vocal cords are called unvoiced.

Another important characteristic of speech signals is the transfer function of the vocal tract. The vocal tract resembles a sound horn whose shape varies continuously with the movement of articulators. Therefore, the vocal tract can be modeled as a time-varying linear system. Its varying speed is, however, relatively moderate. In a short time segment, about 20 milliseconds (ms), the transfer function of the vocal

tract can be considered as equal to a time-invariant system.

Figure 2.1 illustrates typical examples of a voiced sound and an unvoiced sound. All y-axes on the graphs of speech signals in this thesis are relative to the amplitude gain. The input speech signals were sampled at a sampling rate of 8 kHz. 320 samples at 8 kHz (125 μs in each sampling interval) is equivalent to the time span of 40 ms. The top graph of Figure 2.1 shows part of a voiced sound [ʌ] as in 'run' for 40 ms. The waveform is almost periodic. On the other hand, a waveform of the unvoiced sound [s] as in 'sit' in the bottom graph of Figure 2.1 is random noise-like and there is no obvious sign of periodicity.

2.2 Linear Predictive Coding (LPC) Vocoder

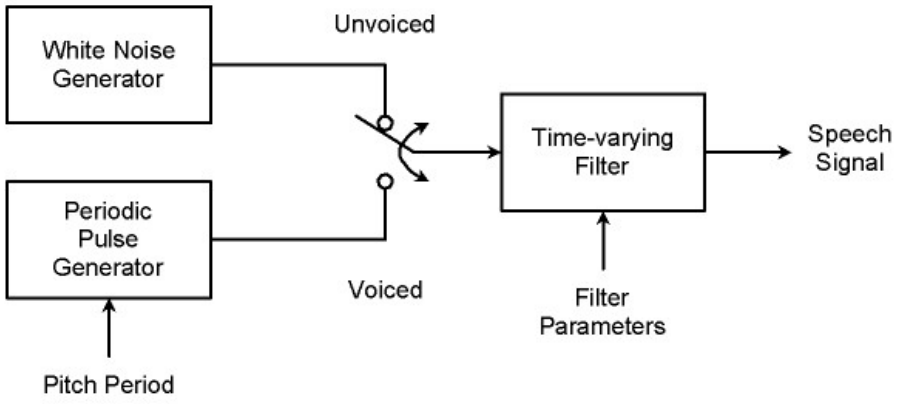


Figure 2.2 Simplified model of the generation of a speech signal

The generation of a speech signal can be modeled from the characteristics of human speech signals discussed in the previous section. The model in Figure 2.2 illustrates two sound source generators: a white noise generator and a periodic pulse generator; a voiced and unvoiced decision switch; and a time-varying filter. Although the vocal tract is considered as a time-varying filter, during a short-time segment the transfer function of the vocal tract is assumed to be time-invariant. Therefore, by changing the filter parameters every short-time segment, one of the sound source generators excites the filter and a speech signal is generated. Switching between the

two generators depends on a voiced or unvoiced decision. This model can be applied to the synthesis part of the LPC vocoder.

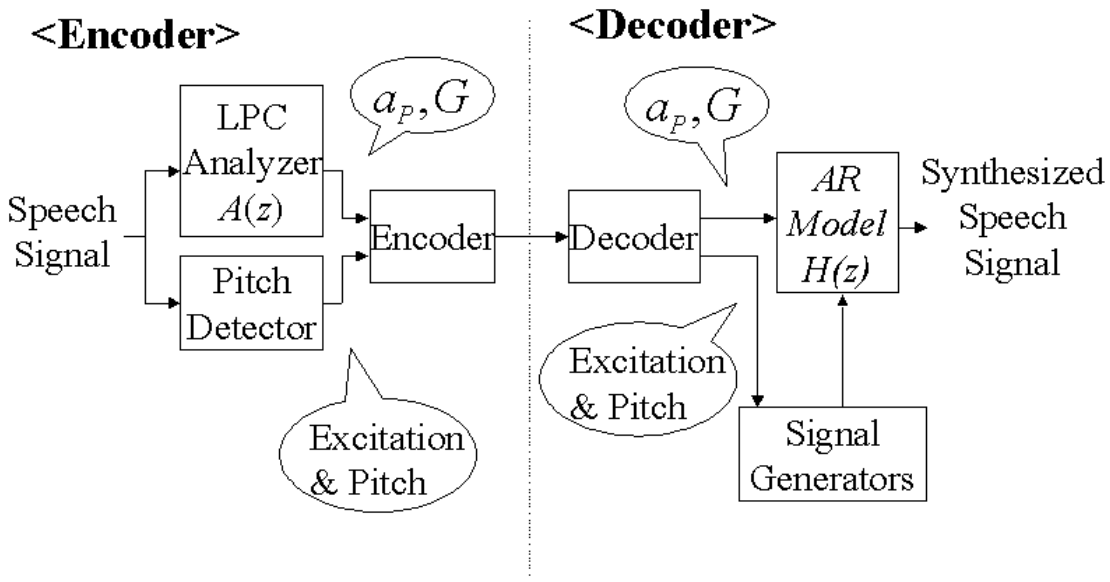


Figure 2.3 Encoder and decoder for LPC vocoder

Figure 2.3 is a block diagram of the LPC vocoder that consists of two parts, an encoder and a decoder. At the encoder, a speech signal is divided into short-time segments. Each speech segment is analyzed. The filter coefficients a_P and the gain G are determined at the LPC analyzer $A(z)$, which will be discussed later. The pitch detector detects whether the sound is voiced or unvoiced. If the sound is voiced, the pitch period is determined by the pitch detector. Then, all parameters are encoded into a binary sequence.

At the decoder, the transmitted data is decoded and the signal generators generate excitation signals, periodic pulses or white noise, depending on the voiced or unvoiced decision. This excitation signal goes through the Autoregressive (AR) Model $H(z)$ with a_P and G as the filter parameters, and then a synthesized speech signal is produced at the output of the filter.

Figure 2.4 illustrates the LPC analyzer. The linear predictive coding (LPC) approximates a speech sample as a linear combination of previous speech samples. LPC uses Equation 2.1 to estimate the current value $\hat{x}[n]$ from P previous values of a

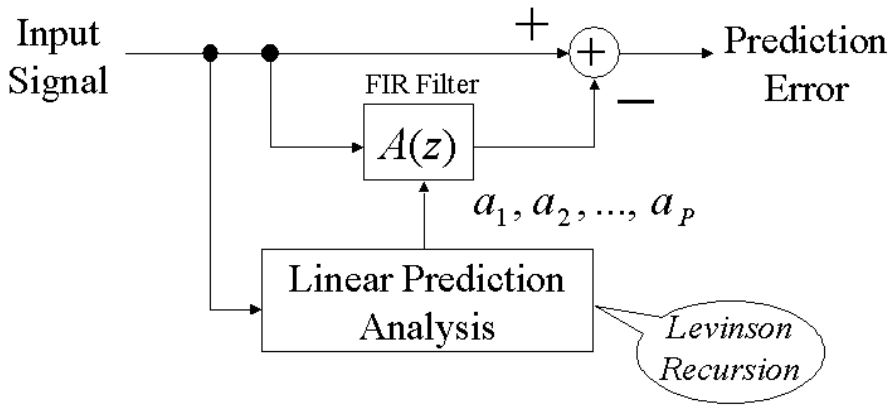


Figure 2.4 LPC analyzer

sample sequence $x[n]$ [24].

$$\hat{x}[n] = -a_1 x[n-1] - a_2 x[n-2] - \cdots - a_P x[n-P] \tag{2.1}$$

The estimation error, or the prediction error, which is the difference between the actual signal and the predicted signal, is given by

$$\varepsilon[n] = x[n] - \hat{x}[n] \tag{2.2}$$

or

$$\varepsilon[n] = x[n] + a_1 x[n-1] + a_2 x[n-2] + \cdots + a_P x[n-P] \tag{2.3}$$

A unique set of the prediction filter coefficients a_1 to a_P can be determined by solving the Normal equations, whose solution gives the optimal coefficients, to minimize the sum of the prediction error. To obtain these coefficients, an algorithm known as Levinson recursion is used. This algorithm provides a fast method for solving the Normal equations. Linear prediction, Levinson recursion and related matters will be discussed in Chapter 3.

If the order P of the filter is in the range of 8 to 10, then $A(z)$ approximates the input signal well, and the error becomes nearly a random signal. When the input is a speech signal, the ideal output of this filter is white noise or periodic pulses, depending on whether the sound is voiced or unvoiced. Therefore, the input speech signal can be represented by the filter coefficients.

The inverse of the Finite Impulse Response (FIR) filter $A(z)$, called Autoregressive (AR) model, $H(z)$, is used at the synthesis filter of the decoder. The AR model is one of the common methods to generate a target waveform by exciting $H(z)$ by a random signal. The speech signals are obtained from the output of the AR model $H(z)$ excited by a source signal, either a white noise or periodic pulses. The AR model is obtained by inverting the prediction error filter $A(z)$.

$$H(z) = \frac{1}{A(z)} \tag{2.4}$$

This model will then produce a synthesized signal with the same characteristics as those of the original signal. Figure 2.5 shows the relationship between the prediction error filter and the AR model.

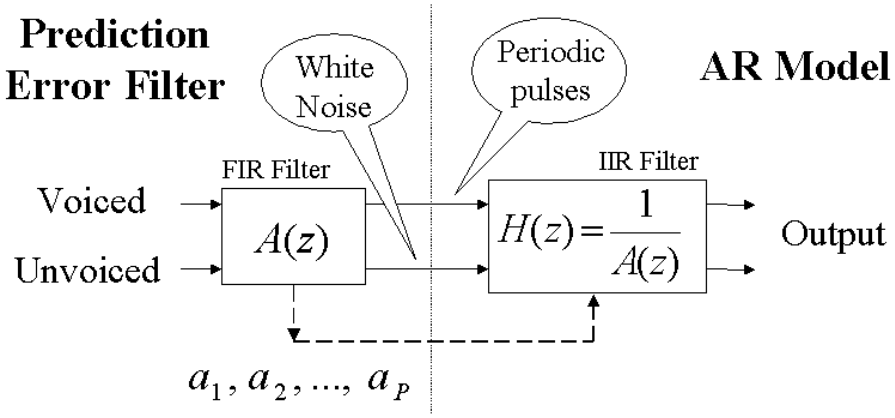


Figure 2.5 Relationship between prediction error filter and AR model

There are some drawbacks of LPC. A simple periodic pulse as an excitation signal for voiced speech has limitations in reproducing complex human speech sounds. It is also very difficult to recognize and distinguish human speech as voiced or unvoiced perfectly. These problems result in synthesized speech having a strong synthetic quality. It sounds buzzy and mechanical or has annoying thumps and tonal noises [9].

LPC was employed by the US Department of Defense (DoD) as a standard for secure communications over dial-up telephone lines in 1975 [4]. This standard is called Federal Standard FS-1015 known as LPC-10. This 10^{th} order LPC system encodes

at 2.4 kbps and transmits a segment of 54 bits every 22.5 ms. Each segment consists of four parameters: 41 bits for 10 prediction error coefficients, 7 bits for pitch, 5 bits for gain, and 1 bit for framing. The excitation signal for voiced speech is produced by repeating a prescribed waveform that resembles a glottal pulse [25].

2.3 Existing LPC-based Vocoder Systems

Since the basic LPC vocoder had room for further improvement in terms of the quality of sound and the compression rate, a variety of LPC-based vocoders have been developed. Two of these commonly used LPC-based vocoders are introduced in the following subsections.

2.3.1 Code Excited Linear Prediction (CELP) Vocoder

Code Excited Linear Prediction (CELP) vocoders have been adopted as standards for digital transmission of voice on cellular radio systems in North America, Europe, and Japan. The CELP vocoder is discussed by Atal et al [20]. The CELP vocoders enable high quality synthesized speech as low as 4.8 kbps for 8 kHz sampling rate [7]. The CELP vocoder of 4.8 kbps compression rate was standardized by DoD in 1991 [4].

It is known that the estimation error (residual) signals of speech approach Gaussian noise when linear prediction analysis is accurate. The CELP vocoder is a method that quantizes the estimation error signals into vectors using the Gaussian noise codebook. It encodes 40 samples of a residual signal into a vector at 8 kHz sampling rate. 40 samples at this rate are a block of 5-ms duration. These 40 samples are represented by one of 1024 vectors in the codebook. The length of a code is just 10 bits since 1024 is 2^{10}. Since one vector in the codebook refers to a corresponding residual signal of 40 samples, the CELP vocoder compresses the 40 samples down to a 10-bit word [28].

The encoder of the CELP system analyzes a speech signal with the LPC analyzer to obtain the residual signals and the prediction filter coefficients. Then it calculates the error comparing the residual signal (40 samples) with a vector in the Codebook and repeats this process 1024 times. After this comparison process, it chooses an

optimal vector in the Codebook to match the 40-sample block of the residual signal. This process is done every 40 samples. The decoder chooses a residual sequence from the same Codebook as the encoder, using the received vector as a common index, then synthesizes the speech signal.

The CELP vocoder can achieve a higher quality of synthesized speech than the basic LPC vocoder. It fully compensates for the smaller compression rate of 4.8 Kbps than the rate of the basic LPC vocoder. However, to obtain an optimal vector, establishing a Codebook of a large size is required. The computational speed is also crucial because the Codebook search takes many operations.

2.3.2 Mixed Excitation Linear Prediction (MELP) Vocoder

The Mixed Excitation Linear Prediction (MELP) vocoder was first proposed by McCree et al in 1991 [9]. It can encode speech at 2.4 kbps while it maintains the synthesized speech quality equivalent to that of the 4.8-kbps DoD CELP vocoder [10]. Instead of using simple periodic pulse excitation, the MELP vocoder uses a combination of periodic pulses and white noise as an excitation signal to enhance the quality of synthesized speech. The MELP vocoder was selected for U. S. Federal Standard at 2.4 Kbps by DoD to replace FS-1015 (LPC-10) [22].

In some cases of voiced sounds, such as whispered sounds, the vocal tract is excited by a mixture of glottal pulse and noise. The MELP vocoder is designed to reproduce this characteristic. In the primary stage of MELP vocoder systems, input speech was classified as voiced, jittery voiced, or unvoiced. If either marginal periodicity or peakiness in the input speech is measured, the sound would be classified as jittery voiced.

To produce the excitation signals for voiced sounds, the MELP vocoder controls the mixture ratio of low-pass filtered pulse and high-pass filtered noise, depending on the relative power of the input speech signal. For the voiced sound, the ratio is 80% of pulse and 20% of noise. Jittered, aperiodic pulses are used for jittery voiced sounds. The mixture ratio for jittery voiced sounds is 50% of pulse and 50% of noise [9].

In the U. S. Federal Standard 2.4-kbps MELP vocoder, the input speech is filtered into five frequency bands for accurate analysis and 10^{th} order linear prediction is used. Recently, the reduced bit rate MELP vocoders [13] [27] and versions of MELP having improved sound quality [1] [21] have been proposed.

2.4 Outline of Residual-Excited Linear Predictive (RELP) Vocoder

The Residual-Excited Linear Predictive (RELP) vocoder, which is the focus of this research, was proposed by Un et al. in 1974 [8]. The RELP vocoder uses LPC analysis for vocal tract modeling like the two other vocoder systems discussed in previous sections. Linear prediction error (residual) signals are used for the excitation. There is no voiced/unvoiced detection or pitch detection required. The RELP vocoder, which Un et al. proposed, encodes speech between 6 and 9.6 kbps [26], depending on the quality of the synthesized speech desired.

Using the residual signals as the excitation improves the quality of the synthesized speech and makes it more natural than the basic LPC vocoders, because there are no misclassification of voiced/unvoiced sounds or miscalculation of pitches. The excitation signals of the RELP vocoder are very close to the ones the vocal tract produces. In contrast, the excitation signals (periodic pulses) of the basic LPC vocoder are completely artificial. However, the total encoding rate of the RELP vocoder is larger than most of the other LPC-based vocoder systems. The RELP vocoder needs to encode sequences of residual signals per segment, which is a large volume of data, while several bits are needed to encode the voiced/unvoiced decision, pitch, and gain for the other LPC systems. The RELP system is discussed in detail in Section 3.6.

2.5 Vowel Characterization

When the RELP vocoder system is considered as a tool of content-based speech response applications, it is a question of whether or not the parameterized information available from the RELP vocoder alone can identify speech contents or determine

what word was spoken.

Figure 2.6 illustrates examples of speech signals and their signal spectra. The speech signals are the short time segments of English vowels uttered by the same speaker. A speech signal at 8-kHz sampling rate is segmented in blocks of 20 ms to apply the LPC algorithm to fit an 8^{th} order FIR filter. The top graph of each column shows a segment of a vowel and the middle graph is its signal spectrum. The frequency response of the LPC filter whose parameters were obtained from this particular segment is shown in the bottom graph. The frequency response of the LPC filter is a low order approximation of the signal spectrum.

When a large number of frequency responses of the LPC filter for the same vowel sound spoken by the same speaker are observed, distinguishable characteristics can be recognized. This suggests that the frequency response of the LPC filter may be a more useful measure to classify vowel sounds than extracted parameters a_1 to a_P.

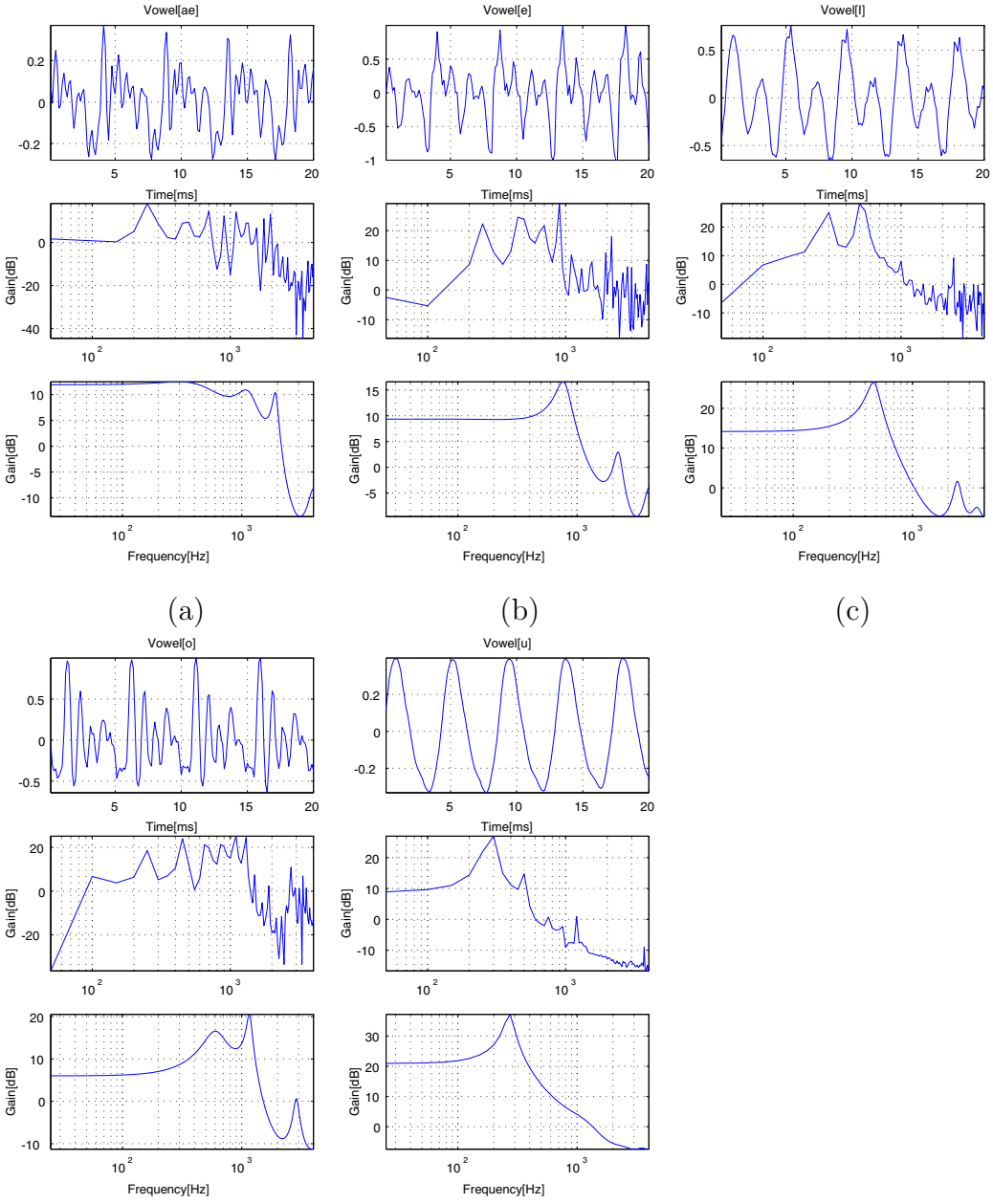


Figure 2.6 Examples of speech waveform (top), signal spectrum (middle) and frequency response of LPC filter (bottom) - vowel [æ] (a), vowel [e] (b), vowel [ɪ] (c), vowel [o] (d), vowel [u] (e)

3. LINEAR PREDICTIVE CODING

Linear predictive coding (LPC) for speech analysis and synthesis is based on linear prediction. In this chapter, linear prediction and related matters are discussed. In the last section, the RELP vocoder system is presented.

3.1 LINEAR PREDICTION

Linear mean-square estimation develops optimal filtering such as linear prediction. Linear mean-square estimation is derived from the basic principle of orthogonality which provides the foundation of minimum mean-square error. The orthogonality principle is described in Appendix A.1.

Linear prediction estimates the current value $x[n]$ of a random sequence x from P previous values of x to reduce redundant information from the sequence. The estimate $\hat{x}[n]$ can be written as

$$\hat{x}[n] = -a_1 x[n-1] - a_2 x[n-2] - \cdots - a_P x[n-P] \tag{3.1}$$

and the error in the estimate is given by

$$\varepsilon[n] = x[n] + a_1 x[n-1] + a_2 x[n-2] + \cdots + a_P x[n-P] \tag{3.2}$$

If one defines

$$a_0 \equiv 1 \tag{3.3}$$

then the error in Equation 3.2 can be expressed as

$$\varepsilon[n] = \sum_{k=0}^{P} a_k x[n-k] \tag{3.4}$$

This is the output of a finite impulse response (FIR) filter with impulse response $h[k] = a_k,\ k = 0, 1, 2, \ldots, P$. A problem of linear prediction is to obtain these

coefficients a_k when they minimize $\varepsilon[n]$ or the mean-square error

$$\begin{aligned} \sigma_\varepsilon^2 &= E\{|\varepsilon[n]|^2\} \\ &= E\{|x[n] - \hat{x}[n]|^2\} \end{aligned} \tag{3.5}$$

σ_ε^2 is called the prediction error variance in linear prediction.

To obtain the coefficients, the following Normal equations must be solved

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[P] \\ R_x[-1] & R_x[0] & \cdots & R_x[P-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[-P] & R_x[-(P-1)] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_P \end{bmatrix} = \begin{bmatrix} \sigma_\varepsilon^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.6}$$

The derivation of the Normal equations is in Appendix A.2.

3.2 Autoregressive (AR) Model

The linear prediction problem relates to the fact that when a system outputs white noise for a given input, the inverse system produces the output from white noise.

The linear prediction error filter is an FIR filter, as described in the previous section, and the transfer function is

$$A(z) = 1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_P z^{-P}. \tag{3.7}$$

If the order of the filter is large enough, it is known that the output becomes nearly white noise with variance σ_ε^2.

Now consider a filter which is inverted and excited by white noise whose variance $\sigma_w^2 = \sigma_\varepsilon^2$. The output of this filter will be a reproduction of the original sequence $x[n]$. This system can be written as

$$x[n] = -\mathrm{a}_1 x[n-1] - \mathrm{a}_2 x[n-2] - \cdots - \mathrm{a}_P x[n-P] + w[n] \tag{3.8}$$

This is equivalent to Equation 3.2 in recursive form. A linear combination of the variables $x[n-1]$ to $x[n-P]$, which are "independent", represents the variable $x[n]$,

which is "dependent." Thus $x[n]$ is called an autoregressive or AR process because the process is considered to be "regressed upon itself." Figure 3.1 shows the diagrams of the prediction error filter and the AR model.

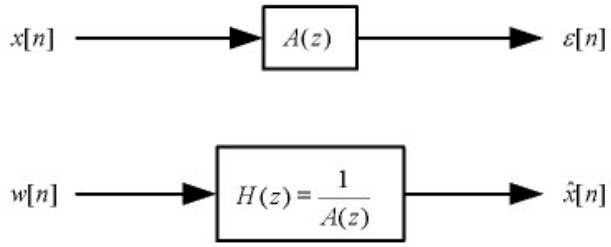


Figure 3.1 Prediction error filter and AR model

As in this figure, the AR model is an Infinite Impulse Response (IIR) filter whose transfer function is given by

$$H(z) = \frac{1}{A(z)}, \tag{3.9}$$

where

$$A(z) = 1 + \mathrm{a}_1 z^{-1} + \mathrm{a}_2 z^{-2} + \cdots + \mathrm{a}_P z^{-P}. \tag{3.10}$$

To obtain the parameters $\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_P$ and σ_w^2 of the AR model, solving Normal equations is done. The equations can be expressed in matrix form as

$$\begin{bmatrix} R_x[0] & R_x[-1] & \cdots & R_x[-P] \\ R_x[1] & R_x[0] & \cdots & R_x[-(P-1)] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[P] & R_x[P-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ \mathrm{a}_1 \\ \vdots \\ \mathrm{a}_P \end{bmatrix} = \begin{bmatrix} \sigma_w^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.11}$$

The derivation of this matrix is shown in Appendix A.3. These equations for the AR model are referred to as the Yule-Walker equations. They are identical in form to Equations 3.6 noting that $R_x[l] = R_x[-l]$. The differences between Equations 3.6 and the Yule-Walker equations are their parameters and the reversed correlation matrices.

3.3 Levinson Recursion

The Levinson recursion is a method to solve the Normal equations faster than the matrix inversion method. It starts with a filter order 0 and recursively generates

filters of order $1, 2, 3, \cdots$, up to the order P. The detailed mathematical derivations of the Levinson recursion and the associated topics, backward linear prediction and the anticausal AR model are described in Appendices A.4 and A.5.

The essential equations to compute the Levinson recursion process can be written as

$$\gamma_p = \frac{\mathbf{r}_{p-1}^T \tilde{\boldsymbol{a}}_{p-1}}{\sigma_{\varepsilon_{p-1}}^2} \tag{3.12}$$

$$\boldsymbol{a}_p = \begin{bmatrix} \boldsymbol{a}_{p-1} \\ \\ -- \\ 0 \end{bmatrix} - \gamma_p \begin{bmatrix} 0 \\ -- \\ \\ \tilde{\boldsymbol{a}}_{p-1} \end{bmatrix} \tag{3.13}$$

$$\sigma_{\varepsilon_p}^2 = (1 - |\gamma_p|^2)\sigma_{\varepsilon_{p-1}}^2 \tag{3.14}$$

where $p = 1, 2, \ldots, P$ and initial conditions are

$$\boldsymbol{a}_0 = [1]; \quad \mathbf{r}_0 = R_x[1]; \quad \sigma_{\varepsilon_0}^2 = R_x[0]. \tag{3.15}$$

The parameter γ_p is called the forward reflection coefficient. It is also known as partial correlation coefficient, or PARCOR coefficient. From Equations 3.13 and 3.15, the following facts can be derived:

$$a_p^{(p)} = -\gamma_p \tag{3.16}$$

and

$$0 \leq |\gamma_p| < 1. \tag{3.17}$$

Equation 3.16 indicates that a negative value of p^{th} PARCOR coefficient is always equal to the p^{th} linear prediction coefficient.

3.4 Lattice Filter Structure

The lattice filter is a very useful form of a filter representation in digital speech processing. It requires only the reflection coefficients, γ_p, $p = 1, 2, \ldots, P$, to form the filter. Since the reflection coefficients are always between -1 and 1 from Equation 3.17, the filter has guaranteed stability in the IIR form. For the same reason,

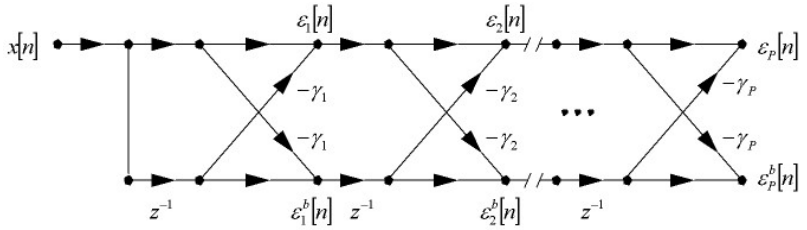


Figure 3.2 The lattice realization of an FIR filter

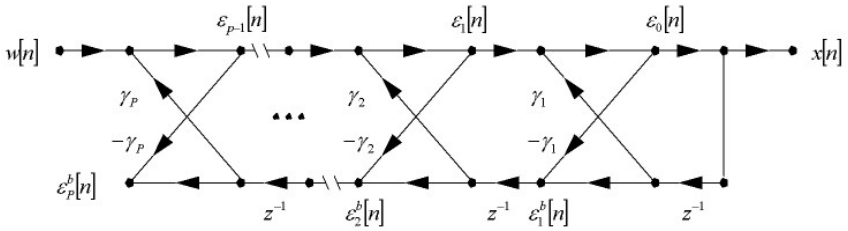


Figure 3.3 The lattice realization of an IIR filter

the reflection coefficients are able to be encoded into smaller data than the prediction coefficients. For example, if the prediction coefficients were to be coded, they would require between 8 to 10 bits per coefficient, but the reflection coefficients require 6 bits per coefficient at the same accuracy requirements [6]. The lattice representation of the p^{th} order filter is given by

$$\begin{bmatrix} \varepsilon_p[n] \\ \varepsilon_p^b[n] \end{bmatrix} = \begin{bmatrix} 1 & -\gamma_p \\ -\gamma_p & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{p-1}[n] \\ \varepsilon_{p-1}^b[n-1] \end{bmatrix} \tag{3.18}$$

where the backward prediction error $\varepsilon_p^b[n]$ (Appendix A.4) is

$$\varepsilon_p^b[n] = \varepsilon_p^{'}[n-p] \tag{3.19}$$

Equation 3.18 means that if the forward and backward prediction errors of the $(p-1)^{th}$ order filter and the reflection coefficients are known, the prediction errors of the p^{th} order filter can be easily obtained. Figure 3.2 and 3.3 are the lattice realization of FIR filter and IIR filter, respectively.

3.5 AR Spectrum Estimation

The power spectral density function of a model is given by

$$\sigma_w^2 |H(e^{j\omega})|^2$$

where $H(e^{j\omega})$ is the spectrum domain representation of a random process $H(z)$. σ_w^2 is the variance of white noise. An estimate of the spectrum of the random process is given by this function. σ_w^2 is assumed to be equal to one for the AR model. When the AR model is used to generate a spectral estimate, the corresponding estimate for the power density spectrum of the random process $\hat{S}_{AR}(e^{j\omega})$ has the form

$$\hat{S}_{AR}(e^{j\omega}) = \frac{\sigma_P^2}{|A(e^{j\omega})|^2} \tag{3.20}$$

where $A(z)$ is given as Equation 3.10 and σ_P^2 is a P^{th} order error variance. This shows that the frequency response obtained from $z = e^{j\omega}$ is equivalent to what is obtained directly from the spectrum estimation.

3.6 RELP Vocoder System

The Residual Excited Linear Prediction (RELP) vocoder is one of the Linear Predictive Coding (LPC)-based vocoders. Its compression rate is moderate because the RELP vocoder needs to encode a sequence of residual signals for exciting the vocal tract model synthesized from speech signals. However, the quality of synthesized speech is superior to other kinds of LPC vocoders. The system is robust since there is no need to analyze whether the sound is voiced or unvoiced nor to analyze the pitch period. Un et al. [26] state that the RELP vocoder consists of five functional blocks: an LPC analyzer, a residual encoder, a residual decoder, a spectral flattener, and an LPC synthesizer. Figure 3.4 is a block diagram of the RELP vocoder.

The key concepts of the RELP vocoder are as follows. In LPC analysis, the LPC analyzer computes the prediction coefficients, and then produces the residual signals. A speech signal is low-pass filtered with cut-off frequency of 3.2 kHz and is sampled at the rate of 6.8 kHz by the analog-to-digital converter before LPC analysis. The speech samples are windowed by the Hamming window. The length of the Hamming window is 196 sample points. 136 sample points are specified as a 20-ms analysis block. 30 samples of the previous block and 30 samples of the next block are added to the analysis block for the overlaps. These overlaps ensure smooth transitions to avoid abrupt changes between analysis blocks.

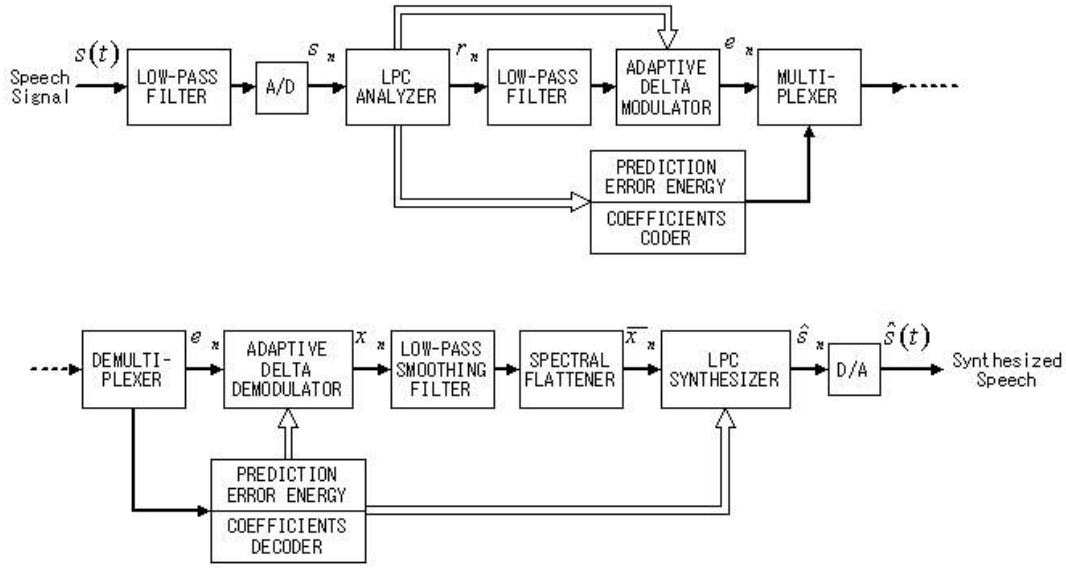


Figure 3.4 Block diagram of RELP vocoder [26] pp.1467

The Adaptive Delta Modulation (ADM) encoder is used for the residual coding. The residual signals are low-pass filtered at 800 Hz to reduce the transmission rate before coding. The sign bit e_n of the ADM is generated from the filtered residual signal r_n as

$$e_n = \begin{cases} +1 & \text{if } r_n \geq x_n \\ -1 & \text{if } r_n < x_n \end{cases} \tag{3.21}$$

with

$$\begin{aligned} x_n &= 0.99x_{n-1} + e_{n-1}\Delta_{n-1} \\ \Delta_n &= \left[\alpha\ E^{(j)}\right]\gamma_n \\ \gamma_n &= \beta_n\gamma_{n-1} \\ \beta_n &= f(e_n, e_{n-1}, e_{n-2}, e_{n-3}, e_{n-4}) \end{aligned}$$

where $0 < \gamma_{min} \leq \gamma_i \leq \gamma_{max}$ for all i, Δ_n is the n^{th} step size, and β_n is a multiplication factor. The initial step size Δ_0 is obtained by

$$\Delta_0 = \alpha E^{(j)}$$

where $\gamma_0 = 1$, α is a scale factor, and $E^{(j)}$ is the average residual energy in the j^{th} analysis segment. The multiplication factor β_n is decided by the sequence of the

Table 3.1 ADM Logic Rule [26] pp.1470

e_n	e_{n-1}	e_{n-2}	e_{n-3}	e_{n-4}	β_n
+	+	+			1.5
-	-	-			1.5
-	-	+			1
+	+	-			1
-	+	+			0.66
+	-	-			0.66
-	+	-			0.66
+	-	+			0.66
-	+	+	+	+	1
+	-	-	-	-	1

present sign bit and the previous four sign bits. Table 3.1 shows that the ADM logic rule to determine β_n.

At the residual decoding process, the reproduced residual signals are generated from the transmitted sign bits through the ADM decoder, which has the opposite structure of the ADM encoder. The output signals of the ADM decoder are low-pass filtered for the purpose of smoothing them. Because the residual signals are low-pass filtered before the ADM encoder, the spectral flattener is used to recover the high-frequency harmonics of the residual signals. The synthesized speech then is generated by the LPC synthesizer, with the recovered residual as the excitation signal.

With this RELP vocoder system, Un et al. [26] state that the total transmission rate of 9600 bps is comprised of residual: 6800 bps; coefficients: 2250 bps; gain: 200 bps; normalized energy: 200 bps; frame synchronization: 150 bps at 6.8 kHz sampling rate with 10^{th} order LPC filter. At this rate, the synthesized speech is intelligible and the speaker can be easily identified.

4. IMPLEMENTATION OF RELP VOCODER

Some of the required elements of the digital signal processor (DSP) development system, Texas Instruments' TMS 320C6711 DSP starter kit (DSK), are discussed in this chapter. The first section focuses on the hardware aspect of the DSK; the second section explains its software. The DSK is used to implement the RELP vocoder. The detailed implementation of the RELP vocoder using the DSK is explained in the third section of this chapter.

4.1 TMS320C6711 DSK

TMS320C6711 DSP starter kit (DSK) is Texas Instruments' (TI) DSP development tool. It includes the hardware (DSK board) and the software (Code Composer Studio) for real-time signal processing.

4.1.1 DSP Starter Kit (DSK) Board

Figure 4.1 is a block diagram of the DSK board. The DSK board includes a floating-point digital signal processor, TMS320C6711, which will be discussed in the next subsection. The board provides 150 MHz clock speed for the CPU which enables 900 million floating-point operations per second (MFLOPS) execution. The two voltage regulators provide 1.8 V for the C6711 core and 3.3 V for its memory and peripherals. There are two 4-MB synchronous dynamic random-access memory (SDRAM) and one 128-kB flash read-only memory (ROM) on the board. The connection between the board and a host Personal Computer (PC) for developing programs is made by using a standard parallel port.

The board also includes a 16-bit codec TLC320AD535 for analog input and out-

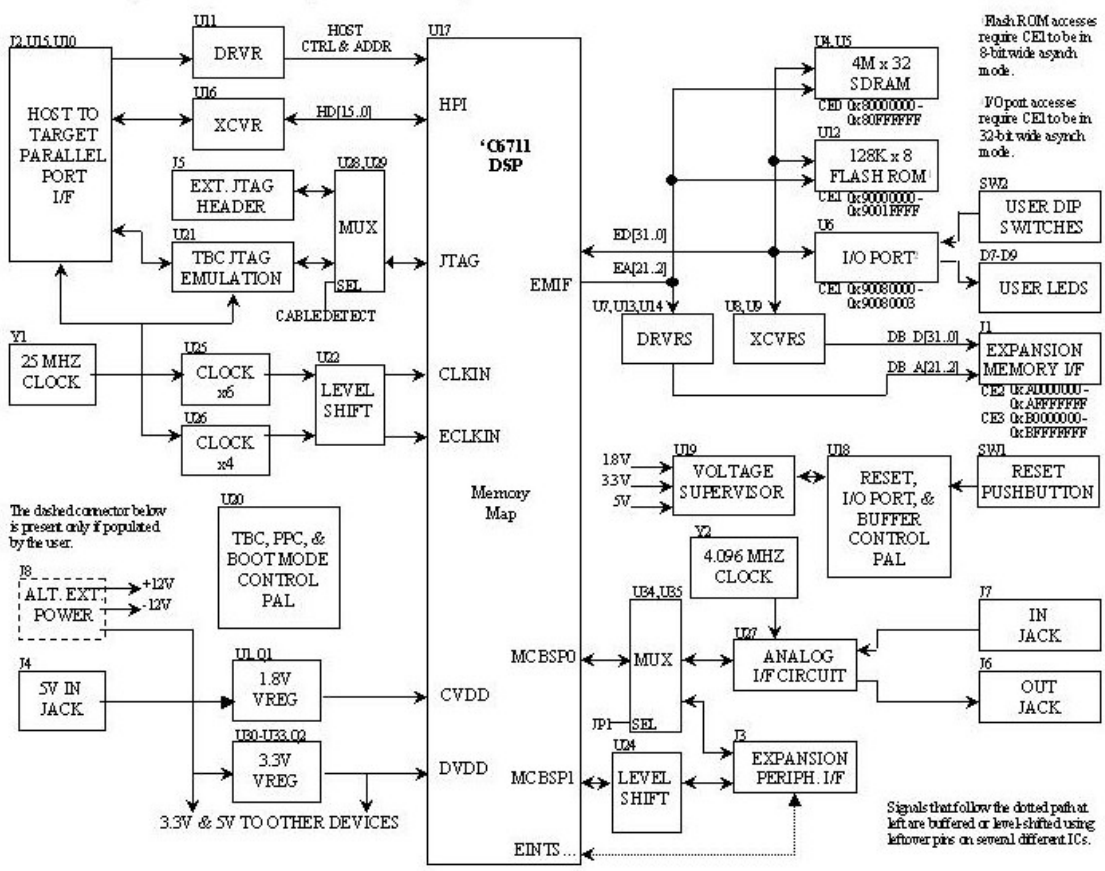


Figure 4.1 Block diagram of TMS320C6711 DSK [34]

put. The AD535 is a dual-channel voice/data codec which provides analog-to-digital conversion (ADC), digital-to-analog conversion (DAC), and all required filtering functions [29] in one chip. The onboard codec has one input and one output accessible through two 3.5 mm audio cable connectors, J7 and J6. Although the AD535 is capable of operating at different sampling rates up to 11.025 kHz, the sampling rate of the onboard AD535 is fixed at 8 kHz. The codec master clock MCLK is 4.096 MHz and this frequency sets the sampling rate F_s,

$$F_s = \text{MCLK}/512 = 8 \text{ kHz}$$

Therefore, the sampling rate of the RELP vocoder using the DSK in this research is set at 8 kHz. There is a low-pass filter before the ADC, in fact, the 3-dB passband is up to 3.6 kHz for this sampling rate.

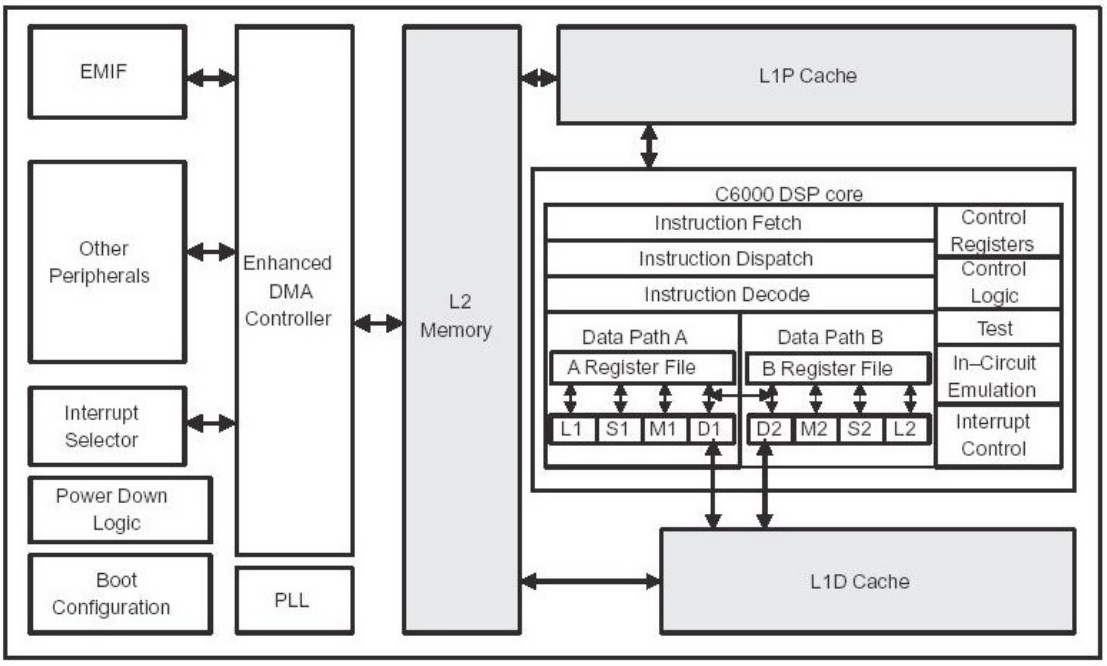


Figure 4.2 Block diagram of TMS320C671x [32] pp.3-2

4.1.2 TMS320C6711 Processor Architecture

The TMS320C6711 belongs to TI's C6x floating-point processor family. It is based on advanced very-long-instruction-word (VLIW) architecture and is able to execute up to eight instructions per cycle [32]. The processor is designed so that the efficiency of a C compiler is taken into consideration on the assumption that C is the language used for the DSP programming. Therefore, it is possible that a program written in C can run 80 - 90% of processing speed compared to the one written in assembly code [15]. Figure 4.2 shows the functional block diagram of TMS320C671x. Table 4.1 illustrates the main features of the processor.

The C6711 has two 4-kB level 1 internal cache memories: the program cache (L1P) and the data cache (L1D); and one level 2 internal 64-kB RAM (L2) for data/program allocation. There are eight functional units, which are six arithmetic logic units (ALU) and two multiplier units.

The following are peripherals of the C6711: two multichannel buffered serial ports (McBSPs), which can handle up to 128 channels and provide a direct interface to

Table 4.1 Main Features of TMS320C6711 [32] [15]

Instruction cycle time	6.67 ns (Clock speed: 150 MHz)	
Max. operations	900 MFLOPS, with 6 functional units	
Max. instructions	1200 MIPS, 8 instructions/cycle	
Data support	8/16/32-bit, fixed-point operations 32/64-bit, floating-point operations	
Address space	4 GB	
Internal memory	Level 1 program cache Level 1 data cache Level 2 cache/RAM	4 kB 4 kB 64 kB
General-purpose registers	2 sets of 32-bit x 16	
Functional units	ALU Multiplier unit	6 2
Power Voltage	3.3 V (I/O), 1.8 V (internal)	

industry-standard external peripherals; two 32-bit general-purpose timers which can be used to time and count events, to interrupt the CPU, or to send synchronization events to the direct memory access (DMA); 16-bit host port interface (HPI), which is a parallel port to communicate with a host PC; and 32-bit external memory interface (EMIF), which supports a glueless interface to a variety of external devices. Internal buses of the C6711 are as follows: 32-bit program address, 256-bit program data, two 32-bit data addresses, two 64-bit data addresses, two 64-bit data, and two 64-bit store data.

The eight functional units are divided into two data paths, A and B, as shown in Figure 4.2. Unit (.L) is for logical and arithmetic operations. Unit (.S) is for branch, bit manipulation, and arithmetic operations. Unit (.M) is for logical and arithmetic operations. Unit (.D) is for loading/storing and arithmetic operations. These four units are in both data paths A and B. (.L) and (.S) units are floating/fixed-point ALUs and (.D) units are fixed-point ALUs. (.M) units are floating/fixed-point multipliers.

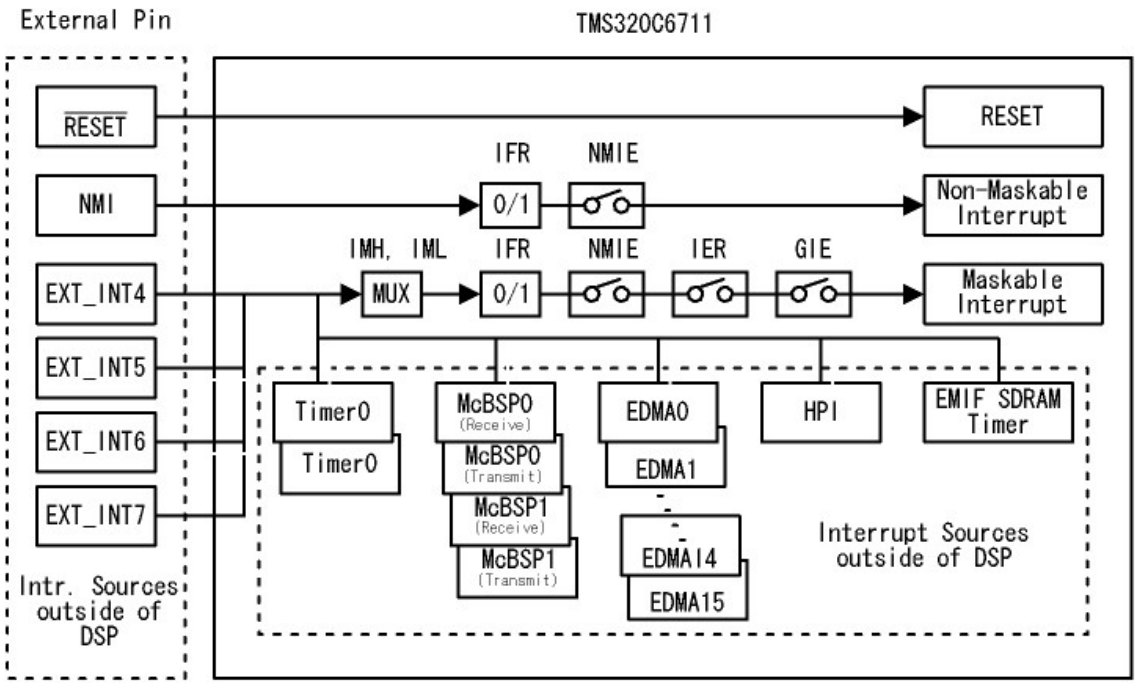


Figure 4.3 Hardware structure of TMS320C6711 for interrupts [15] pp.274

4.1.3 Interrupts

An interrupt occurs when multiple external asynchronous events require the DSP to process tasks. An interrupt stops the CPU's current process and lets the CPU execute the task initiated by the interrupt. There can be internal or external sources of the interrupt, such as an ADC, a timer, or other peripherals. When an interrupt occurs, the states of the current process in the CPU are saved in registers, enabling recovery of them after the tasks which the interrupt requests are completed. The hardware structure of the TMS320C6711 for interrupts is shown in Figure 4.3. There are three types of interrupts: reset, non-maskable interrupt (NMI), and maskable interrupt. Reset has the highest priority followed by NMI and maskable interrupts.

The TMS320C6711 has 13 maskable interrupts: two timer interrupts; four external interrupts; four McBSP interrupts; one enhanced DMA (EDMA) which can be selected from 16 of the EDMA; a host processor to DSP interrupt; and EMIF SDRAM timer interrupt [31] [32]. In this RELP vocoder project, an analog voice signal is pro-

cessed in real time. For an interface between the DSP and TLC320AD535, McBSP0 is used [15]. Therefore, the McBSP0 interrupt is used in this vocoder implementation.

In Figure 4.3, IMH and IML are the interrupt multiplexer high and low registers. They determine the mapping between the interrupt sources. IFR is the interrupt flag register which indicates the status of interrupts. When an interrupt occurs, the IFR is set to one. IER is the interrupt-enable register which enables or disables individual interrupts. NMIE is the nonmaskable interrupt enable bit. It is bit one of the IER and when this bit is zero, reset is the only allowed interrupt. GIE is the global interrupt enable bit, the bit zero of the control status register (CSR), and when it is set to one, the maskable interrupts are allowed.

4.2 Program Development

The necessary issues of program development using the TMS320C6711 DSK are discussed in this section. The TMS320C6711 DSK includes code generation tools, Code Composer Studio (CCS). CCS helps to develop programs with the TMS320C6000 processors whose development processes are complicated and difficult.

4.2.1 Overview of Code Composer Studio (CCS)

Code Composer Studio (CCS) is a TI software support tool which provides an integrated development environment (IDE). The interface is similar to that of the Microsoft Visual C++. CCS includes a C/C++ compiler, an assembler, a linker, a debugger, and other utilities. It also supports real-time processing such as analysis, scheduling, and data exchange with DSP/BIOS.

The C/C++ compiler of CCS builds the American National Standards Institute (ANSI) C programs into C6000 assembly language source code using a sophisticated optimization pass. It can generate efficient and compact code. The assembler creates a machine-language object file from an assembly source code. The linker produces an executable file by combining object files and object libraries. The produced executable file is ready to be loaded to the DSK and run on the TMS320C6711 processor. The

debugger has an ability to show data graphically so that a user can find and fix errors quickly and efficiently. DSP/BIOS is a scalable real-time kernel which is designed for real-time applications such as scheduling, synchronization and host-to-target communication. It allows analysis of an application program in real time without stopping the digital signal processor.

CCS keeps all information of an application project in a project file (with extension .pjt). It stores file names of source code and object libraries, code generation tool options, and include file dependencies.

When C is used to develop a project, the following files, at least, need to be prepared by a programmer: C source code file(s) (with extension .c); a vector source file written in assembly code, which describes reset or interrupt (with extension .asm); and a linker command file (with extension .cmd).

4.2.2 CCS and Programming Codes

Developing programs with CCS, available programming codes are C/C++, linear assembly code, and conventional assembly code. The C/C++ compiler of CCS is capable of handling burdensome tasks such as instruction selection, parallelizing, pipelining, and register allocation. It optimizes the C/C++ source code into efficient assembly code so that a programmer does not need to do so by hand [33].

An assembler optimizer is available in CCS, if a more efficient code than the one produced by the C/C++ compiler is necessary. Linear assembly code (with extension .sa) is used to write for the assembly optimizer [30]. It is an assembly code that has not been register-allocated or scheduled. A programmer does not need to include information about instruction latencies or register usage. The assembler optimizer takes care of those as well as does the C/C++ compiler. Generally, the object code produced by the assembler optimizer is more efficient than the code produced by the C/C++ compiler.

Conventional assembly code is also available when a more efficient code is needed.

In this research, the language C is used to develop the programs.

4.2.3 C for TMS320C6000

The variant of C used for the TMS320C6000 is in conformity with the ANSI C. The sizes of data type vary. Table 4.2 shows some of the data types and the size used for TMS320C6000. The C/C++ compiler of CCS adds some keywords to extend C/C++ language. However, only the `interrupt` keyword is introduced here because this is the only keyword that is used in the project program. When `interrupt` is declared in a function, the function is regarded as an interrupt routine by the compiler. The constants of all registers which are used before the interrupt occurs are saved. The special return sequence for interrupts is generated. The C/C++ compiler generates these tasks.

In C, the `#pragma` directive is used so that a programmer can place compiler instructions in the source code [17]. The C/C++ compiler supports a set of `#pragma`s. A `#pragma` called `DATA_SECTION` is used in the project program to allocate data in external memory. The syntax is as follows:

```
#pragma DATA_SECTION (symbol, "section name");
```

It allocates space for `symbol` in a section named `section name` [30].

4.2.4 Support Programs/Files

The minimal required files which a programmer must prepare when a project is built with CCS were mentioned in Subsection 4.2.1. There are several other support files used to build a project. Note that a linker command file is specific to this project program.

`C6xdsk.h` and `C6xinterrupts.h` are TI support files included with the DSK. `C6xdsk.h` is a DSK header file which defines addresses of the external memory interface, the serial ports, the timers, interrupt, and so on. `C6xinterrupts.h` initializes interrupt functions.

Table 4.2 TMS320C6000 C/C++ Data Types (excerpt) [30]

TYPE	SIZE	REPRESENTATION
char, unsigned char	8 bits	ASCII
short	16 bits	2s complement
unsigned short	16 bits	Binary
int, signed int	32 bits	2s complement
unsigned int	32 bits	2s Binary
long, signed long	40 bits	2s complement
unsigned long	40 bits	Binary
float	32 bits	IEEE 32-bit
double	64 bits	IEEE 64-bit

The following three support files are provided by Chassaing [3]. These files are so convenient for designing a DSP program that this project program includes them as support files.

1. `C6xdskinit.c` is an initialization/communication file which initializes the DSK, the AD535, and McBSP. It configures the transmit interrupt INT11 and enables it.

2. `C6xdskinit.h` is the header file.

3. A vector file, `vectors_11.asm` is a modified version of `vectors.asm` included with CCS. This vector file can handle interrupts and selects INT11.

Finally, a linker command file, `proto.cmd` is listed below. This is modified from a generic linker command file. The only difference is an addition of a memory space, `buffer_ext`. This is the space for `lpc_data` section which stores LPC coefficients, PARCOR parameters, gain, residual data, input signal and output signal up to 10 seconds for 10^{th} order prediction at 8 kHz sampling frequency. It enables one to

analyze these data after processing voice signals in real-time to store them in external memory.

```
/*proto.cmd -- Linker command file for LPC project */

MEMORY
{
  VECS:       org =         0h, len =      0x220
  IRAM:       org = 0x00000220, len = 0x0000FDC0 /*internal memory*/
  buffer_ext  org = 0x80000000, len = 0x001F0000 /*external memory*/
  SDRAM:      org = 0x801F0000, len = 0x01000000 /*external memory*/
  FLASH:      org = 0x90000000, len = 0x00020000 /*flash memory*/
}

SECTIONS
{
  lpc_data :> buffer_ext
  vectors  :> VECS
  .text    :> IRAM
  .bss     :> IRAM
  .cinit   :> IRAM
  .stack   :> IRAM
  .sysmem  :> SDRAM
  .const   :> IRAM
  .switch  :> IRAM
  .far     :> SDRAM
  .cio     :> SDRAM
}
```

Figure 4.4 Linker command file (`proto.cmd`)

4.3 Implementation of RELP Vocoder

The implementation of the RELP vocoder with TMS320C6711 DSK is discussed in this section. The entire main program of the RELP vocoder (`protoxx.c`) is in Appendix B.1. The program performs as both an encoder and a decoder and stores most of the data, such as the PARCOR parameters, the LPC coefficients, the residues, the input signal, and the synthesized signal, in external memory. The data can then be transferred to a host PC and analyzed, if necessary, after voice signals are processed.

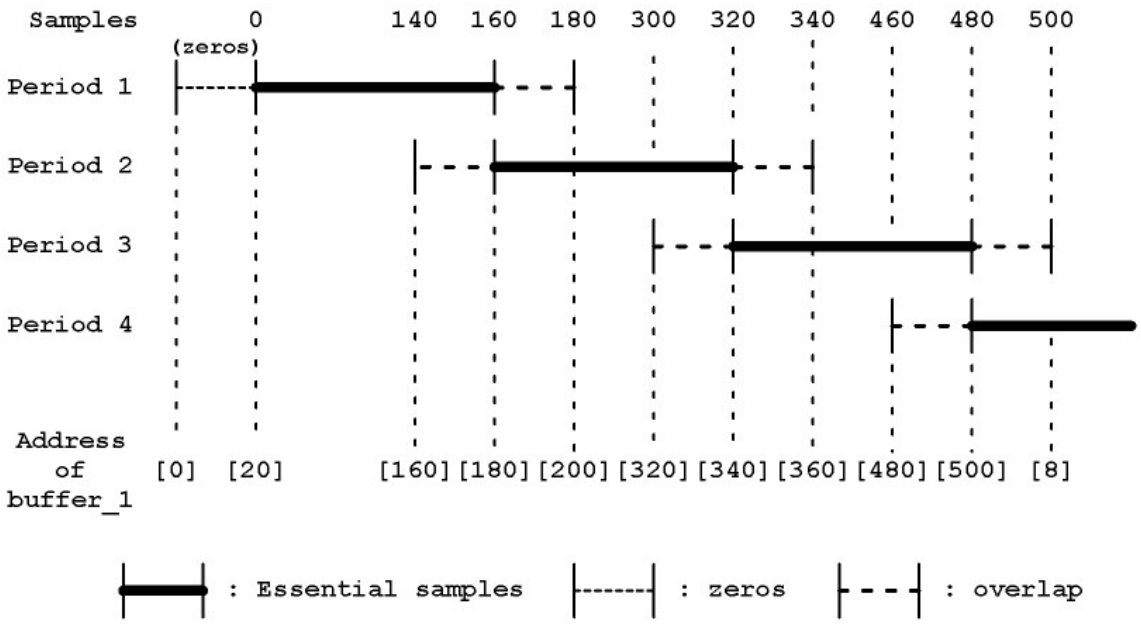


Figure 4.5 The relationship between incoming samples and ring buffer `buffer_1`

The interrupt service routine (ISR) handles the input signal and the output synthesized signal. The main routine calls the ISR at every sampling period. The input signal is sampled at 8,000 Hz by the codec AD535 and the RELP vocoder treats a 20 millisecond time segment as one block. Therefore, every 160 samples of the input signal are sampled and stored, then the block is processed by the vocoder.

However, the estimation of LPC at the beginning of a segment is not accurate because of the lack of previous samples. This may cause an abrupt transition of synthesized waveforms between segments. Overlaps are necessary to avoid abrupt transitions and accomplish high quality synthesized speech. 20 samples (2.5 ms) of the end of the previous segment and of the beginning of the next segment are added to the analysis block in the RELP vocoder as overlaps. The overlaps are discarded at the decoder after a speech segment is synthesized.

At first, the samples of the input signal are saved in `buffer_1`. This is a ring buffer which is able to store 512 samples. The pointer `Ptr_dly` points to the address of the latest sample in the `buffer_1`. Every time 160 samples are stored in `buffer_1`,

the 200 samples, including the overlaps, are transferred to `buffer_2` to be processed by the function `trans` (Appendix B.2).

```
/* ======================= interrupt service routine ========================= */
interrupt void c_int11()
{
 short sample_data;

 sample_data = input_sample();              /* new input data                  */
 buffer_1[*Ptr_dly] = (float)sample_data;   /* store input data into buffer    */
 i++;                                       /* increment counter               */
/* ======================== After the 180th sample ========================== */
 else if (status == 1 && i == 160)       /* process every 160 samples (20 ms)  */
 {
    trans(Ptr_dly, buffer_1, buffer_2);  /* transfer data into buffer_2        */
    flag = 1;                            /* call the functions                 */
    i = 0;                               /* reset i to 0                       */
 }
/* increment pointer Ptr_dly to the last address of ring buffer                */
 *Ptr_dly = (*Ptr_dly + 1) % BUFFER_1_SIZE;
 return;                                 /* return from ISR                    */
}
/* ================= the end of interrupt service routine ==================== */
```

Since the first block does not have an overlap of a previous segment, the ISR treats the first block differently. The initial value of `Ptr_dly` is set for 20, so that the values of the first 20 samples of the first block are all zeros as an overlap. Moreover, the ISR counts until the 180^{th} sample comes in for the 20 samples of the overlap of the next segment instead of 160. Figure 4.5 illustrates how the ring buffer, `buffer_1`, incoming samples, and each processing period, are related. "Samples" at the top of the figure indicates the incoming samples. "Period 1", $\cdots$, "Period 4" indicate the processing periods in the figure.

```
/* ========================= The first 180 samples =========================== */
 if (status == 0 && i == 180)            /* the very first processing period   */
 {
    trans(Ptr_dly, buffer_1, buffer_2);  /* transfer data into buffer_2        */
    flag = 1;                            /* call the functions                 */
    status = 1;                          /* change status to 1                 */
    i = 0;                               /* reset i to 0                       */
```

```
 }
/* ==================== The end of the first 180 samples ====================== */
```

The ISR executes to output the synthesized signal through the AD535 once in every sampling period when the data of the output signal is calculated by the RELP functions and becomes ready for output.

```
static int z = 0;

if (z < ((BUFFER_2_SIZE - 40) * 50 * REC_TIME))
    output_sample((int)out_buff[z++]);
```

After the samples are transferred to `buffer_2`, `flag` is set to 1 by the ISR, then the RELP processing functions in the main routine are executed. These functions are `LPC_encode`, `fwd_lattice`, `encode`, and `inv_lattice`. `LPC_encode` calculates the LPC coefficients, the PARCOR parameters, and the gain of the input signal. `fwd_lattice` produces the residual signal using the PARCOR parameters. The function `encode`, at the end of Appendix B.1, encodes all data and stores them in external memory space: the LPC coefficients; the PARCOR parameters; the gain; and the residual signal. The function `inv_lattice` obtains the data from external memory and reproduces the signal. The series of these functions must be executed within a sampling interval of 20 milliseconds by the processor. Then `flag` is set back to 0. Thus, the real-time operation is achieved by the described program structure that uses ISR. The following list is the main routine of the program.

```
/* ======================= main routine ======================================== */
 comm_intr();                              /* init DSK, codec, McBSP           */
 while(1)                                  /* infinite loop                    */
 {
    a = (float *)calloc(ORDER+1, sizeof(float));
    gamma = (float *)calloc(ORDER, sizeof(float));
    e = (float *)calloc(BUFFER_2_SIZE, sizeof(float));

    while(flag == 1)
    {
        LPC_encode(buffer_2, a, gamma, gn); /* obtain the coefficients         */
```

```
            fwd_lattice(buffer_2, gamma, e);    /* obtain residue signal           */
            encode(a, e, gamma, gn);            /* encode into files               */
            inv_lattice(e, gamma, x);           /* reproduce synthesized signal    */
            for (t = 0; t < 160; t++)           /* store the signal in out_buff[]  */
                out_buff[s + t] = x[t+20];
            s = s + 160;
            flag = 0;                           /* set flag to zero                */
            count++;
            if (count == (50 * REC_TIME)-1)
            {
                flag = 2;
                status = 2;
            }
        }
        free(a);
        free(gamma);
        free(e);
     }
    }
    /* ======================= the end of main routine ============================ */
```

The constants and declarations used in the programs are placed in the header file `proto.h` (Appendix B.7).

4.3.1 LPC Encoding

When `flag` turns to 1, the function `LPC_encode` (Appendix B.3) receives the data of a 25-ms voice signal, including overlaps in `buffer_2`. It calculates the LPC coefficients, the PARCOR parameters, and the gain, by using the Levinson recursion algorithm. First, it calculates an autocorrelation of the 25-ms signal array `s[]` using the autocorrelation function `a_corr` (Appendix B.4). This function is a modified version of TI's DSPLIB function `DSP_autocor`. It is modified to be able to handle the floating point operations.

```
    /* ================     Autocorralation of input signal     ================ */
        a_corr(Rx, s, BUFFER_2_SIZE, BUFFER_2_SIZE);
```

After the array of the autocorrelation `Rx[]` is calculated, the Levinson recursion algorithm is processed. In `Levinson Recursion`, all the necessary initial conditions

(Equations A.38, 3.15) are set at first.

```
/* ====   initial conditions & initialization   ==== */
    a[0] = 1;
    rev_a[0] = 0;
    power = Rx[0];
    for (m = 1; m <= ORDER; m++)
    {
        a[m] = rev_a[m] = 0;
        r[m-1] = Rx[m];
        gamma[m-1] = 0;
    }
```

where `a[]` is an array of the LPC coefficients, `rev_a[]` is a reversal matrix of `a[]`, `power` is $\sigma^2_{\varepsilon_0}$, `r[]` is $\mathbf{r}_p$, and `gamma[]` is an array of the PARCOR parameters γ_p. The first inner `for` loop calculates a numerator of Equation 3.12.

```
for (p=0; p<=n; p++)
    ra += (double)r[p]*(double)a[n-p];
```

that means;

$$\texttt{ra} = \mathbf{r}^T_{p-1}\tilde{\boldsymbol{a}}_{p-1} \tag{4.1}$$

In the next line, p^{th} PARCOR parameter, γ_p is obtained.

```
gamma[n] = (double)ra/(double)power;     /* "actual" gamma[n] is     */
                                         /* gamma[n+1]               */
```

The second inner `for` loop creates `rev_a[]`, which is a new reversal matrix of `a[]`, $\tilde{\boldsymbol{a}}_{p-1}$, and the third inner `for` loop obtains the array of the next order LPC coefficients as in Equation 3.13. Then the p^{th} order prediction error variance, $\sigma^2_{\varepsilon_p}$, is calculated in the next line (equation (3.14)).

```
for(p=0; p<=n; p++)                 /* make a reversal matrix of a[] */
    rev_a[p] = a[n-p];

for(p=1; p<=n+1; p++)
    a[p] = a[p]-((double)gamma[n]*(double)rev_a[p-1]);

power = (double)power*(1 - ((double)gamma[n]*(double)gamma[n]));
```

The `Levinson Recursion` is repeated until the desired order is reached.

4.3.2 Forward Lattice Filter

The forward lattice filter function `fwd_lattice` (Appendix B.5) produces a prediction error signal or a residual signal `e[]` from an input signal `s[]` through a prediction error FIR lattice filter, using the PARCOR parameters `gamma[]`. As shown in Figure 3.2, the lattice filter requires a delay line `dly[]`, which is a ring buffer, so that the data of the input signal are stored by one sample shifted behind it. Note that the initial first element of the array `dly[]` must be always zero.

```
for (m = 0; m < BUFFER_2_SIZE-1; m++)
{
    e[m] = s[BUFFER_2_SIZE + m];
    dly[*Ptr_lat + m + 1] = s[BUFFER_2_SIZE + m];
}
e[BUFFER_2_SIZE-1] = s[2*BUFFER_2_SIZE-1];
dly[*Ptr_lat] = 0;
```

The inner `for` loop shown below represents Equation 3.18. The inner `for` loop calculates a series of segments of a residual signal by passing through one lattice, then the outer `for` loop is repeated the number of times set by the filter order.

```
    for (n = 0; n <ORDER; n++)
    {
        for (m = 0; m < BUFFER_2_SIZE; m++)
        {
            dum = e[m];
            p = (*Ptr_lat + m) % BUFFER_2_SIZE;        /* for dly[]          */
            e[m] = dum - (double)gamma[n]*(double)dly[p];
            dly[p] = -1*(double)gamma[n]*(double)dum + dly[p];
        }
        *Ptr_lat = (*Ptr_lat + BUFFER_2_SIZE - 1) % BUFFER_2_SIZE;
        dly[*Ptr_lat] = 0;
    }
}
```

4.3.3 Inverse Lattice Filter

The inverse lattice IIR filter (Appendix B.6) performs as a decoder of the RELP vocoder. The filter reproduces a voice signal using the PARCOR parameters and a residual signal as an input. The structure of the filter is the inverse of the forward lattice filter shown in Figure 3.3. The lattice representation of the p^{th} order IIR filter is given by

$$\varepsilon_{p-1}[n] = \varepsilon_p[n] + \gamma_p \varepsilon^b_{p-1}[n-1] \tag{4.2}$$

$$\varepsilon^b_p[n] = -\gamma_p \varepsilon_{p-1}[n] + \varepsilon^b_{p-1}[n-1] \tag{4.3}$$

The C code of the filter part is shown below. Each sample of the residual signal passes through the filter p times, which is the order of the filter, to produce a sample of the synthesized signal in the inner `for` loop. The routine is repeated a number of times equal to the buffer length by the outer `for` loop. This is one of the differences compared with the forward lattice filter.

```
for(n = ORDER; n >= 1; n--)
{
    f[n-1] = f[n] + (double)gamma[n-1]*(double)g[n-1];
    g[n] = -1*(double)gamma[n-1]*(double)f[n-1] + g[n-1];
}
```

To test the quality of the synthesized voice signal, which depends on the compression rate of the residual signal, the following code is inserted in the inverse lattice filter. The code changes the data of the residual signal to fixed point format from floating point, then it discards lower bit(s) of the fixed point data by shifting toward right. The data is then shifted back left by the same bit(s) size. This operation changes the size of the residual signal data.

```
fx_res = (short)residue[m];  // fixed point version
fx_res = fx_res * R_sift_2;
fx_res = fx_res * L_sift_2;
f[ORDER] = (float)fx_res;

/*f[ORDER] = residue[m];*/
```

5. VOWEL CHARACTERIZATION

5.1 Observations of Vowel Sounds

Vowel classification was discussed briefly in Section 2.5. By observing a large sample of speech segments, frequency responses of the LPC (RELP) filter show distinguishable characteristics in each vowel sound produced by the same speaker. The work in this chapter has been presented at PACRIM '03 [23]. Figure 5.1 through Figure 5.10 show more examples of the vowel speech waveforms, the signal spectra, and the LPC frequency responses of consecutive 20-ms segments. The tested English vowels are [æ] as in 'at', [e] as in 'bed', [ɪ] as in 'Kim', [o] as in 'Tom', and [u] as in 'too'.

The speech signals of the graphs are consecutive in alphabetical order in each figure. Two different sets of utterances for each vowel are shown as examples to illustrate the similarity of frequency responses. The number of the segments differs in each utterance. This is because the duration of each utterance is slightly different from that of the others. For example, in the phoneme [ɪ], the duration time is much shorter than that of other phonemes. Even if the same speaker utters the same word or the same phoneme, the duration differs from time to time.

These vowels were uttered individually (not within a word) by the English speaking female speaker and recorded at 8 kHz sampling rate in the WAVE file format. The segments whose waveform is relatively constant were extracted from each utterance by observing the whole waveform using a digital audio analyzing software called GoldWave [5]. The 8^{th} order LPC algorithm is applied to each 20-ms segment and the frequency responses are of the 8^{th} order LPC filters.

The frequency responses of the vowel [æ] in Figures 5.1 and 5.2 tend to be relatively

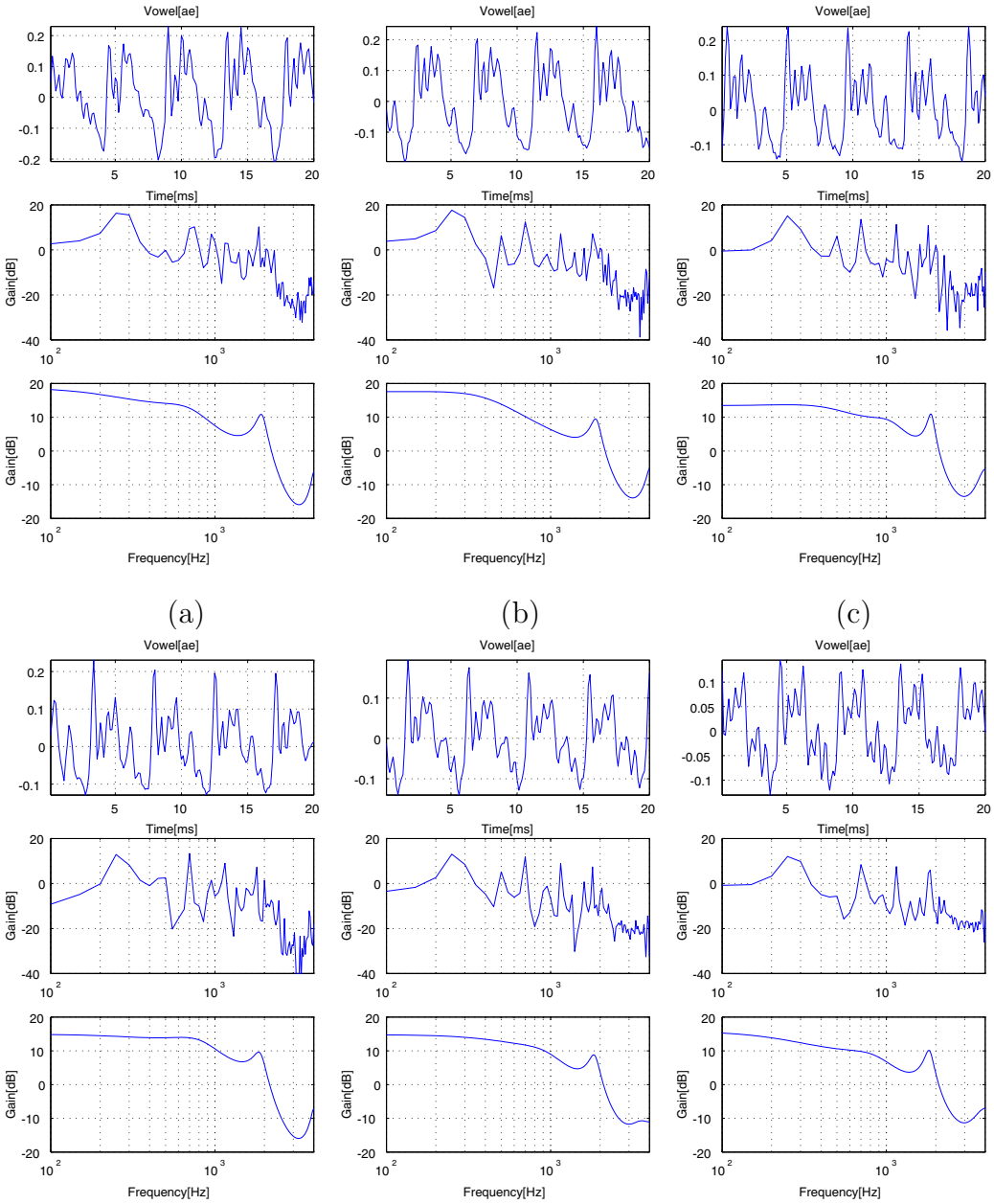


Figure 5.1 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [æ] I: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

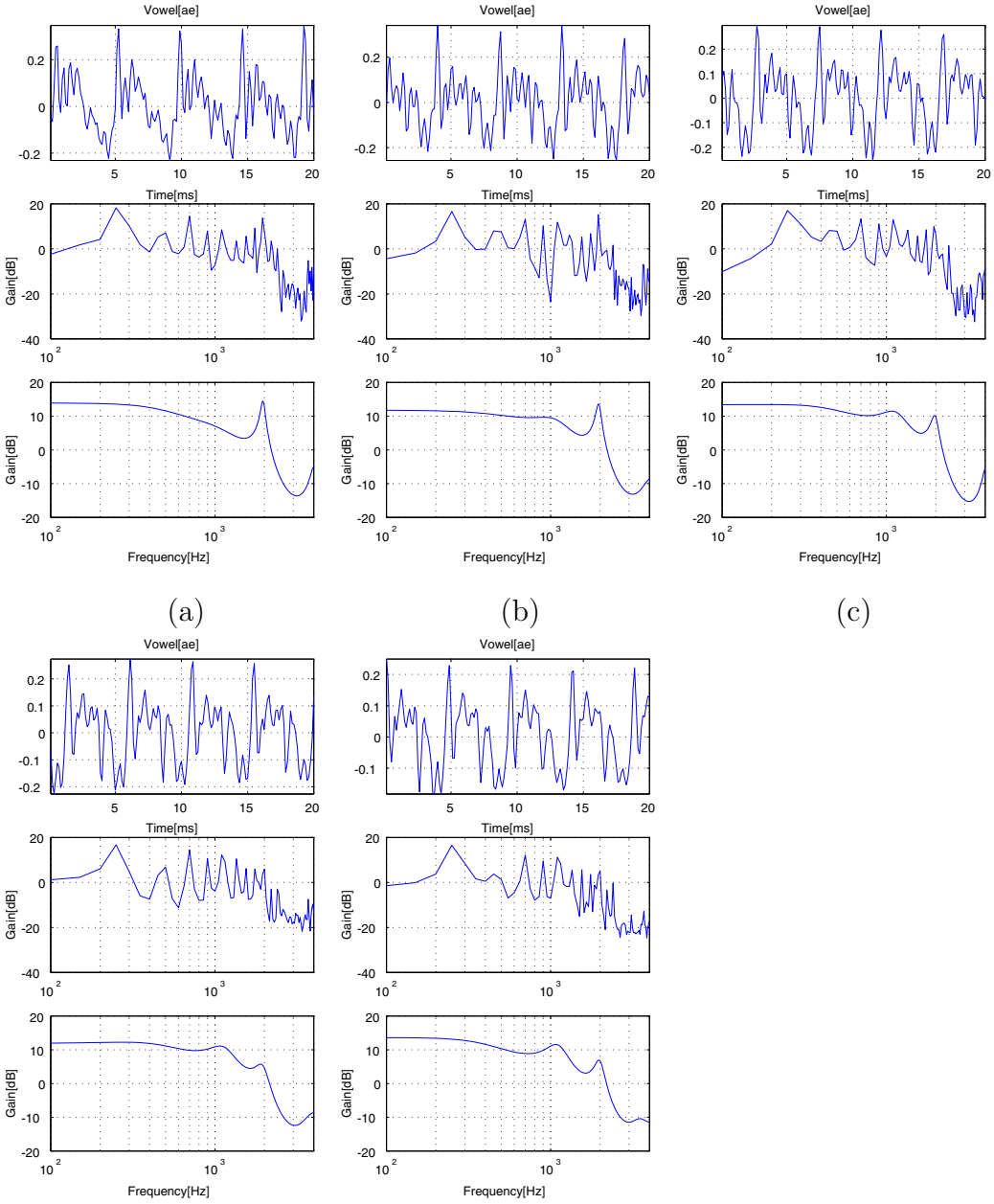


Figure 5.2 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [æ] II: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5

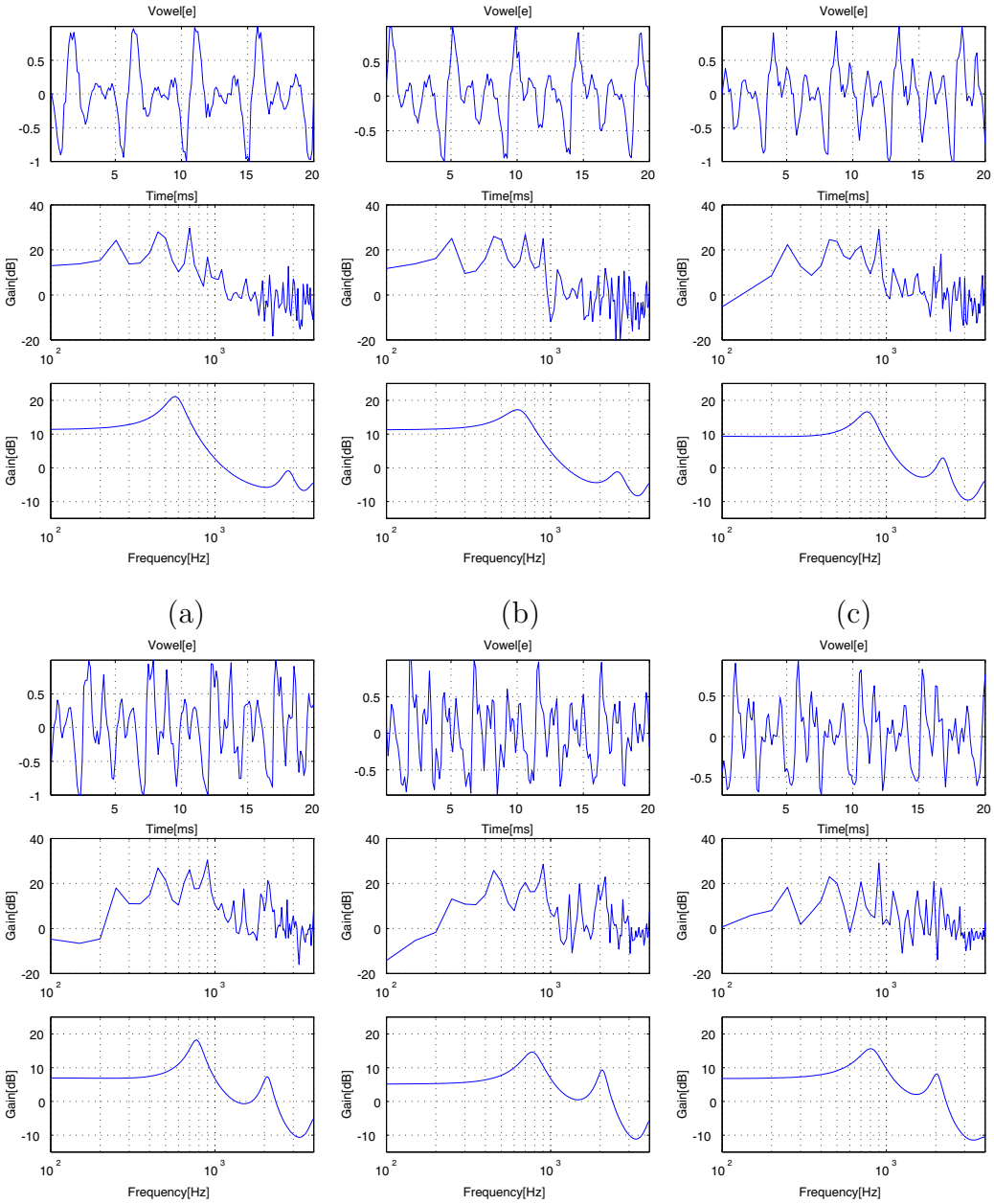


Figure 5.3 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [e] I: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

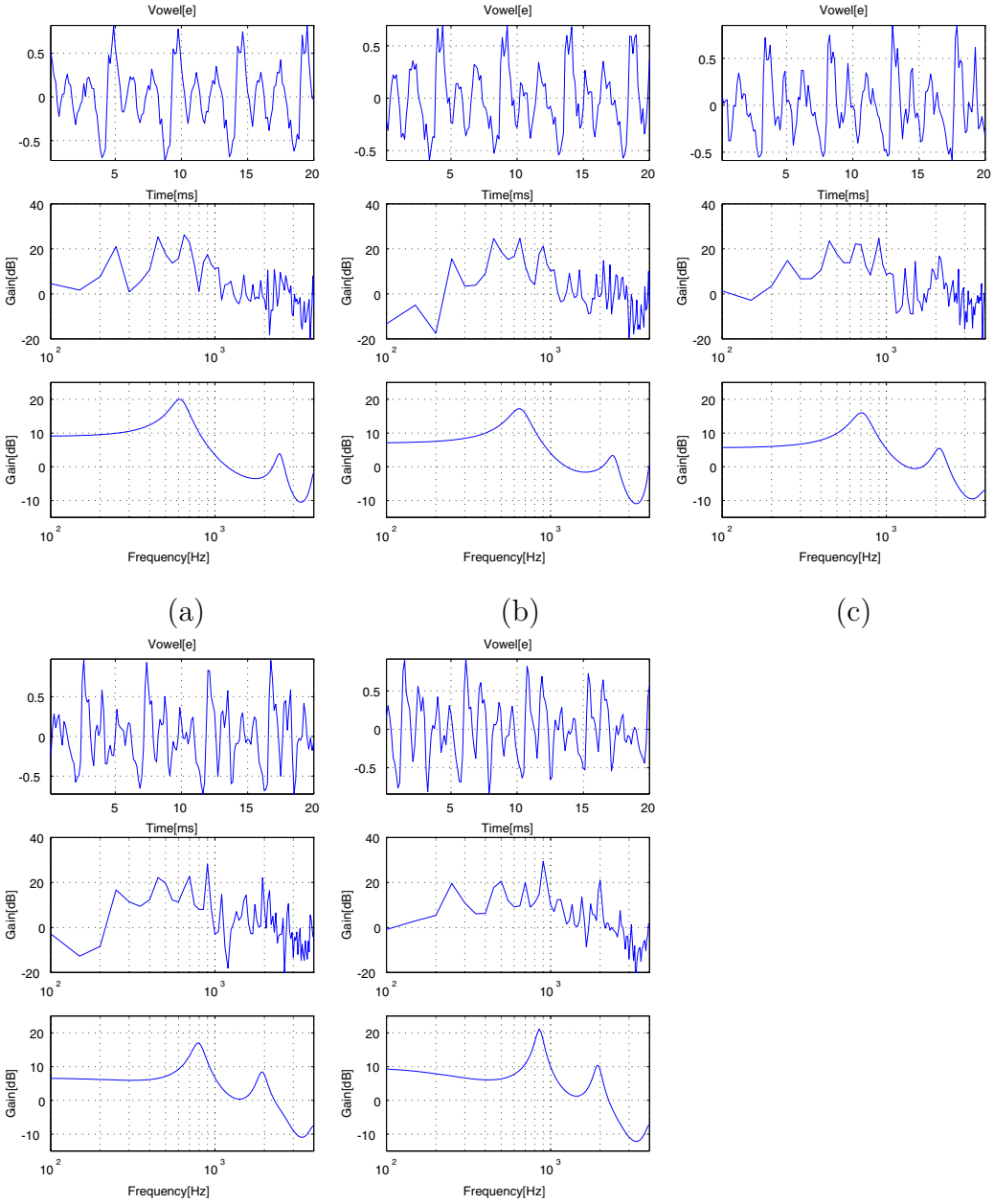
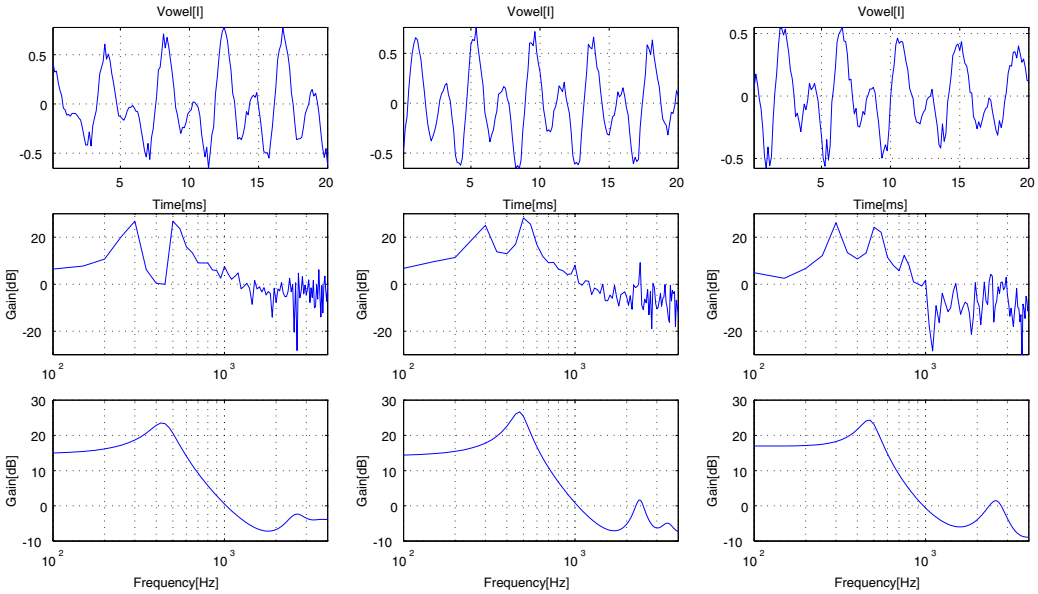
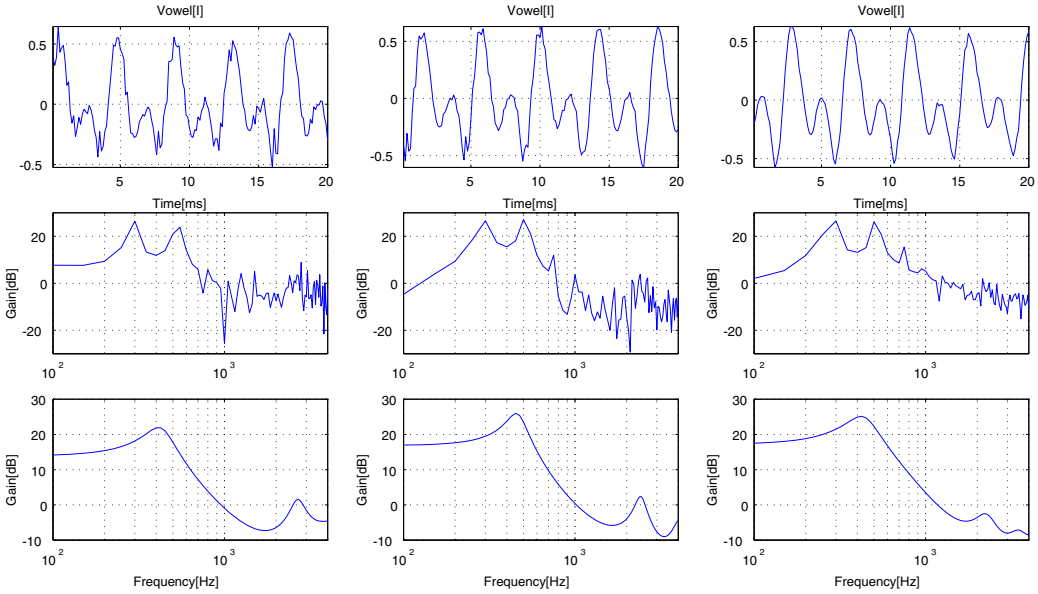


Figure 5.4 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [e] II: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5



(a) (b) (c)

Figure 5.5 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [ɪ] I: (a) segment 1, (b) segment 2, (c) segment 3



(a) (b) (c)

Figure 5.6 Vowel [ɪ] II: (a) segment 1, (b) segment 2, (c) segment 3

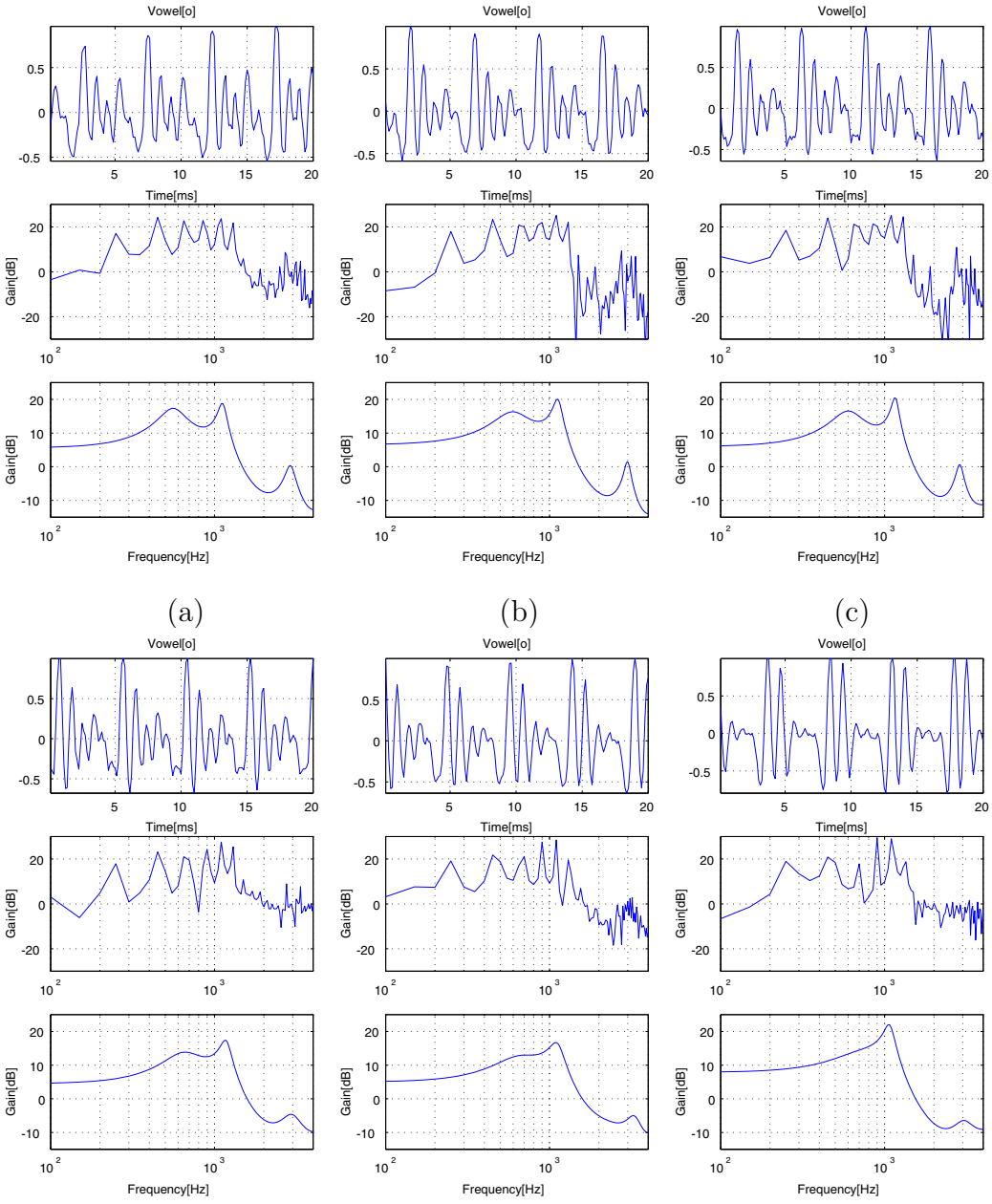


Figure 5.7 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [o] I: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

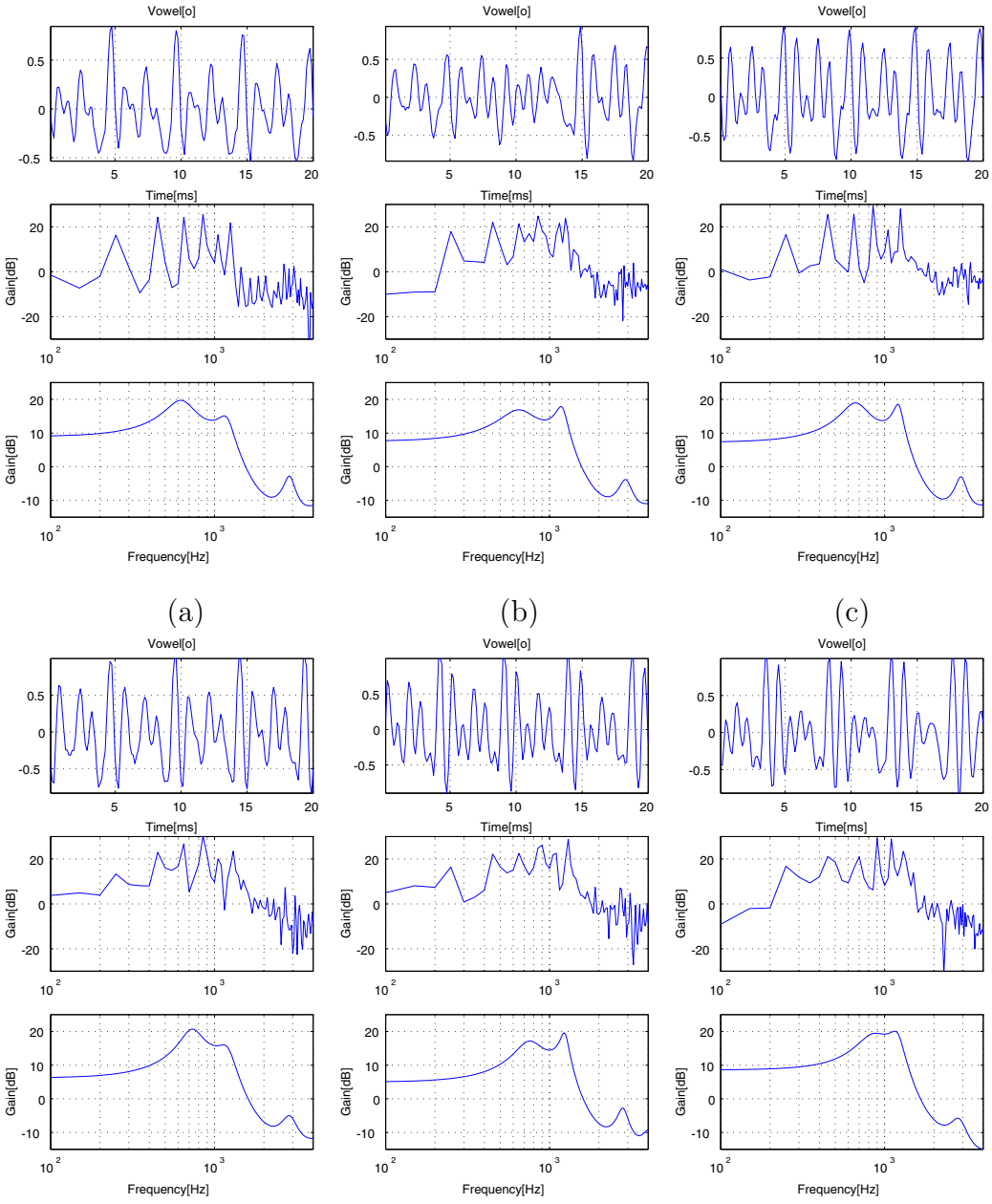


Figure 5.8 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [o] II: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

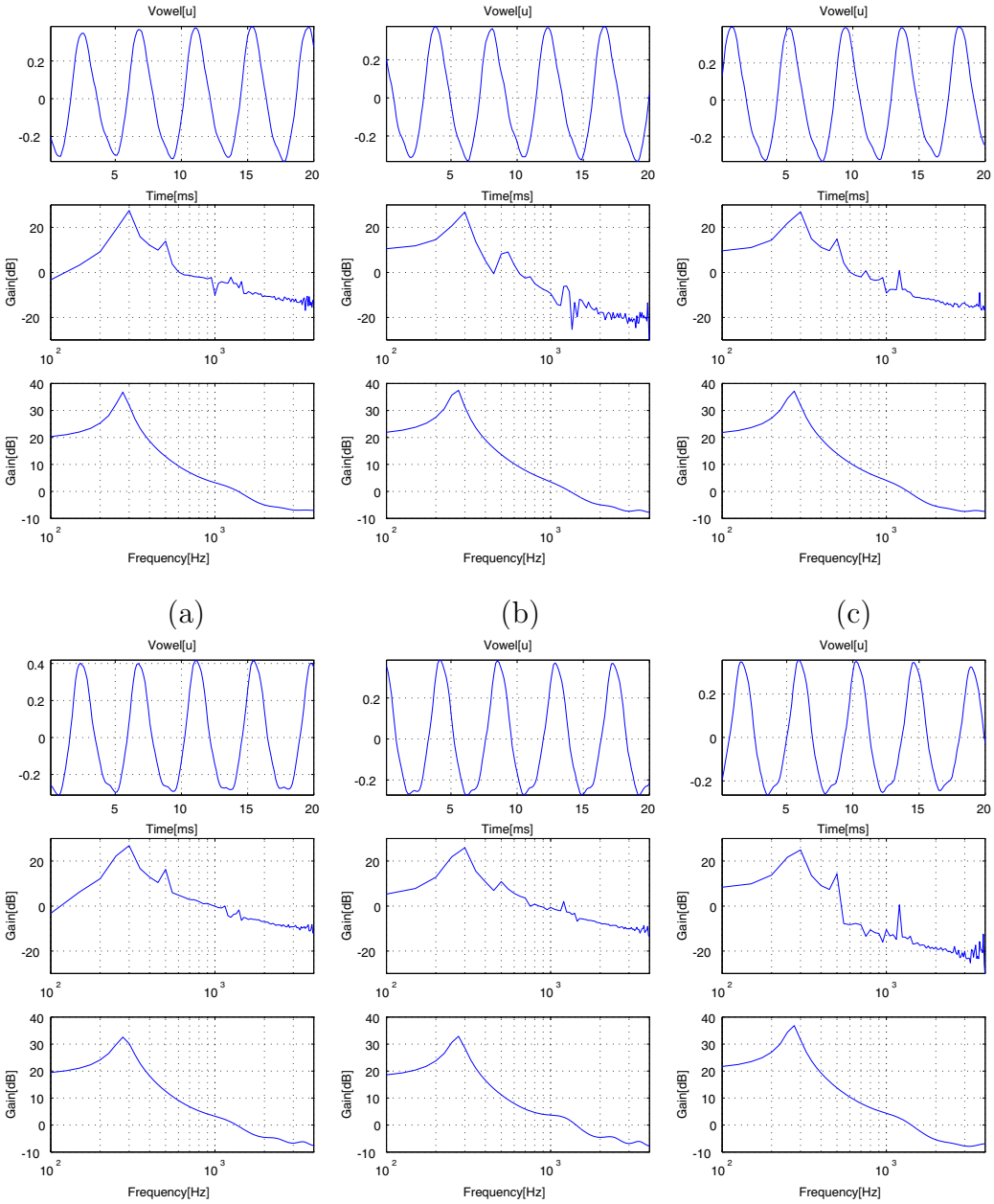


Figure 5.9 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [u] I: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

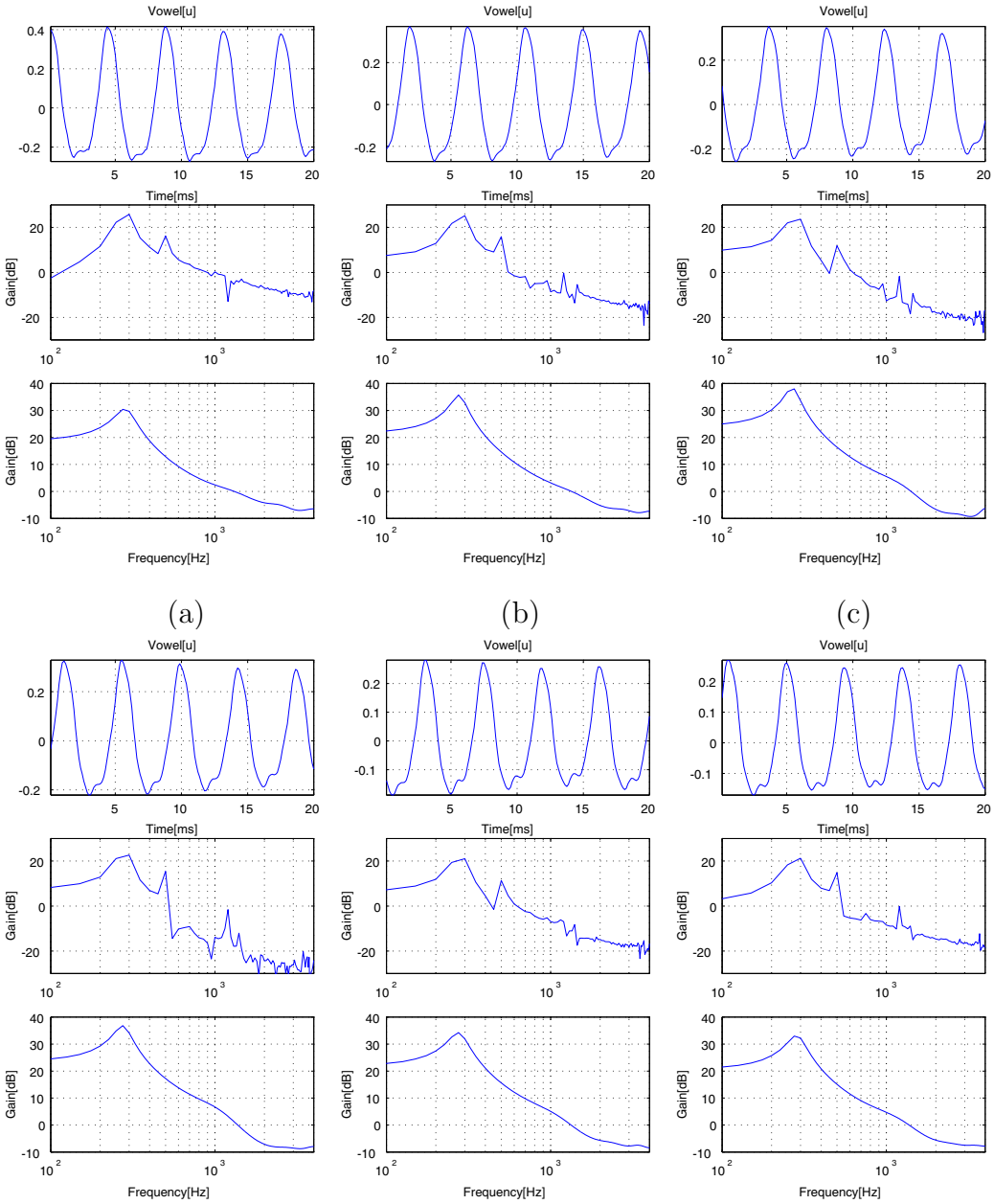


Figure 5.10 Speech waveform, signal spectrum and LPC frequency response of consecutive 20 ms segments - vowel [u] II: (a) segment 1, (b) segment 2, (c) segment 3, (d) segment 4, (e) segment 5, (f) segment 6

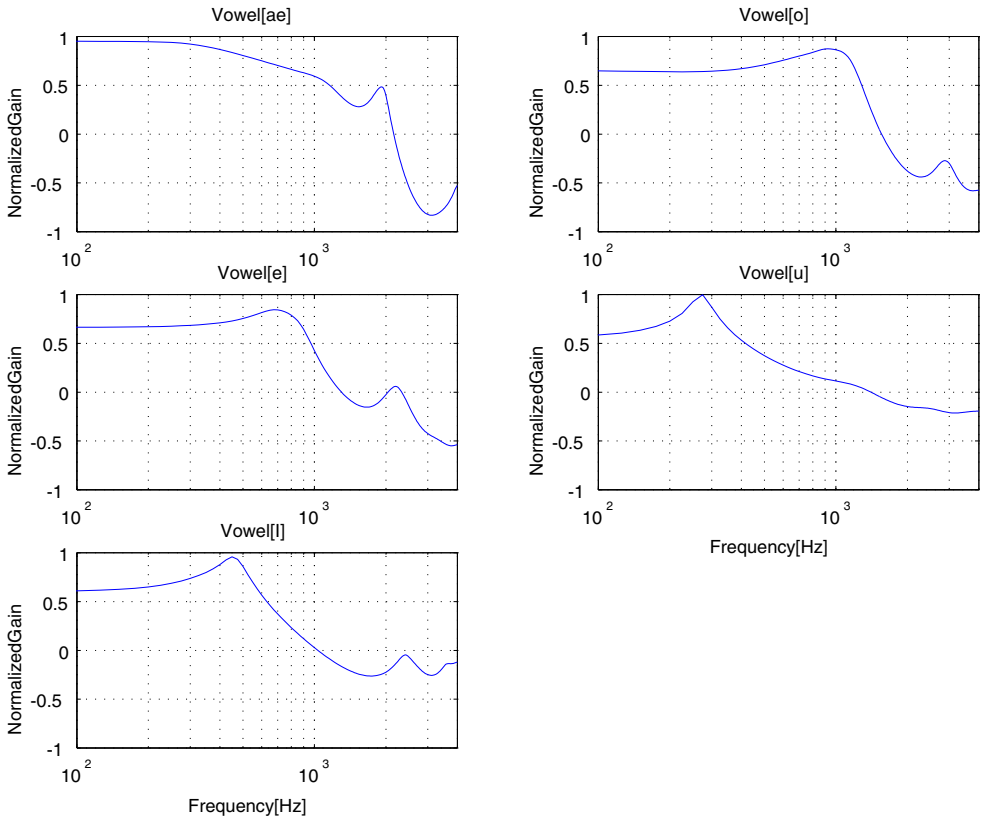


Figure 5.11 Vowels' average frequency responses of LPC filters: top row: vowel [æ], vowel [o], middle row: vowel [e], vowel [u], and bottom row: vowel [ɪ]

flat up to 1 kHz, with a small dip before a peak around 2 kHz, then decaying towards a dip at 3 kHz. The vowel [e] in Figures 5.3 and 5.4 has its first peak between 600 Hz and 800 Hz, then there is a dip, with the second peak between 2 kHz and 3 kHz. The vowel [ɪ] in Figures 5.5 and 5.6 has a peak between 400 Hz and 500 Hz, a deep dip between 1.5 kHz and 2 kHz, then a small peak between 2 kHz and 3 kHz. In most segments of the vowel [o] in Figures 5.7 and 5.8, two peaks between 500 Hz and 1.2 kHz can be observed, and the two peaks are of relatively the same gain level. There is one more small peak around 3 kHz. The vowel [u] in Figures 5.9 and 5.10 has only one peak around 300 Hz and decays gradually after the peak. It is clearly evident that each of the five vowels has different characteristics.

Figure 5.11 shows the average LPC frequency responses of the five different nor-

malized English vowel sounds by the same speaker. The number of 20-ms segment samples used to obtain the average frequency responses are: [æ] - 57, [e] - 51, [ɪ] - 43, [o] - 63, [u] - 73. It is evident that collecting the data of the vowel sounds and sorting the LPC frequency responses leads one to recognize some patterns of them.

5.2 Vowel Characterization

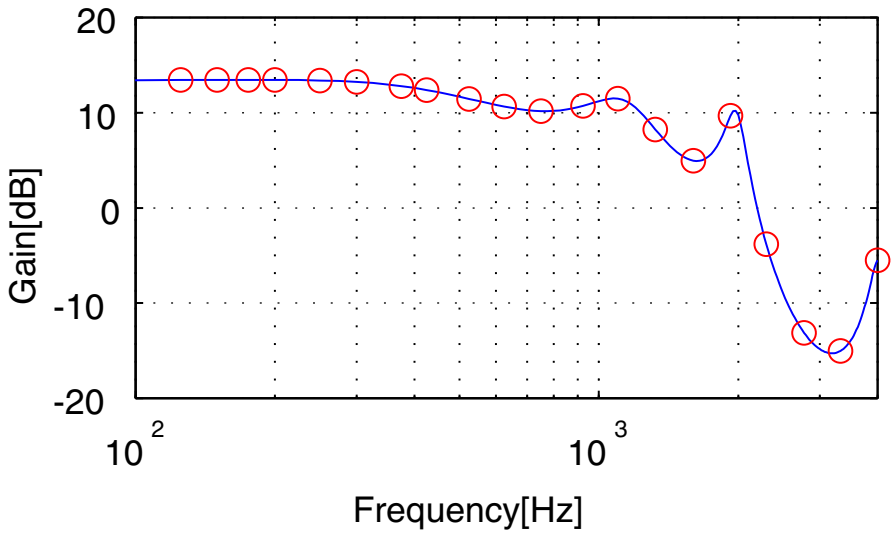


Figure 5.12 Example of 20 sample points in logarithmic scale

The proposed method for classifying a vowel sound uses Euclidian distance. Euclidian distance is a classical distance measure for pattern classification problem [14]. The Euclidian distance between two vectors and in n-dimensional space is defined by

$$\boldsymbol{d_1(x,y)} = \sqrt{\sum_{i=1}^{n}(\boldsymbol{x}_i - \boldsymbol{y}_i)^2} \tag{5.1}$$

Since important information of human speech is contained in the lower frequency range of the spectrum [19], and there are 160 samples in a 20-ms segment at 8 kHz sampling rate, 20 sample points x_i $(i = 1, 2, \ldots, 20)$ of an LPC frequency response in each segment are taken equally in logarithmic scale according to

$$x_i = \log_{10} f_{min} + \frac{\log_{10} f_{max} - \log_{10} f_{min}}{20} i \tag{5.2}$$

where $f_{max} = 4000$ (Hz) and $f_{min} = 100$ (Hz). Figure 5.12 is an example of how the 20 sample points are located in a logarithmic scale.

Then, the values of these corresponding sample points are normalized dividing by the maximum value of the segment. After the normalization, they are considered as an LPC frequency response vector $\boldsymbol{y}_j = [y_{j1}, y_{j2}, \ldots, y_{j20}]$.

Taking an adequate number, N, of the same sound segments, the average LPC frequency response vector $\overline{\boldsymbol{y}}$ is calculated by

$$\overline{\boldsymbol{y}} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{j=1}}^{\boldsymbol{N}} \boldsymbol{y}_j \tag{5.3}$$

Applying this procedure to the five individual vowel sounds, [æ], [e], [ɪ], [o], and [u], a set of five average LPC frequency response vectors, $\overline{\boldsymbol{y}}_k$ $(k = 1, 2, 3, 4, 5)$, can be made. Using these average vectors, $\overline{\boldsymbol{y}}_k$, representing the five vowels, a speech segment can be classified into one of the five vowel groups by calculating Euclidian distance $D(\boldsymbol{y}_j, \overline{\boldsymbol{y}}_k)$

$$\boldsymbol{D(y_j, \overline{y}_k)} = |\boldsymbol{y_j} - \overline{\boldsymbol{y}}_{\boldsymbol{k}}| = \sqrt{\sum_{\boldsymbol{i=1}}^{\boldsymbol{20}} (\boldsymbol{y_{ji}} - \overline{\boldsymbol{y}}_{\boldsymbol{ki}})^{\mathbf{2}}} \tag{5.4}$$

where $\boldsymbol{y}_j$ is an unknown sample segment to be classified.

First, five Euclidian distances between the unknown sample and the five average LPC frequency response vectors are calculated. Then, comparing the values of the five distances, the vowel group that has the smallest distance to the unknown sample is the group to which the unknown sample belongs.

6. RESULTS AND DISCUSSION

6.1 Outputs of RELP Vocoder

The RELP vocoder system on TMS320C6711 DSK was designed to process encoding and decoding simultaneously. The delay time was 20 ms. At the encoder, the input signal was analyzed. The PARCOR parameters, the LPC coefficients, the residual signal, and the input signal were extracted. Using the extracted PARCOR parameters and its residual signal, the synthesized signal was produced to output at the decoder. As well, the system stored the PARCOR parameters, the LPC coefficients, the residual signal, the input signal, and the synthesized signal in external memory. It was capable of storing all data up to ten seconds. The performance of the RELP vocoder was tested by comparing the input waveform of the ADC IN J7 with the output waveform of the DAC OUT J6. Instead of using voice signals as an input of the DSK directly from a microphone, pre-recorded voice signals were used. They were adjusted to fit in a fixed time duration, as well as to have a silent portion at the beginning.

6.1.1 Experimentation

The English words were stored in Microsoft (MS) WAVE sound file format (with extension .wav) in a PC prior to testing the performance of the RELP vocoder. The WAVE files were created by MS Windows' Sound Recorder of the PC, using an audio monophonic condenser microphone, which is for conventional PC use. The sampling rate of the Sound Recorder was set at 8 kHz. A male speaker and a female speaker uttered English words described in Table 6.1 into the microphone and their speech was saved in the PC. The speakers tried to utter the words not showing any emotion or fatigue. Each word was saved as one WAVE file so that it could be

Table 6.1 Tested English Words for RELP Vocoder

VOWEL			CONSONANT		
WORD	TYPE	phoneme	WORD	TYPE	phoneme
run	short	ʌ	boy	stop (voiced)	b
cat	short	æ	dog	stop (voiced)	d
bed	short	e	get	stop (voiced)	g
sit	short	ɪ	cat	stop (unvoiced)	k
cosy	short	i	pal	stop (unvoiced)	p
see	short	i:	top	stop (unvoiced)	t
hot	short	*a*	valentine	fricative (voiced)	v
no	long	ou	zoo	fricative (voiced)	z
made	long	eɪ	vision	fricative (voiced)	zh
wine	long	*a*ɪ	this	fricative (voiced)	th
read	long	ɪ:	funny	fricative (unvoiced)	f
fume	long	u:	see	fricative (unvoiced)	s
arm	r-controlled	*a*:	she	fricative (unvoiced)	ʃ
her	r-controlled	ə:	thin	fricative (unvoiced)	θ
pore	r-controlled	o:	lad	semivowel (liquid)	l
ago	other	ou	wall	semivowel (liquid)	w
put	other	ʊ	red	semivowel (glide)	r
too	other	u:	yes	semivowel (glide)	j
my	diphthong	*a*ɪ	me	nasal	m
pipe	diphthong	əɪ	no	nasal	n
how	diphthong	*a*u	ring	nasal	ng
house	diphthong	ʌu	chip	affricate	tʃ
day	diphthong	ei	jar	affricate	dzh
boy	diphthong	oi	hot	whisper	h
			loch	clearing throat sound	x

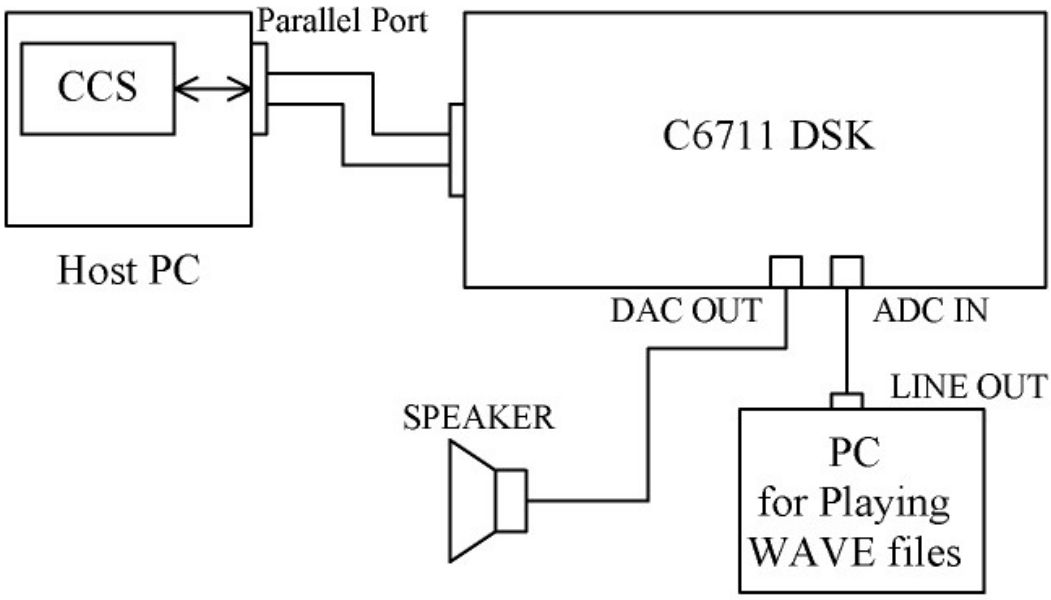


Figure 6.1 Block diagram of experimental system

played back individually. The tested words were chosen to cover American English phonemes found in Rabiner and Schafer [18], as well as additional phonemes listed in The Canadian Oxford Dictionary [16]. The words recorded and their phonetic classifications are listed in Table 6.1.

Figure 6.1 is a block diagram of the experimental system to test the performance of the RELP system. The DSK was connected to a parallel port of the host PC. CCS of the PC controlled the DSK. Another PC which stored the WAVE files of the spoken words was used to play back the files. The line out of the PC was connected to the ADC IN J7 of the DSK. The DAC OUT J6 of the DSK board was connected to the active speakers so that one could monitor the output synthesized signal. The orders of both of the LPC filter and the lattice filter were set eight. Each WAVE file was tested by changing the format of the residual signal: 32-bit floating point, 16-bit fixed point, 14-bit fixed point, 12-bit fixed point, 10-bit fixed point, and 8-bit fixed point. Each time a word file was processed by the vocoder, the numerical data stored in external memory on the DSK, such as the PARCOR parameters, the LPC coefficients, the synthesized speech signals, was transferred to the host PC and saved as text files for subsequent analysis.

6.1.2 Results

Figures 6.2 through 6.5 illustrate some of the results. They are the waveforms of the input and synthesized speech signals plotted by MATLAB® using the data transferred from the external memory of DSK. The utterances of Figures 6.2 and 6.3 were by the female speaker. The words were 'cat' and 'sit' respectively. Figures 6.4 and 6.5 were by the male speaker and the words were 'cat' and 'sit', as well. Each figure shows: 1. input signal (top row), 2. output signal using floating point residual (second row), 3. output signal using 16-bit fixed point residual (third row, left), 4. output signal using 14-bit fixed point (third row, right), 5. output signal using 12-bit fixed point (fourth row, left), 6. output signal using 10-bit fixed point (fourth row, right).

As these figures show clearly, the synthesized signals which used the floating point residual as an excitation signal were reproduced almost perfectly compared to the input signal in terms of the amplitude of big swings as well as the feeble waves of some consonants. It was difficult to distinguish the synthesized speech of the speakers from the original input speech. The synthesized speech sounded very natural and smooth.

With the fixed point format residuals, the graphs of the output signals show that most of the contents of the original speech utterances remain in the outputs. However, as the bit rate gets smaller, periodic noises become noticeable, especially in the low amplitude parts. The periodic noises are synchronized to the vocoder's processing period, 20 ms. This causes changes in the waveforms of the weak utterances. The effect can be observed easily in the graphs of 10-bit fixed point residuals. The sounds of the output using fixed point residual have most of characteristics of the words and the speaker's voice so that one can easily understand the words and recognize the speakers, even though the noises were mixed in their speech.

The waveforms of 8-bit fixed point residual, which are not shown, depart significantly from the original waveforms. It was difficult to understand words in the synthesized speech.

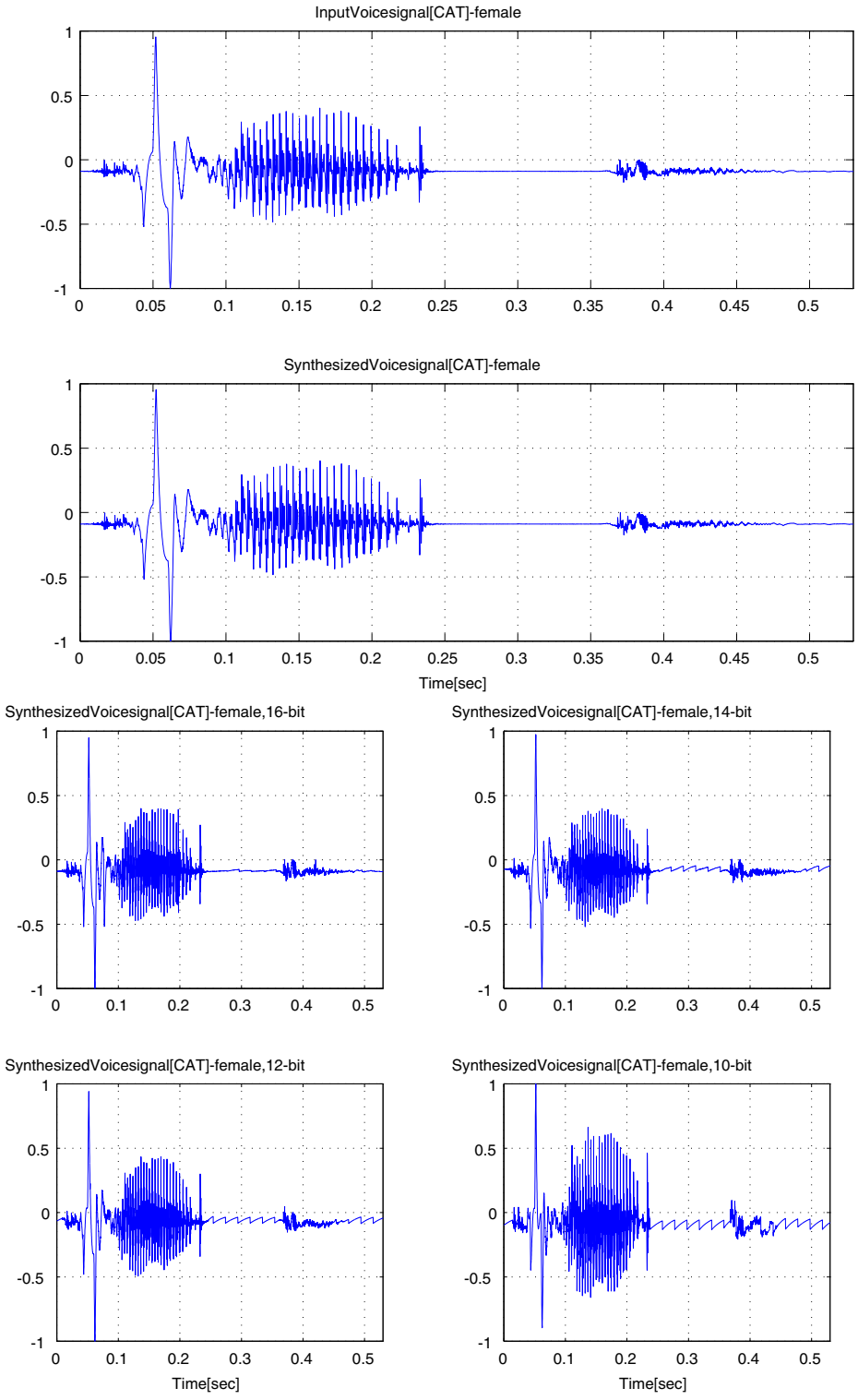


Figure 6.2 Input and synthesized voice signals - [CAT (female)] (Top - Bottom): Input, Synthesized (Floating point residual, Fixed point residuals - 16-bit, 14-bit, 12-bit, 10-bit)

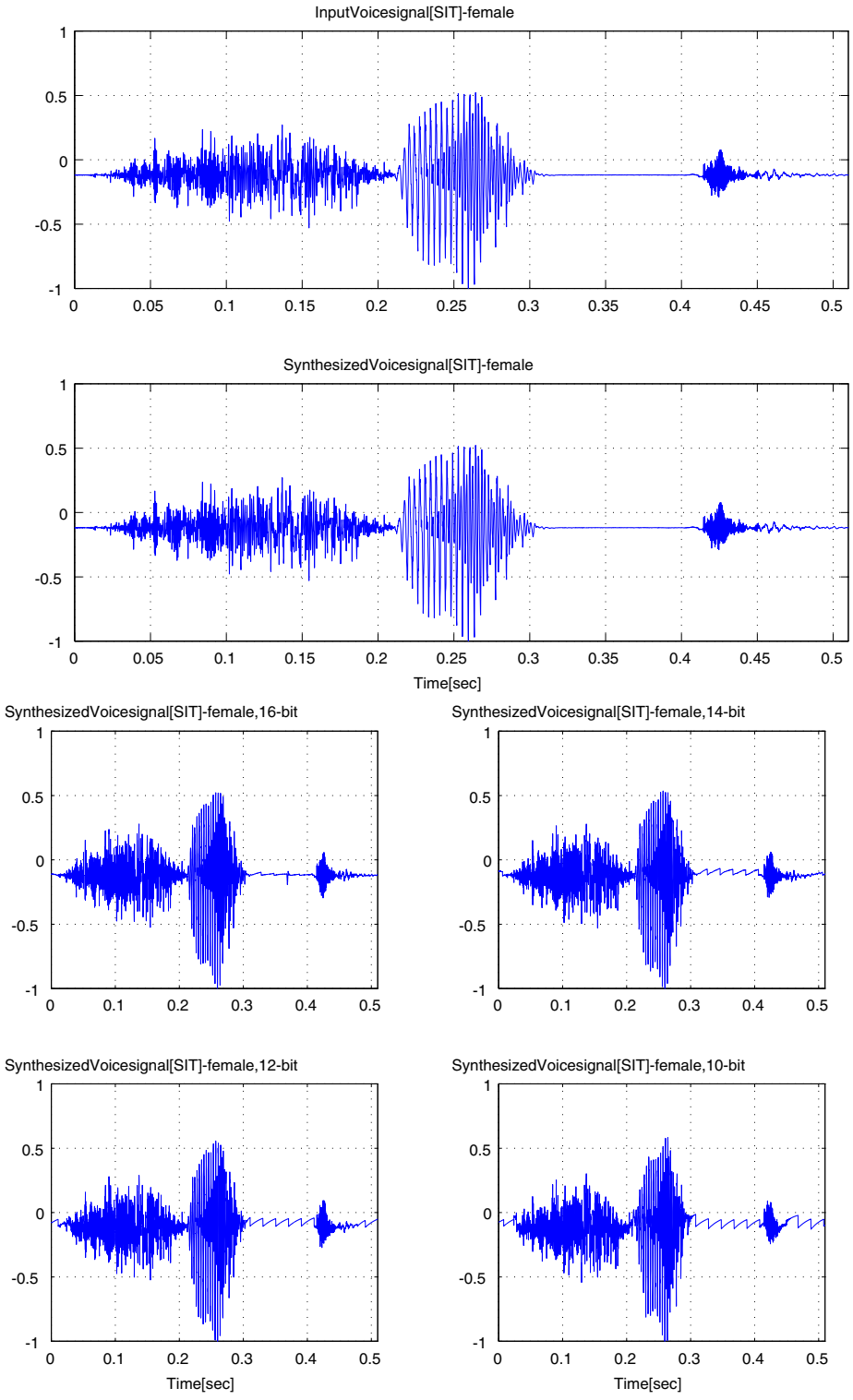


Figure 6.3 Input and synthesized voice signals - [SIT (female)] (Top - Bottom): Input, Synthesized (Floating point residual, Fixed point residuals - 16-bit, 14-bit, 12-bit, 10-bit)

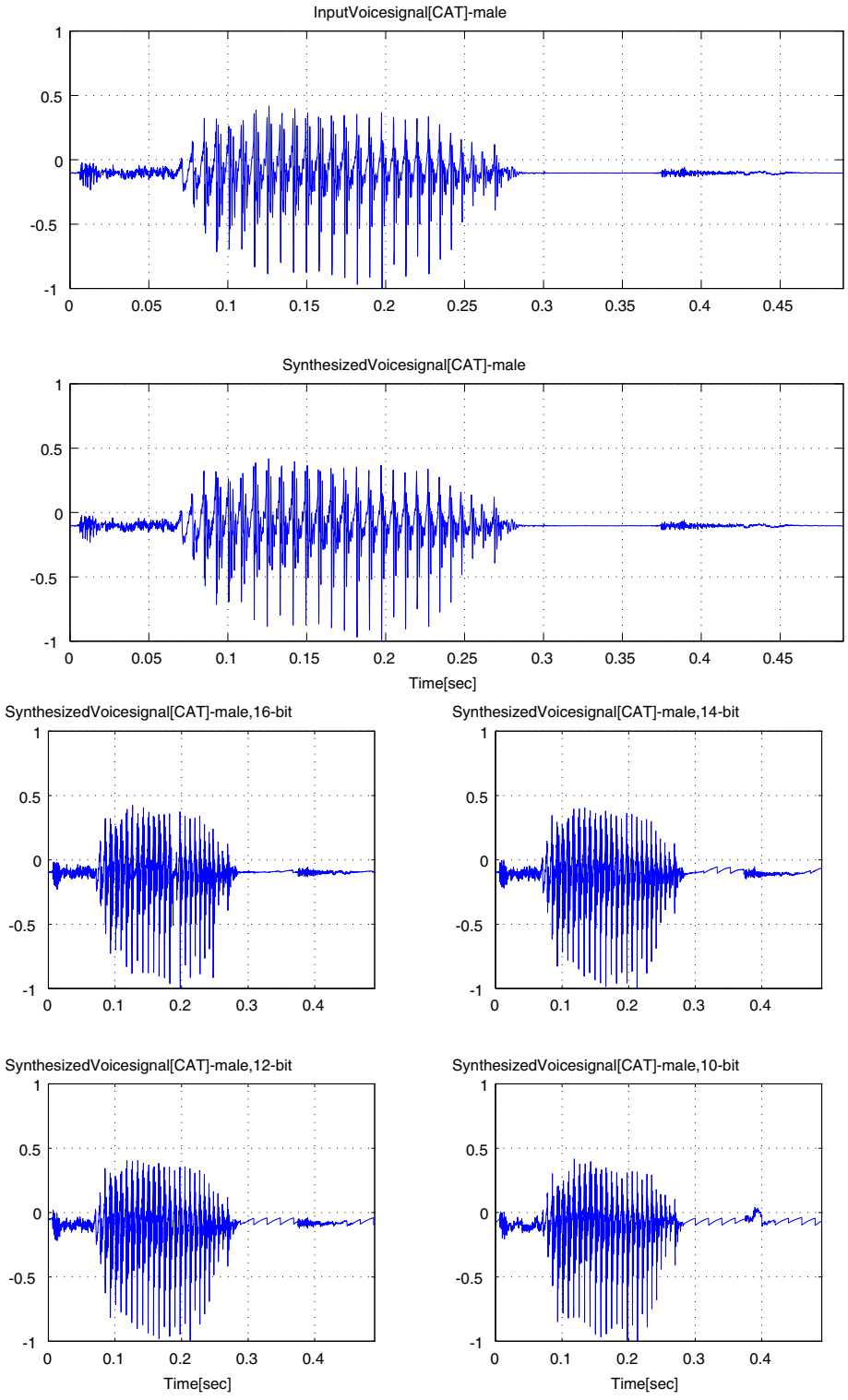


Figure 6.4 Input and synthesized voice signals - [CAT (male)] (Top - Bottom): Input, Synthesized (Floating point residual, Fixed point residuals - 16-bit, 14-bit, 12-bit, 10-bit)

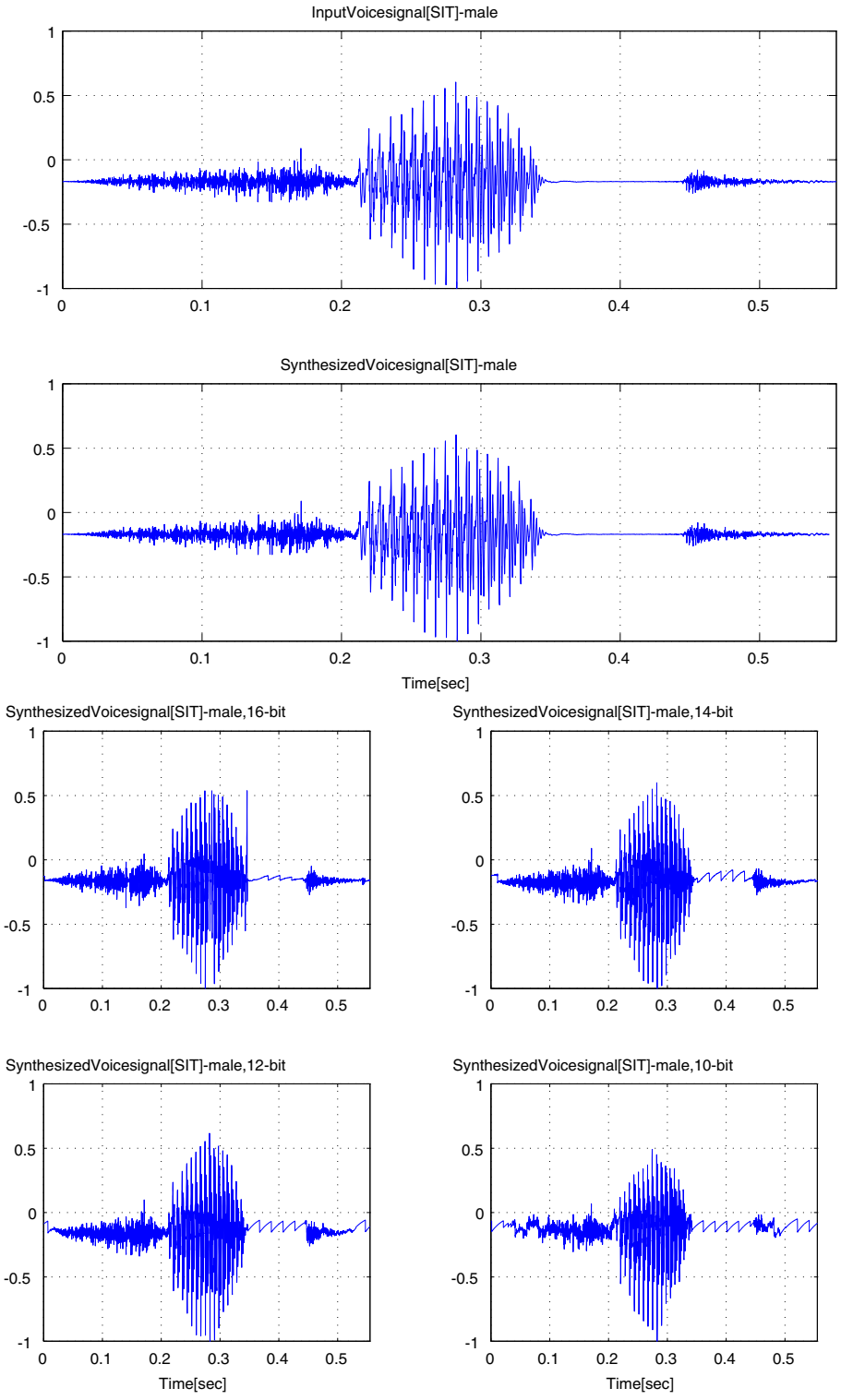


Figure 6.5 Input and synthesized voice signals - [SIT (male)] (Top - Bottom): Input, Synthesized (Floating point residual, Fixed point residuals - 16-bit, 14-bit, 12-bit, 10-bit)

6.2 Vowel Characterization

The simulation of vowel characterization was done by MATLAB® programs. A simulation program of the RELP vocoder was programed so that (an) arbitrary segment(s) can be processed and the required LPC frequency response vector(s) of the segment(s) can be calculated off-line. The five English vowels, [æ], [e], [ɪ], [o] and [u], were chosen to be examined. To verify whether the vowel classification method is effective for vowels in words, a one-syllable English word was chosen for each of the five vowels. The selected words were 'at' ([æ]), 'bed' ([e]), 'Kim' ([ɪ]), 'Tom' ([o]), and 'too' ([u]).

6.2.1 Experimental Note

The five individual vowels and the five words were uttered by the same speaker. For individual vowels, ten utterances of each vowel were recorded in the manner described in Subsection 6.1.1. Twelve utterances of each word were recorded for vowels in one-syllable words. Segments of 20 ms were determined from each utterance by observing the entire waveform of the utterance displayed by the software GoldWave [5]. Then the segments were processed through the RELP vocoder program, which outputs the LPC frequency response vectors. The five average LPC frequency response vectors were subsequently calculated from the vectors for each vowel. These average vectors were treated as reference vectors to classify unknown vowels. The number of vectors to calculate a reference vector varied from 43 to 73.

The number of segments taken from each utterance varied because the length of the vowel utterance differs depending on the chosen vowel or word. For instance, there were typically five or six segments of [æ], four or five segments of [e], three or four segments of [ɪ], six or seven segments of [o], and seven or eight segments of [u]. The lengths of the vowels in one-syllable words tended to be shorter than those of the individual vowels.

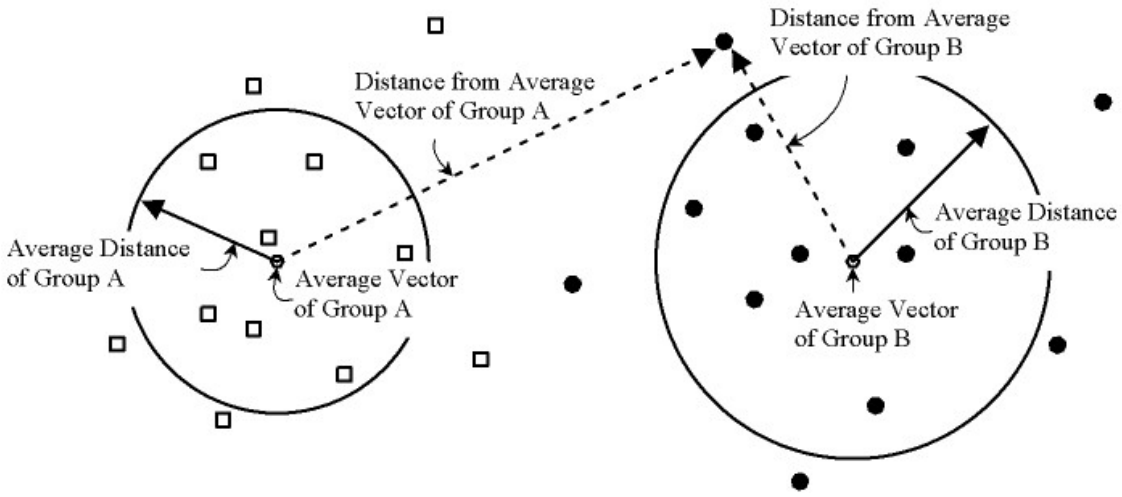


Figure 6.6 Example of distances and average distances

6.2.2 Individual Vowels

The Euclidian distances of each average LPC frequency response vector to the LPC frequency response vectors of all five vowel groups were calculated. Figure 6.6 illustrates the terms used in this chapter. In the figure, there are two groups of samples, (vowel) Group A and (vowel) Group B. The squares are the samples (the LPC frequency response vectors) of Group A and the black dots are the samples of Group B. The two circles are drawn to indicate the average distance of a group. The center of each circle is the average LPC frequency response vector. The distance from the center of each circle to a sample represents a Euclidian distance. The dotted lines show the Euclidian distances from each average vector to a sample. The distances from an average vector to samples were compared for all vowels to determine the minimum distance.

Figures 6.7 through 6.11 are histograms of the Euclidian distances from an average LPC frequency response vector to vectors of vowel groups. In each figure, the left histogram of the top row illustrates the distribution of the distances from an average vector to vectors of its own vowel group. The rest of the histograms indicate the distribution of the distances to vectors of the other four vowel groups. In each figure, the histograms show that the clusters of the distances to vectors of its own vowel

group are the closest to the origin.

Table 6.2 shows the average values of the Euclidian distances from each average LPC frequency response vector to the vectors of the five vowel groups. Figure 6.12 illustrates this in a graph. Both indicate that the average values to its own LPC frequency response vector are the smallest in each of the five vowel groups.

Of note is that the loci of the vowels [ɪ] and [u] in Figure 6.12 are similar when compared with the other three vowels. The average values of the vowel [ɪ] and the vowel [u] are the second smallest to each other (0.4488, 0.9759 and 0.1586, 0.8685). Both are placed in the farthest positions from the average vectors of the vowels [æ] and [o]. As observed in the average frequency responses of LPC filters (Figure 5.11), the shapes of the two average LPC frequency responses are similar. This indicates that the vowels [ɪ] and [u] may not be clearly distinguishable from each other.

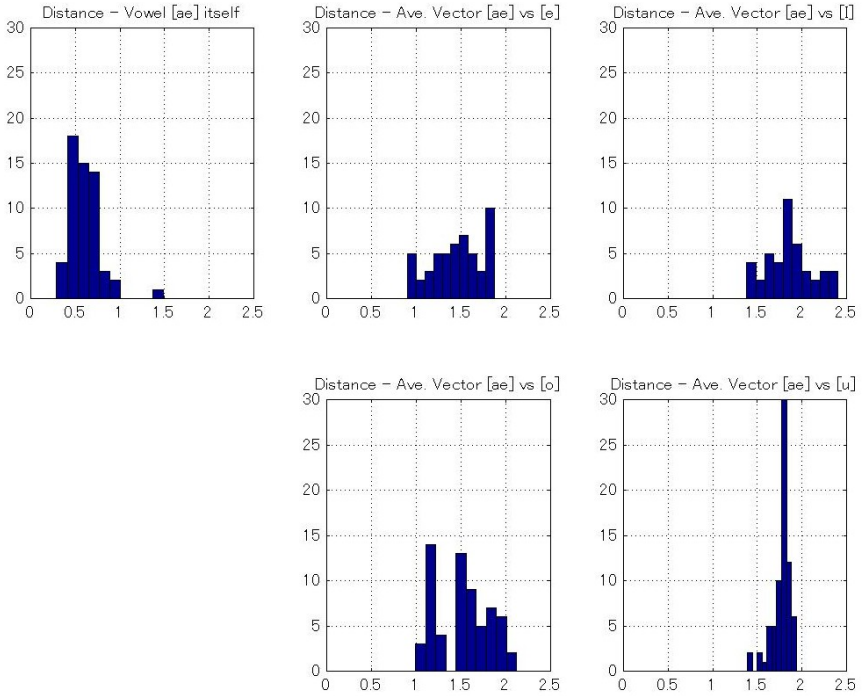


Figure 6.7 Histograms of distances to average LPC frequency response vector of vowel [æ]

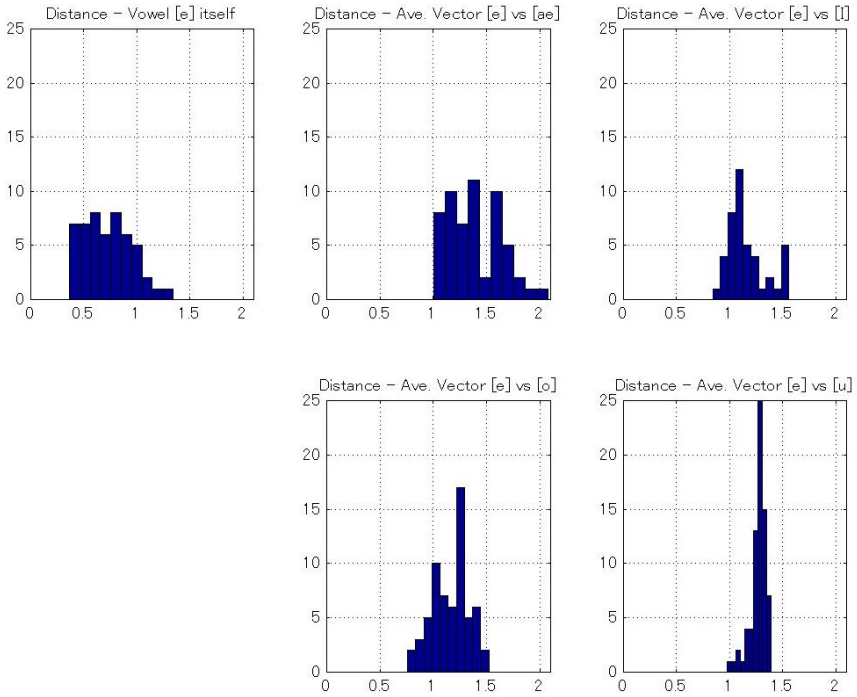


Figure 6.8 Histograms of distances to average LPC frequency response vector of vowel [e]

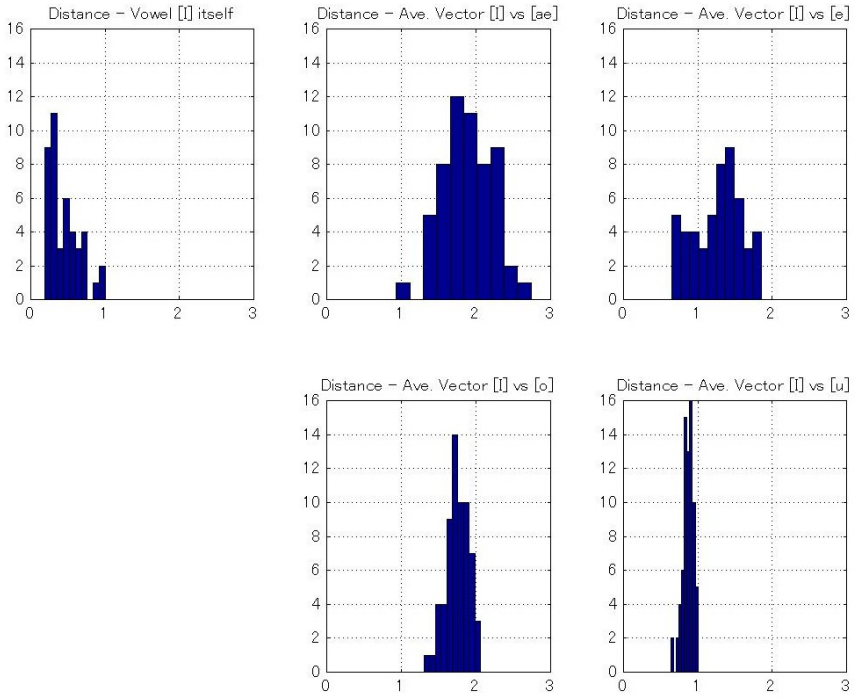


Figure 6.9 Histograms of distances to average LPC frequency response vector of vowel [ɪ]

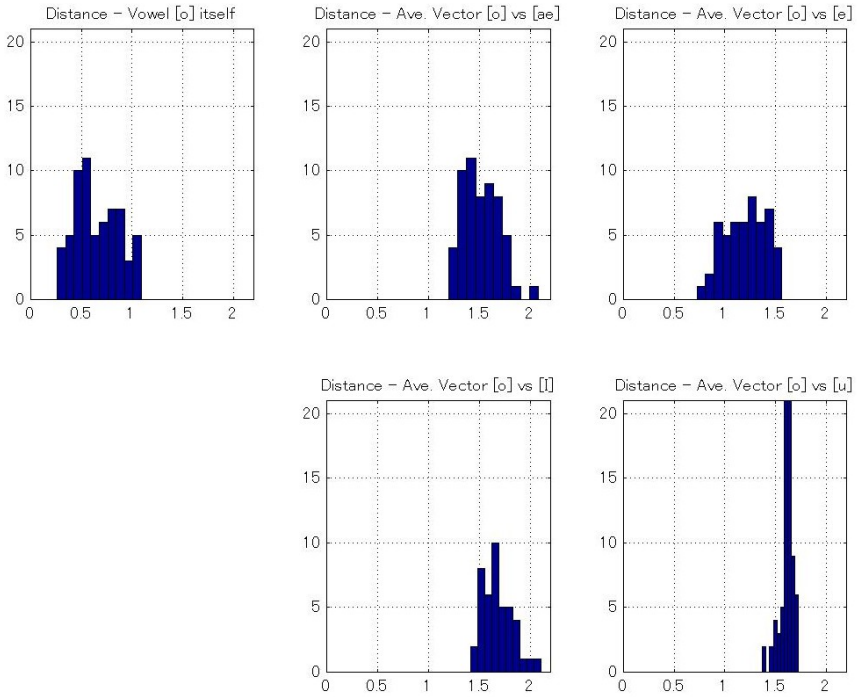


Figure 6.10 Histograms of distances to average LPC frequency response vector of vowel [o]

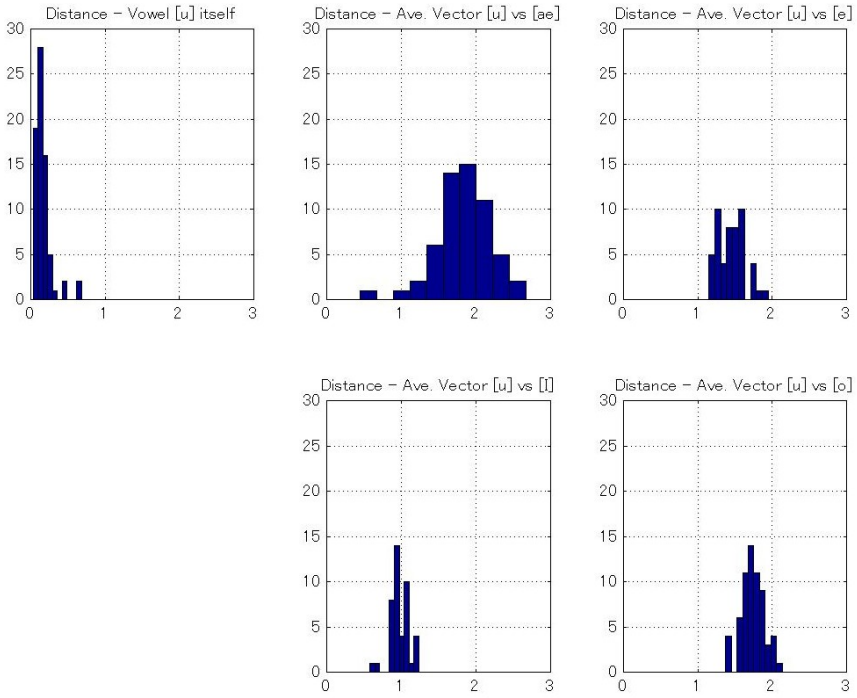


Figure 6.11 Histograms of distances to average LPC frequency response vector of vowel [u]

Table 6.2 Average Euclidian Distances of Individual Vowels by the Female Speaker

	[æ]	[e]	[ɪ]	[o]	[u]	#†
[æ]	0.5926	1.4554	1.8665	1.5250	1.7697	57
[e]	1.3942	0.7242	1.1538	1.1676	1.2703	51
[ɪ]	1.8912	1.2608	0.4488	1.7495	0.8685	43
[o]	1.5120	1.2030	1.6818	0.6587	1.6101	63
[u]	1.8330	1.4574	0.9759	1.7368	0.1586	73

†: Number of sample segments

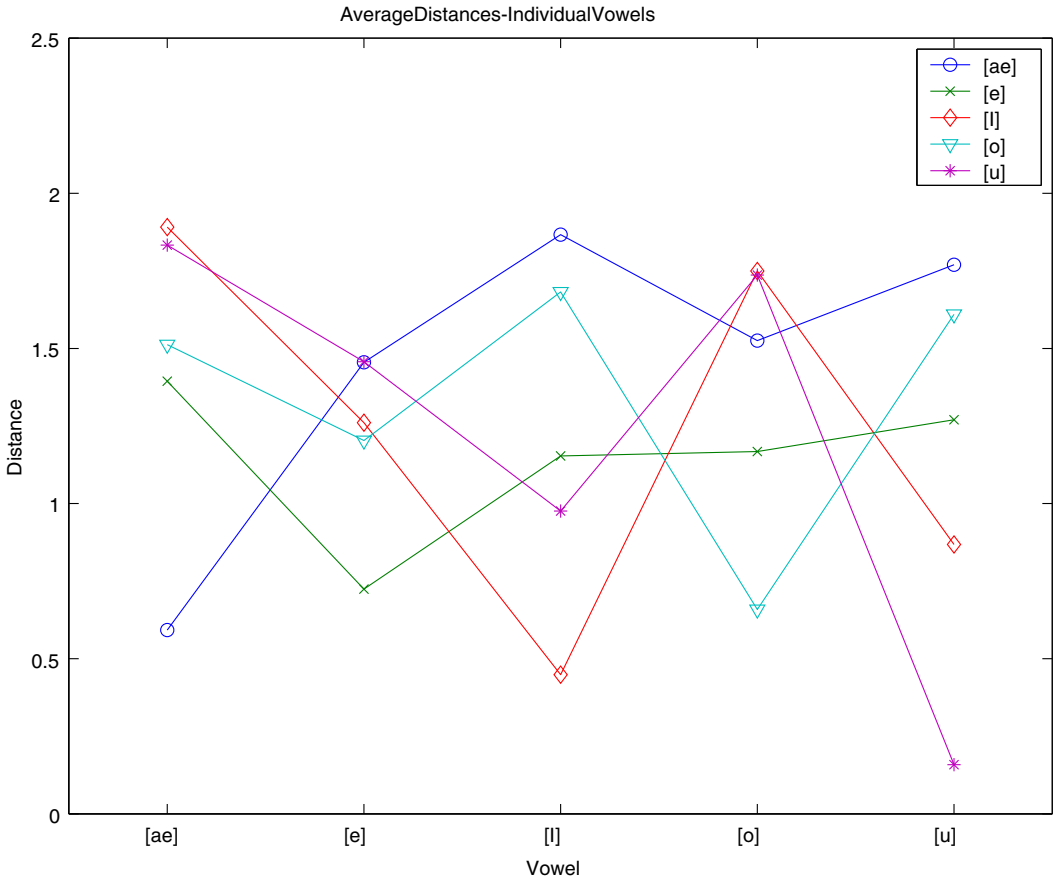


Figure 6.12 Average Euclidian distances of individual vowels

6.2.3 Vowels in One-syllable Words

Vowels in one-syllable words were examined to determine how vowels attached before and/or after a consonant affect the proposed method of vowel classification. One-syllable words, 'at', 'bed', 'Kim', 'Tom', and 'too' are used to extract [æ], [e], [ɪ], [o], and [u] influenced by consonants. The number of the segments of [e] in 'bed' was 37. This is because the utterances having a reasonable quality to be processed could not be available from the recording. Figure 6.13 through Figure 6.17 are histograms of the Euclidian distances to each of the average LPC frequency response vectors used in Subsection 6.2.2.

From these figures, it is found that the clusters of the vowel groups are the closest to their own vowel's average LPC frequency response vector. However, the clusters to their own average vector and the clusters to the other average vectors have more overlapped areas than those of individual vowels as in Figures 6.7 to 6.11.

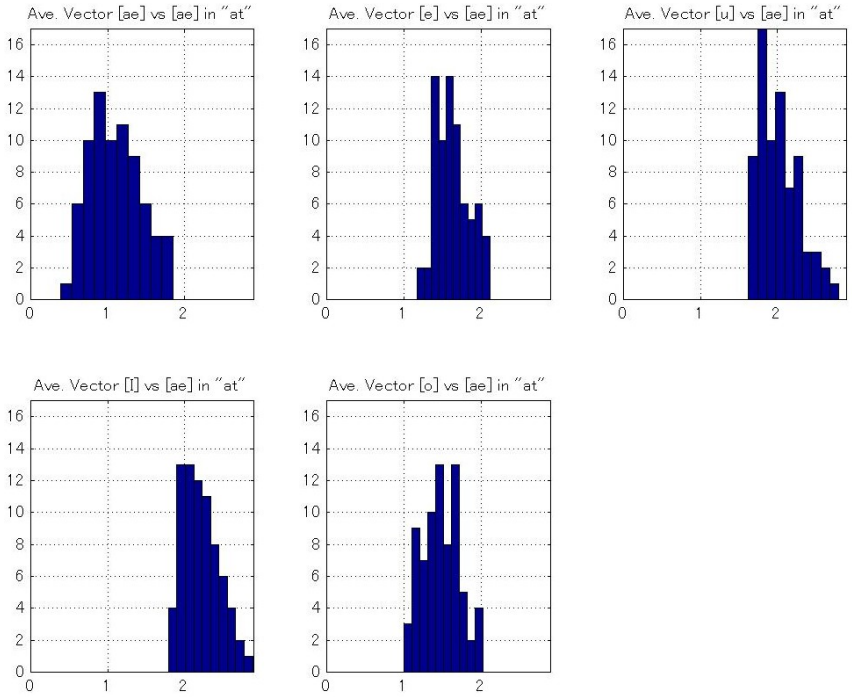


Figure 6.13 Histograms of distances to average LPC frequency response vector of vowel [æ]

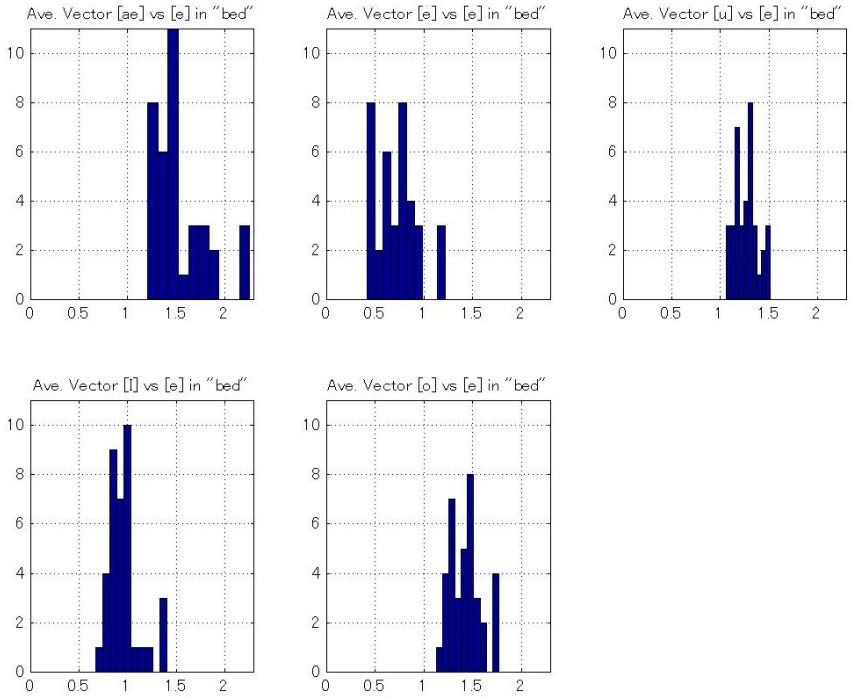


Figure 6.14 Histograms of distances to average LPC frequency response vector of vowel [e]

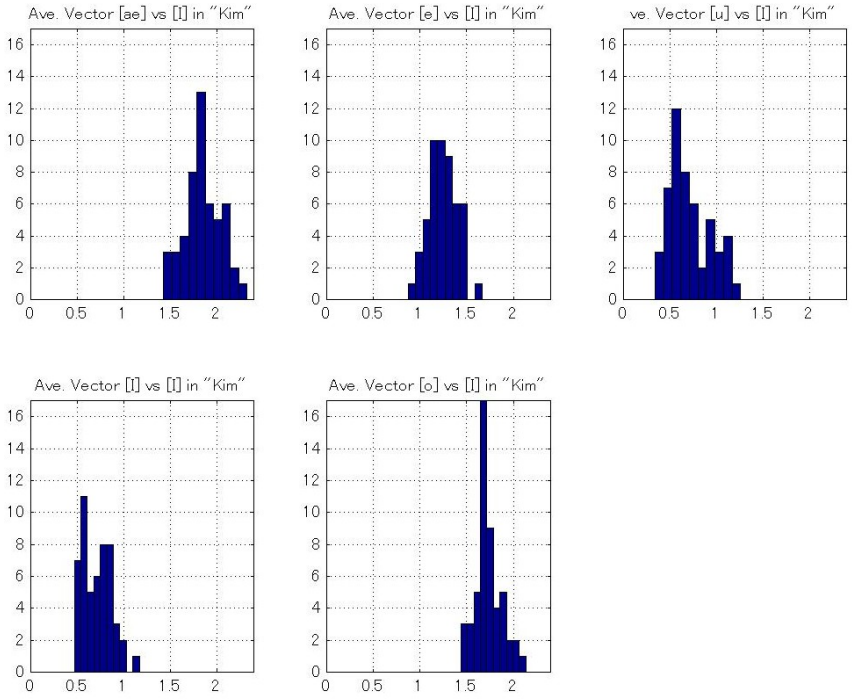


Figure 6.15 Histograms of distances to average LPC frequency response vector of vowel [ɪ]

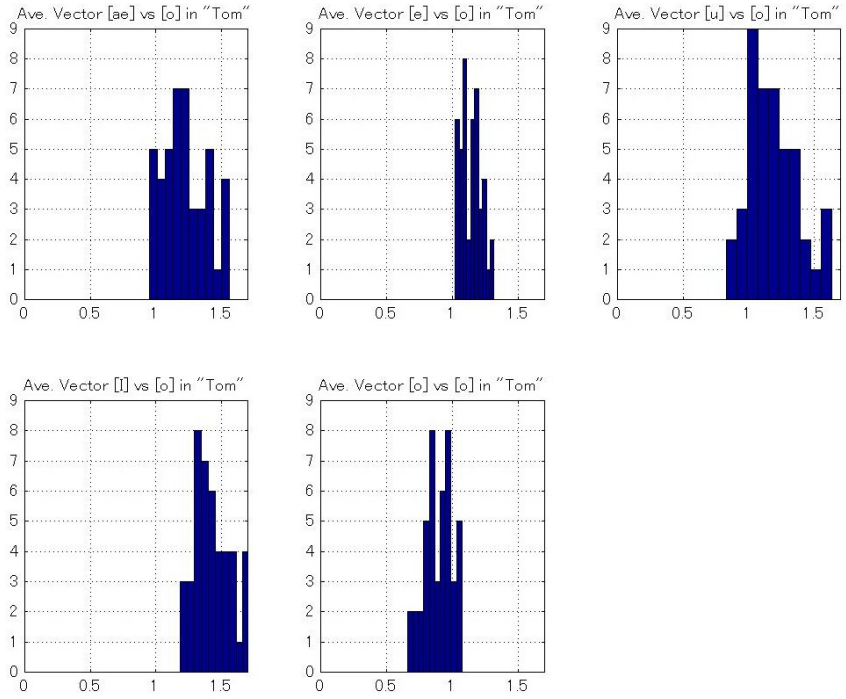


Figure 6.16 Histograms of distances to average LPC frequency response vector of vowel [o]

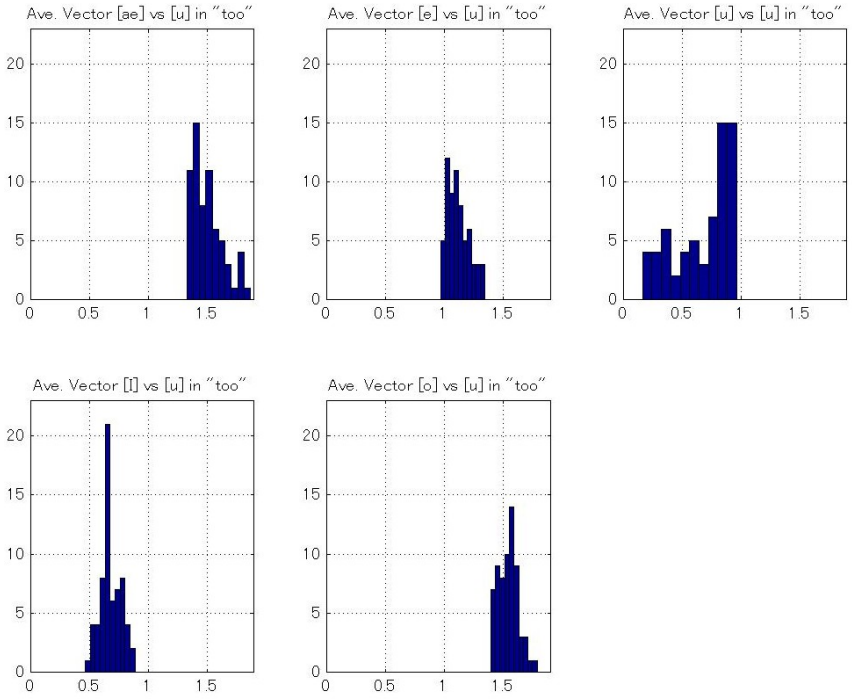


Figure 6.17 Histograms of distances to average LPC frequency response vector of vowel [u]

In Figure 6.13, for example, the cluster found in Ave. Vector [æ] vs [æ] in "at" is distributed between 0.5 and 1.8. The clusters found in Ave. Vector [e] vs [æ] in "at" and Ave. Vector [o] vs [æ] in "at" are both distributed between 1 and 2. This indicates the vowel [æ] in "at" could be misclassified into either vowel groups [e] or [o] because of the overlapped areas. Similarly, the overlapped areas between vowel [ɪ] and vowel [u] are significant, as can be seen in Figure 6.15 and Figure 6.17.

Table 6.3 shows the average values of Euclidian distances from each average LPC frequency response vector to the vectors of five vowel groups. Figure 6.18 illustrates this as a graph, as well. As shown in Table 6.3, the average values of Euclidian distances of the vowels [æ], [e], [o], and [u] in one-syllable words are the closest to their own average vector. However, all of the average values of the Euclidian distances of their own average vector are increased compared to the average values in Table 6.2. Furthermore, the average values of Euclidian distances of the vowels [ɪ] and [u] are too close to distinguish one from the other. The tendency described in the end of the previous subsection seems to appear largely because vowel sounds in normal conversation (words) are more varying than when uttered alone. In other words, the shapes of the LPC frequency responses are not stable.

6.2.4 Discussion of Vowel Characterization

The results prove that the approach to vowel characterization using LPC frequency responses has a potential capability. However, to obtain more efficient results, it is necessary to establish more accurate average frequency response vectors of vowels. That includes increasing the order of LPC filter and/or the necessity of obtaining a larger number of sample segments of vowels. In addition, the feature used in the method to classify vowels was LPC frequency response, only. Combining other features, such as fundamental frequency, would improve correct classification.

One may wonder if using the LPC coefficients or the PARCOR parameters to calculate Euclidian distance would give the same results, since the LPC frequency responses are obtained from them. Therefore, the Euclidian distances of LPC coef-

ficients were calculated to see what effect they have in classifying the vowels. The method to calculate them was the same as the one with LPC frequency responses, except all eight LPC coefficients were used instead of using 20 excerpted points from 160 samples (Section 5.2).

Table 6.4 shows the average values of the Euclidian distances of LPC coefficients and Figure 6.19 illustrates them as a graph. The LPC coefficients represent a vowel by the poles of $\frac{1}{A(z)}$. The poles are directly related to the frequency response, whereas LPC coefficients are a polynomial representation of all poles and the coefficients do not have one-to-one correspondence to poles. In the table, the average value of [e] to its own LPC vector is 0.6826 and the average value of [u] to the LPC vector [e] is 0.8215. In Table 6.2, on the other hand, the values are 0.7242 and 1.4574, respectively. This is a typical example. Compared with the tables and Figures 6.12 and 6.19, the separations of the frequency responses are better than those of the LPC coefficients. Based on the small number of sample data sets compared, the frequency samples of the LPC frequency responses appear to be more suitable than the LPC coefficients in classifying vowels. This is expected from the fact that the coefficients result from polynomial expansion Equation 6.1 of the factorized estimation result, $[m_1, m_2, \cdots, m_P]$, which represents zeros and in turn the frequency response.

$$1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_P z^{-P} = (1 - m_1 z^{-1})(1 - m_2 z^{-1}) \cdots (1 - m_P z^{-1}) \quad (6.1)$$

Table 6.3 Average Euclidian Distances of Vowels in One-Syllable Words by the Female Speaker

	[æ]	[e]	[ɪ]	[o]	[u]	#†
[æ] in 'at'	1.1037	1.6295	2.2148	1.4849	2.0197	74
[e] in 'bed'	1.5415	0.7222	0.9671	1.4221	1.2682	37
[ɪ] in 'Kim'	1.8459	1.2451	0.7128	1.7324	0.7118	51
[o] in 'Tom'	1.2245	1.1437	1.4275	0.8966	1.1900	44
[u] in 'too'	1.5037	1.1161	0.6802	1.5400	0.6777	65

†: Number of sample segments

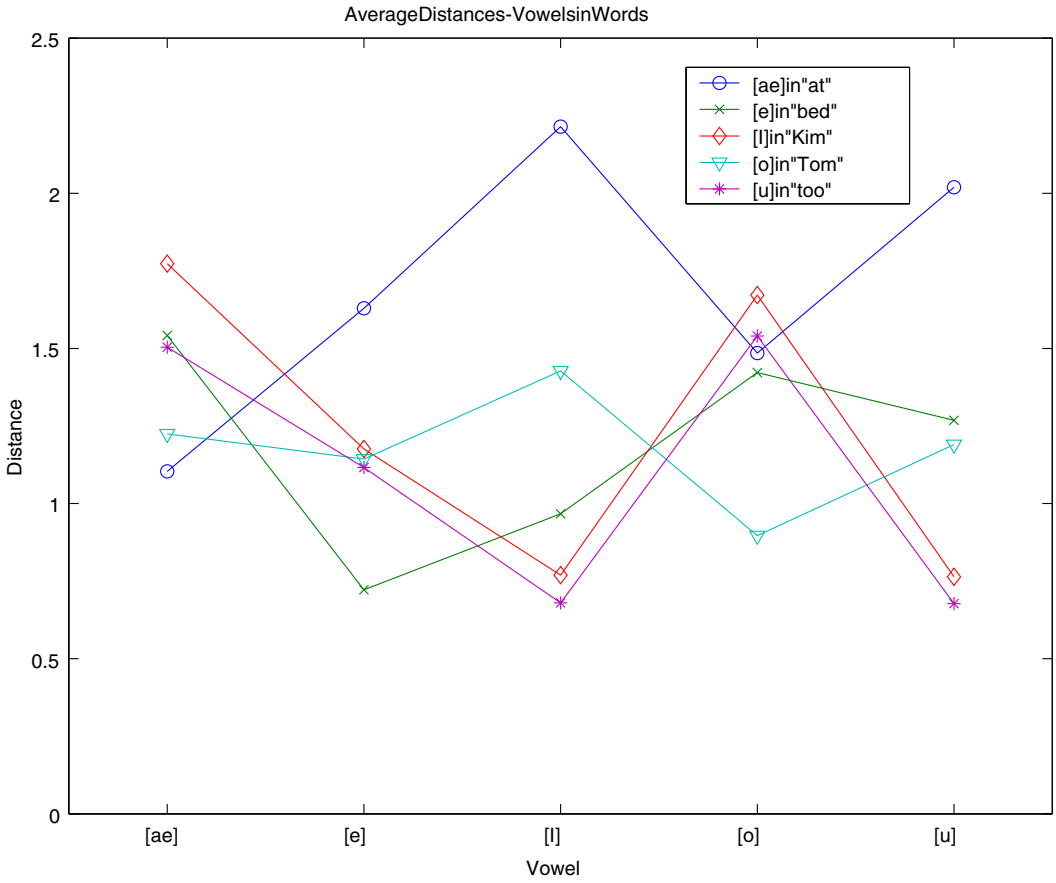


Figure 6.18 Average Euclidian distances of vowels in words

Table 6.4 Average Euclidian Distances of LPC Coefficients

	[æ]	[e]	[ɪ]	[o]	[u]	#†
[æ]	0.6684	1.2890	1.9419	1.6701	1.5893	57
[e]	1.2731	0.6826	1.2212	1.2475	1.0335	51
[ɪ]	1.8927	1.1733	0.5821	1.7425	0.9297	43
[o]	1.6339	1.1867	1.7707	0.6001	1.1445	63
[u]	1.4699	0.8215	0.7910	1.0106	0.1718	73

†: Number of sample segments

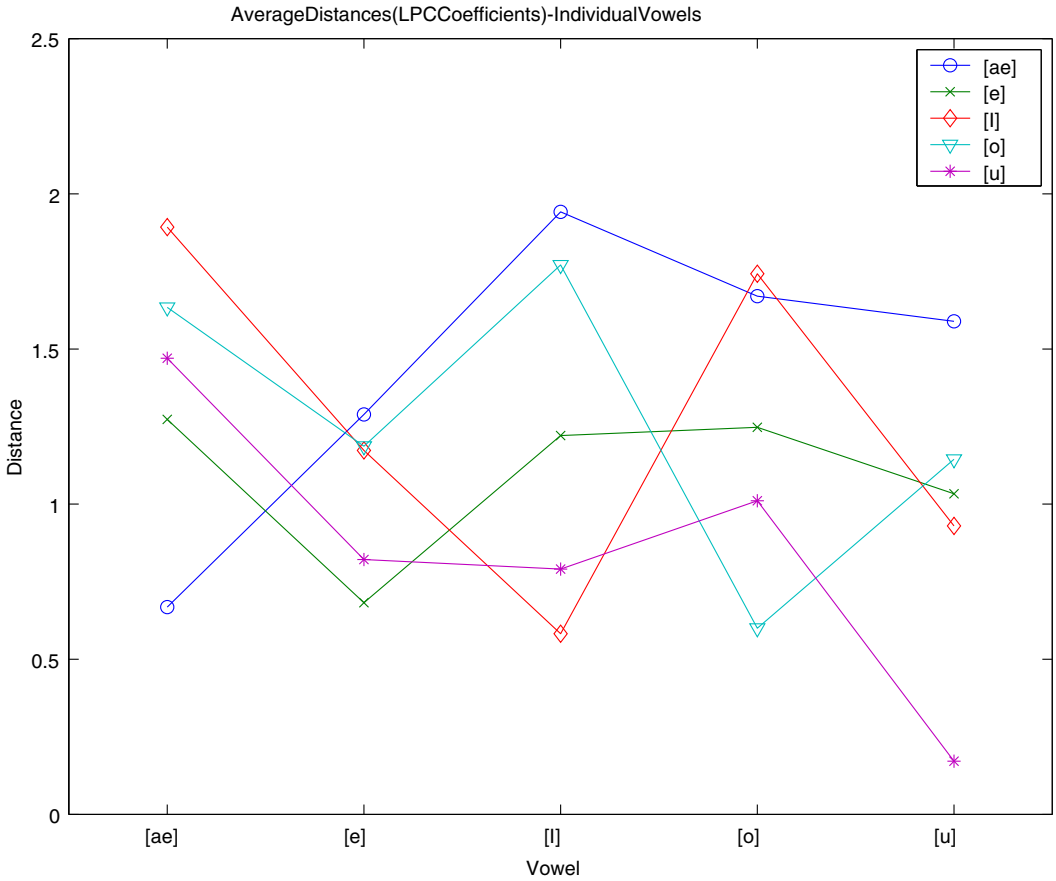


Figure 6.19 Average Euclidian distances of LPC coefficients

7. CONCLUSIONS AND FURTHER STUDY

7.1 Conclusions

Linear prediction is the core algorithm for linear predictive coding (LPC). In speech analysis and synthesis, LPC is one of the most useful methods for voice coding. Among the many voice coding systems which use LPC, the residual-excited linear predictive (RELP) vocoder was the focus of this research. The reason the RELP system was chosen was that the RELP vocoder can reproduce the synthesized speech at high quality compared with other LPC-based vocoders. Although its transmission rate is higher than the other vocoders, the RELP vocoder is rather simple and robust to implement. As well, the RELP vocoder uses the residual signal as an excitation and does not have a voiced/unvoiced decision switch or a pitch detection.

The area of speech recognition by machine is one of the most popular and complicated subjects in the current multimedia field. Its importance is recognized and expectations are rising. Demand for speech recognition is growing rapidly, even beyond the multimedia field. The potential benefit of speech recognition by machine is extraordinary in human society. However, neither the RELP vocoder nor any other vocoder has been used positively to utilize parameterized speech information to identify speech contents or to determine what word was spoken.

In this thesis, one of the LPC vocoder systems, specifically the RELP vocoder, was studied. The principle of linear prediction, including Levinson recursion and lattice filter structure, was reviewed thoroughly in order to understand the underlying principles of LPC. The prototype of the RELP vocoder system was implemented on the TI's TMS320C6711 DSP Starter Kit (DSK), which is a DSP development tool

containing one of the latest digital signal processors, TMS320C6711. The RELP vocoder system was designed using C language to run in a real-time environment. The foundation of the hardware and software of the DSK was also examined.

The results of the RELP vocoder clearly show that the quality of the synthesized speech is very good. The RELP vocoder also maintains the characteristics and the naturalness of the original speech, even when the residual signal is degraded. The system has achieved the first objective of the research, that is, to establish a prototype of the RELP vocoder system in a real-time environment.

Identifying vowel sounds is one of the most important elements in recognizing speech contents. Previously, little effort and research work had been done to use the LPC coefficients generated by the RELP vocoder for speech recognition. In the present study, the capability of classifying vowels with the RELP vocoder was studied and a method to use frequency responses of the LPC filter to classify vowels was presented.

LPC models the human vocal tract, generally, with an 8^{th} to 10^{th} order AR model. The AR model of the RELP vocoder primarily is used to synthesize speech sounds from the compressed code words. Some of the polynomial coefficients $\boldsymbol{a}$ generated by the vocoder are very sensitive to a small variation of pole positions and others are not. If one plots $\boldsymbol{a}$ in the vector space, they do not present statistically separable clusters with respect to the vowels.

This thesis has introduced an alternative method that uses the same parametric data given by the polynomial coefficients to classify vowel sounds in the form of a frequency response. The frequency response of the LPC filter was sampled in a logarithmic scale, and simple Euclidian distance was applied as a measure to classify vowels.

As discussed in Section 6.2, when a vowel is uttered alone, the distance to its average LPC frequency response vector is smaller than to the other vowels' average vectors. By examining a given vowel frequency response against all known vowels' av-

erage LPC frequency response vectors individually, one can determine to which vowel group the given vowel belongs. When a vowel is uttered with consonants, however, variances and covariances increase. In some cases, distinct differences may not be recognized among the distances to a vowel's own average vector and the distances to the other vowels' average vectors, for example, between the distances of [ɪ] in 'Kim' to its own average vector and the distances to the average vector of [u]. Overall, the results of vowel characterization did indicate an ability of the RELP vocoder to identify and classify single vowel sounds.

7.2 Suggestions for Further Study

Although the implemented RELP vocoder system satisfies the fundamental functions of the system, the data of the residual signal is not compressed. Adding a residual signal compression function to the system will improve the performance from the point of view of the data compression rate. However, except for narrow band media, considering most of today's communication media and the existence of many compression techniques, this improvement to the system may not be important.

Vowel characterization was tested using off-line data with MATLAB® programs. Transplanting the function of vowel characterization to the real-time RELP vocoder will be one of the next steps for this research. Gaining the ability to analyze a vowel sound in a spoken word using the RELP vocoder would be the beginning of the real-time speech recognition system.

To extend the presented method of classifying vowels to full-fledged speech recognition, further study is necessary in how combinations of consonants and vowels, such as a vowel at beginning of the word or a vowel in the middle of the word, will affect the frequency responses of the vowels. For instance, there is a subtle difference between [æ] in 'at' and [æ] in 'cat', since [æ] in 'cat' has a transition from a consonant to a vowel. As well, similar studies should be addressed for unvoiced sounds of consonants. The differences of vowels positioned after different consonants should also be examined, such as [be] in 'bed' or [pe] in 'pet'. These observations together with

linguistic knowledge will give us some insight in how to handle words and sentences.

A female speaker's voice was tested for vowel characterization in this thesis. Considering the results of the RELP vocoder, in which voices of two speakers were tested, the frequency responses of the same vowel sound could be considered to vary according to the speaker. Studying voices of various speakers would help the further development of the method. This study could decide whether: 1) a common database of vowels is enough; 2) individual databases are needed; 3) databases can be categorized by various combinations of gender and age groups. Vowels uttered in different emotional situations, such as sadness, anger, happiness, should be considered to some extent, as well. As discussed in Subsection 6.2.4, studying other features besides LPC frequency response will profit the accurate classification of vowels.

References

[1] N. Aoki and K. Takaya, "Wavelet Subband Filters for Mixed Excitation Linear Predictive (MELP) Vocoder," *1999 IEEE Int. Symp. ISPACS'99*, pp. 753-756, Phuket, Thailand, December 1999.

[2] J. C. Bellamy, *Digital Telephony*, John Wiley & Sons, Inc. , ISBN: 0471620564, 1991.

[3] R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, John Wiley & Sons, Inc. , ISBN: 0471207543, 2002.

[4] S. Deketelaere, O. Deroo, and T. Dutoit, "Speech Processing for Communications : What's New?," *Revue HF*, pp. 5-24, March 2001.

[5] GoldWave Inc. *http://www.goldwave.com*, September 2003.

[6] V. K. Ingle and J. G. Proakis *Digital Signal Processing using MATLAB®*, Brooks/Cole Publishing Company, ISBN: 0534371744, 2000.

[7] P. Kabal, J. L. Moncet, and C. C. Chu, "Synthesis Filter Optimization and Coding: Applications to CELP," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 147-150, New York, NY, April 1988.

[8] D. T. Magill and C. K. Un, "Speech Residual Encoding by Adaptive Delta Modulation with Hybrid Companding," *Proceedings of The National Electronics Conference*, pp. 403-408, October, 1974.

[9] A. V. McCree and T. P. Barnwell III, "A New Mixed Excitation LPC Vocoder," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 1, pp. 593-596, May 14-17, 1991.

[10] A. V. McCree and T. P. Barnwell III, "Implementation and Evaluation of a 2400 bps Mixed Excitation LPC Vocoder," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, pp. 159-162, April 27-30, 1993.

[11] A. V. McCree and T. P. Barnwell III, "Mixed Excitation LPC Vocoder Model for Bit Rate Speech Coding," *IEEE Transactions on Speech and Audio Processing*, Vol. 3, No. 4, pp. 242-250, July 1995.

[12] A. V. McCree, K. Truong, E. B. George, T. P. Barnwell, and V. Viswanathan, "A 2.4 Kbit/s MELP Coder Candidate for the New U. S. Federal Standard," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 1, pp. 200-203, May 7-10, 1996.

[13] A. V. McCree and J. C. De Martin, "A 1.7 Kbps MELP Coder with Improved Analysis and Quantization," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, pp. 593-596, Seattle, WA, May 12-15, 1998.

[14] W. S. Meisel, *Computer-oriented Approaches to Pattern Recognition*, Academic Press, 1972.

[15] N. Mikami, *DSP Programming using Code Composer Studio*, CQ Publishing Co., Ltd., Tokyo, Japan, 2002.

[16] *The Canadian Oxford Dictionary*, Oxford University Press Canada, ISBN: 019541120x, 1998.

[17] S. Prata, *C Primer Plus Fourth Edition*, Sams Publishing, ISBN: 0672322226, 2001.

[18] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Inc. , ISBN: 01321136031, 1978.

[19] R. E. Remez, P. E. Rubin, and D. B. Pisoni, "Coding of the speech spectrum in three time-varying sinusoids," *Annals of The N. Y. Academy of Sciences*, pp. 485-489, 1983.

[20] M. R. Schroeder and B. S. Atal, "Code-Excited Linear Prediction (CELP): high quality speech at very low bit rates," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 937-940, Tampa, Florida, April 1985.

[21] J. Stachurski, A. V. McCree, and V. Viswanathan, "High quality MELP coding at bit-rates around 4 kb/s," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 1, pp. 485-488, Phoenix, AZ, March 15-19, 1999.

[22] L. M. Supplee, R. P. Cohn, J. S. Collura, and A. V. McCree, "MELP: The New Federal Standard at 2400 bps," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, pp. 1591-1594, April 21-24, 1997.

[23] A. Taguchi, K. Takaya and A. S. Mehr, "Capability of Classifying Vowels with a Residual Excited Linear Prediction (RELP) Vocoder," *2003 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing PACRIM'03*, pp. 310-313, Victoria, Canada, August 2003.

[24] C. W. Therrien, *Discrete Random Signals and Statistical Signal Processing*, Prentice-Hall, Inc., ISBN: 0130225452, 1992.

[25] T. E. Tremain, "The Government Standard Linear Predictive Coding Algorithm: LPC-10," *Speech Technology*, pp. 40-49, April 1982.

[26] C. K. Un and D. T. Magill, "The Residual-Excited Linear Prediction Vocoder with Transmission Rate Below 9.6 Kbits/s," *IEEE Transactions on Communications*, Vol. COM-23, No. 12, pp. 1466-1474, 1975.

[27] T. Wang, K. Koishida, V. Cuperman, A. Gersho, and J. S. Collura, "1200 bps speech coder based on MELP," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 3, pp. 1375-1378, Istanbul, Turkey, June 5-9, 2000.

[28] Texas Instruments Japan, *Primer - All of DSP*, Gijutsu-Hyohron Co., Ltd., Tokyo, Japan, 1998.

[29] *TLC320AD535C/I Data Manual Dual Channel Voice/Data Codec (SLAS202B)*, Texas Instruments, Dallas, TX, 2000.

[30] *TMS320C6000 Optimizing Compiler User's Guide (SPRU187I)*, Texas Instruments, Dallas, TX, 2001.

[31] *TMS320C6000 CPU and Instruction Set Reference Guide (SPRU189F)*, Texas Instruments, Dallas, TX, 2000.

[32] *TMS320C6000 Peripherals Reference Guide (SPRU190D)*, Texas Instruments, Dallas, TX, 2001.

[33] *TMS320C6000 Programmer's Guide (SPRU198F)*, Texas Instruments, Dallas, TX, 2001.

[34] *TMS320C6711 DSK Help (SPRH115)*, Texas Instruments, Dallas, TX, 2001.

A. APPENDIX A
LINEAR PREDICTION

Linear predictive coding (LPC) for speech analysis and synthesis is based on linear prediction. In this appendix, the mathematical framework of linear prediction and related matters are discussed in detail. Please note that mathematical notations, equations, and figures are based on Therrien [24].

A.1 Orthogonality Principle

This section introduces the orthogonality principle. First, consider the random vector $\boldsymbol{x}$ whose components are a set of random variables x_1, x_2, ..., x_N and a related random variable y. An estimate for y is desired in the form

$$\hat{y} = \mathbf{a}^{\mathbf{T}}\boldsymbol{x} \tag{A.1}$$

where

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} \tag{A.2}$$

and coefficients a_i are weighting coefficients to minimize the mean-square error

$$E\{|y - \hat{y}|^2\} \tag{A.3}$$

If $\mathbf{a}$ is chosen as $E\{x_i\varepsilon\} = E\{\varepsilon x_i\} = 0$, $i = 1, 2, \ldots, N$, where ε is the error in estimation, $\varepsilon = y - \hat{y}$, or if the error is *orthogonal* to the observations, then $\mathbf{a}$ minimizes the mean-square error $\sigma_\varepsilon^2 = E\{|y - \hat{y}|^2\}$. This is based on the fact that

two random variables u and v are said to be orthogonal if their correlation $E\{uv\} = 0$, and is called the orthogonality principle.

Figure A.1 shows the orthogonality principle illustrated graphically in N-dimensional vector space. If $N = 2$, $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ are vectors forming a two-dimensional space and a linear combination of $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, or estimates $\hat{\boldsymbol{y}}$, is in the same two-dimensional space. Note that $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, and $\hat{\boldsymbol{y}}$ are treated as a vector in R^N, instead of the scalar variable previously defined. The random variable y needs to be treated as vector $\boldsymbol{y}$ of one-dimension higher than $\hat{\boldsymbol{y}}$, i.e. $\boldsymbol{y} \in R^{N+1}$, because $\hat{\boldsymbol{y}}$ is defined in R^N as an estimate of $\boldsymbol{y}$. $\boldsymbol{y}$ is projected $\hat{\boldsymbol{y}}$ to produce.

Since $\boldsymbol{y} = \hat{\boldsymbol{y}} + \boldsymbol{\varepsilon}$ by definition, $\boldsymbol{\varepsilon}$ can be drawn as in the figure below. It is evident that when the vector $\boldsymbol{\varepsilon}$ is orthogonal to the subspace, the length of $\boldsymbol{\varepsilon}$, or the estimation error becomes minimum.

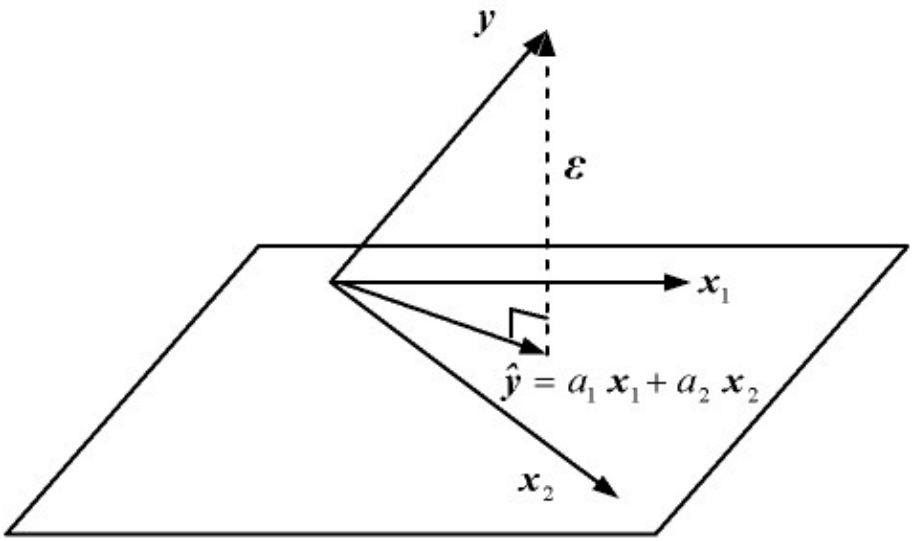


Figure A.1 Vector space interpretation of linear mean-square estimation

A.2 Linear Prediction

Linear prediction estimates the current value $x[n]$ of a random sequence x from P previous values of x to reduce redundant information from the sequence. The

estimate $\hat{x}[n]$ can be written as

$$\hat{x}[n] = -a_1 x[n-1] - a_2 x[n-2] - \cdots - a_P x[n-P] \tag{A.4}$$

and the error in the estimate is given by

$$\begin{aligned} \varepsilon[n] &= x[n] - \hat{x}[n] \\ &= x[n] + a_1 x[n-1] + a_2 x[n-2] + \cdots + a_P x[n-P] \end{aligned} \tag{A.5}$$

Equation A.4 can be also written as

$$\hat{x}[n] = -\sum_{k=1}^{P} a_k x[n-k] \tag{A.6}$$

If one defines

$$a_0 \equiv 1 \tag{A.7}$$

then the error in Equation A.5 can be expressed as

$$\begin{aligned} \varepsilon[n] &= x[n] - \hat{x}[n] \\ &= \sum_{k=0}^{P} a_k x[n-k] \end{aligned} \tag{A.8}$$

This is the output of a Finite Impulse Response (FIR) filter with impulse response $h[k] = a_k,\ k = 0, 1, 2, \ldots, P$. A problem of linear prediction is to obtain these coefficients a_k when they minimize $\varepsilon[n]$ or the mean-square error

$$\begin{aligned} \sigma_\varepsilon^2 &= E\{|\varepsilon[n]|^2\} \\ &= E\{|x[n] - \hat{x}[n]|^2\} \end{aligned} \tag{A.9}$$

σ_ε^2 is called the prediction error variance in linear prediction.

Now if one defines

$$\boldsymbol{x}[n] \equiv \begin{bmatrix} x[n-P] \\ x[n-(P-1)] \\ \vdots \\ x[n] \end{bmatrix} \tag{A.10}$$

and

$$\boldsymbol{a} \equiv \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_P \end{bmatrix} \tag{A.11}$$

Equation A.5 can be expressed as

$$\varepsilon[n] = \boldsymbol{a}^T \tilde{\boldsymbol{x}}[n] \tag{A.12}$$

where $\tilde{\boldsymbol{x}}[n]$ is the reversal of $\boldsymbol{x}[n]$,

$$\tilde{\boldsymbol{x}}[n] = \begin{bmatrix} x[n] \\ x[n-1] \\ \vdots \\ x[n-P] \end{bmatrix} \tag{A.13}$$

To obtain the coefficients which minimize the error $\varepsilon[n]$

$$E\{\tilde{\boldsymbol{x}}[n]\varepsilon[n]\} = \begin{bmatrix} \sigma_\varepsilon^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.14}$$

since the orthogonality principle states that $E\{x[n-i]\varepsilon[n]\} = 0$, $i = 1, 2, \ldots, P$ and $\sigma_\varepsilon^2 = E\{x[n]\varepsilon[n]\}$ when $i = 0$. Substituting Equation A.12 into Equation A.14

$$E\{\tilde{\boldsymbol{x}}[n]\varepsilon[n]\} = E\{\tilde{\boldsymbol{x}}[n]\tilde{\boldsymbol{x}}^T[n]\}\boldsymbol{a} = \begin{bmatrix} \sigma_\varepsilon^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Since $E\{\tilde{\boldsymbol{x}}[n]\tilde{\boldsymbol{x}}^T[n]\}$ is an autocorrelation of $\boldsymbol{x}[n]$, it can be expressed as $\tilde{\mathbf{R}}_{\boldsymbol{x}[n]}$. Then Equation A.14 can be written in form of the Normal equations as

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[P] \\ R_x[-1] & R_x[0] & \cdots & R_x[P-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[-P] & R_x[-(P-1)] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_P \end{bmatrix} = \begin{bmatrix} \sigma_\varepsilon^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.15}$$

A.3 Autoregressive (AR) Model

To obtain the parameters $\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_P$ and σ_w^2 of the AR model, solving Normal equations is done. Since the correlation function satisfies the difference equation, Equation 3.8 can be written as

$$R_x[l] + \mathrm{a}_1 R_x[l-1] + \cdots + \mathrm{a}_P R_x[l-P] = R_{wx}[l] \tag{A.16}$$

If $h[n]$ is the impulse response of the AR model, R_{wx} is

$$R_{wx}[l] = h[l] * R_w[l] = h[l] * \sigma_w^2 \delta[l] = \sigma_w^2 h[l]$$

then

$$R_{wx}[l] = \sigma_w^2 h[-l] \tag{A.17}$$

Equation A.16 can be written as

$$R_x[l] + \mathrm{a}_1 R_x[l-1] + \cdots + \mathrm{a}_P R_x[l-P] = \sigma_w^2 h[-l] \tag{A.18}$$

Since $h[n] = 0$ for $n < 0$ and from the Initial Value Theorem

$$h[0] = \lim_{z\to\infty} H(z) = \lim_{z\to\infty} \frac{1}{1 + \mathrm{a}_1 z^{-1} + \cdots + \mathrm{a}_P z^{-P}} = 1 \tag{A.19}$$

Equation A.18 can be expressed in matrix form as

$$\begin{bmatrix} R_x[0] & R_x[-1] & \cdots & R_x[-P] \\ R_x[1] & R_x[0] & \cdots & R_x[-(P-1)] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[P] & R_x[P-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ \mathrm{a}_1 \\ \vdots \\ \mathrm{a}_P \end{bmatrix} = \begin{bmatrix} \sigma_w^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.20}$$

A.4 Backward Linear Prediction

It is necessary to introduce the concept of backward linear prediction and the associated anticausal AR model so that the Levinson recursion can be developed in later section.

First, consider the random sequence and the data points $x[n-P]$ to $x[n]$. Backward linear prediction estimates a value of the oldest point $x[n-P]$ using values from

the second oldest point $x[n-(P-1)]$ to the current point $x[n]$. The estimate $\hat{x}[n-P]$ can be written as

$$\hat{x}[n-P] = -b_1 x[n-(P-1)] - b_2 x[n-(P-2)] - \cdots - b_P x[n] \tag{A.21}$$

where b_i are the backward linear predictive filter coefficients.

The error in the estimate $\varepsilon'[n-P]$ is given by

$$\begin{aligned} \varepsilon'[n-P] &= x[n-P] - \hat{x}[n-P] \\ &= x[n-P] + b_1 x[n-(P-1)] + b_2 x[n-(P-2)] + \cdots + b_P x[n] \end{aligned} \tag{A.22}$$

If the backward coefficient vector $\boldsymbol{b}$ is defined as

$$\boldsymbol{b} \equiv \begin{bmatrix} 1 \\ b_1 \\ \vdots \\ b_P \end{bmatrix} \tag{A.23}$$

then Equation A.22 is expressed as

$$\varepsilon'[n-P] = \boldsymbol{b}^T \boldsymbol{x}[n] \tag{A.24}$$

The orthogonality principle is applied to minimize the error

$$E\{\boldsymbol{x}[n]\varepsilon'[n-P]\} = \begin{bmatrix} \sigma^2_{\varepsilon'} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.25}$$

where $\sigma^2_{\varepsilon'}$ is the backward prediction error variance. From Equations A.24 and A.25

$$E\{\boldsymbol{x}[n](\boldsymbol{b}^T \boldsymbol{x}[n])\} = E\{\boldsymbol{x}[n]\boldsymbol{x}^T[n]\}\boldsymbol{b} = \begin{bmatrix} \sigma^2_{\varepsilon'} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.26}$$

This equation can be written in matrix form as

$$\begin{bmatrix} R_x[0] & R_x[-1] & \cdots & R_x[-P] \\ R_x[1] & R_x[0] & \cdots & R_x[-(P-1)] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[P] & R_x[P-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ b_1 \\ \vdots \\ b_P \end{bmatrix} = \begin{bmatrix} \sigma^2_{\varepsilon'} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.27}$$

These are the Normal equations for the backward linear prediction.

In a similar manner, consider an anticausal AR model whose difference equation

$$x[n] = -\mathrm{b}_1 x[n+1] - \mathrm{b}_2 x[n+2] - \cdots - \mathrm{b}_P x[n+P] + w'[n] \tag{A.28}$$

or

$$x[n] + \mathrm{b}_1 x[n+1] + \mathrm{b}_2 x[n+2] + \cdots + \mathrm{b}_P x[n+P] = w'[n] \tag{A.29}$$

where $w'[n]$ is a white noise process. Since the correlation function satisfies the difference equation, Equation A.29 can be expressed as

$$R_x[l] + \mathrm{b}_1 R_x[l+1] + \cdots + \mathrm{b}_P R_x[l+P] = R_{w'x}[l] \tag{A.30}$$

If $h'[n]$ is the impulse response of the anticausal AR model, $R_{w'x}[l]$ is

$$R_{w'x}[l] = \sigma^2_{w'} h[-l] \tag{A.31}$$

then Equation A.30 can be written as

$$R_x[l] + \mathrm{b}_1 R_x[l+1] + \cdots + \mathrm{b}_P R_x[l+P] = \sigma^2_{w'} h'[-l] \tag{A.32}$$

Since $h'[n] = 0$ for $n > 0$ because the system is anticausal, and applying the Initial Value Theorem, we can obtain the Yule-Walker equations for the anticausal AR model

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[P] \\ R_x[-1] & R_x[0] & \cdots & R_x[P-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[-P] & R_x[-(P-1)] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ \mathrm{b}_1 \\ \vdots \\ \mathrm{b}_P \end{bmatrix} = \begin{bmatrix} \sigma^2_{w'} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.33}$$

A.5 Levinson Recursion

To obtain the Levinson recursion, consider the forward and backward Normal equations of order p. They are:

$$\tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p)} \boldsymbol{a}_p = \begin{bmatrix} \sigma_{\varepsilon_p}^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.34}$$

or

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[p] \\ R_x[-1] & R_x[0] & \cdots & R_x[p-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[-p] & R_x[-(p-1)] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1^{(p)} \\ \vdots \\ a_p^{(p)} \end{bmatrix} = \begin{bmatrix} \sigma_{\varepsilon_p}^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.35}$$

and

$$\mathbf{R}_{\boldsymbol{x}}^{(p)} \boldsymbol{b}_p = \begin{bmatrix} \sigma_{\varepsilon_p}^{\prime 2} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.36}$$

or

$$\begin{bmatrix} R_x[0] & R_x[-1] & \cdots & R_x[-p] \\ R_x[1] & R_x[0] & \cdots & R_x[-(p-1)] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[p] & R_x[p-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ b_1^{(p)} \\ \vdots \\ b_p^{(p)} \end{bmatrix} = \begin{bmatrix} \sigma_{\varepsilon_p}^{\prime 2} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.37}$$

Using this definition

$$\mathbf{r}_p = \begin{bmatrix} R_x[1] \\ R_x[2] \\ \vdots \\ R_x[p+1] \end{bmatrix} \tag{A.38}$$

then $\tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p)}$ and $\mathbf{R}_{\boldsymbol{x}}^{(p)}$ can be partitioned as follows

$$\tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p)} = \left[\begin{array}{ccc|c} & & & \tilde{\mathbf{r}}_{p-1} \\ & \tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p-1)} & & \\ & & & \\ \hline - & - & - & - \\ \tilde{\mathbf{r}}_{p-1}^{T} & & & R_x[0] \end{array}\right] \tag{A.39}$$

$$\mathbf{R}_{\boldsymbol{x}}^{(p)} = \left[\begin{array}{ccc|c} & & & \tilde{\mathbf{r}}_{p-1} \\ & \mathbf{R}_{\boldsymbol{x}}^{(p-1)} & & \\ & & & \\ \hline - & - & - & - \\ \tilde{\mathbf{r}}_{p-1}^{T} & & & R_x[0] \end{array}\right] \tag{A.40}$$

Assume the linear prediction parameters of order $p-1$ are known and then consider the following Normal equations for the forward problem

$$\tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p)} \left[\begin{array}{c} \boldsymbol{a}_{p-1} \\ \\ - - \\ 0 \end{array}\right] = \left[\begin{array}{ccc|c} & & & \tilde{\mathbf{r}}_{p-1} \\ & \tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p-1)} & & \\ & & & \\ \hline - & - & - & - \\ \tilde{\mathbf{r}}_{p-1}^{T} & & & R_x[0] \end{array}\right] \left[\begin{array}{c} \boldsymbol{a}_{p-1} \\ \\ - - \\ 0 \end{array}\right] = \left[\begin{array}{c} \sigma_{\varepsilon_{p-1}}^{2} \\ 0 \\ \vdots \\ \Delta_p \end{array}\right] \tag{A.41}$$

where Δ_p is

$$\Delta_p = \tilde{\mathbf{r}}_{p-1}^{T} \boldsymbol{a}_{p-1} = \mathbf{r}_{p-1}^{T} \tilde{\boldsymbol{a}}_{p-1} \tag{A.42}$$

Further consider corresponding Normal equations for the backward problem

$$\mathbf{R}_{\boldsymbol{x}}^{(p)} \left[\begin{array}{c} \boldsymbol{b}_{p-1} \\ \\ - - \\ 0 \end{array}\right] = \left[\begin{array}{ccc|c} & & & \tilde{\mathbf{r}}_{p-1} \\ & \mathbf{R}_{\boldsymbol{x}}^{(p-1)} & & \\ & & & \\ \hline - & - & - & - \\ \tilde{\mathbf{r}}_{p-1}^{T} & & & R_x[0] \end{array}\right] \left[\begin{array}{c} \boldsymbol{b}_{p-1} \\ \\ - - \\ 0 \end{array}\right] = \left[\begin{array}{c} \sigma_{\varepsilon_{p-1}}^{\prime 2} \\ 0 \\ \vdots \\ \Delta_p^{\prime} \end{array}\right] \tag{A.43}$$

where $\Delta_p^{\prime}$ is

$$\Delta_p^{\prime} = \tilde{\mathbf{r}}_{p-1}^{T} \boldsymbol{b}_{p-1} = \mathbf{r}_{p-1}^{T} \tilde{\boldsymbol{b}}_{p-1} \tag{A.44}$$

Reverse all of the terms in Equation A.43, multiplied by a constant c_1, and add this

result to Equation A.41

$$\tilde{\mathbf{R}}_{\boldsymbol{x}}^{(p)} \left[\begin{bmatrix} \boldsymbol{a}_{p-1} \\ \\ -- \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 0 \\ -- \\ \\ \tilde{\boldsymbol{b}}_{p-1} \end{bmatrix} \right] = \begin{bmatrix} \sigma^2_{\varepsilon_{p-1}} \\ 0 \\ \vdots \\ \Delta_p \end{bmatrix} + c_1 \begin{bmatrix} \Delta_p^{'} \\ 0 \\ \vdots \\ \sigma^{'2}_{\varepsilon_{p-1}} \end{bmatrix} \tag{A.45}$$

From this result and Equation A.34, when c_1 satisfies

$$\begin{bmatrix} \sigma^2_{\varepsilon_{p-1}} \\ 0 \\ \vdots \\ \Delta_p \end{bmatrix} + c_1 \begin{bmatrix} \Delta_p^{'} \\ 0 \\ \vdots \\ \sigma^{'2}_{\varepsilon_{p-1}} \end{bmatrix} = \begin{bmatrix} \sigma^2_{\varepsilon_p} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.46}$$

then c_1 also satisfies

$$\begin{bmatrix} \boldsymbol{a}_{p-1} \\ \\ -- \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 0 \\ -- \\ \\ \tilde{\boldsymbol{b}}_{p-1} \end{bmatrix} = \boldsymbol{a}_p \tag{A.47}$$

From Equation A.46, the following equations can be derived

$$\sigma^2_{\varepsilon_{p-1}} + c_1 \Delta_p^{'} = \sigma^2_{\varepsilon_p} \tag{A.48}$$

and

$$c_1 = -\Delta_p / \sigma^{'2}_{\varepsilon_{p-1}} \tag{A.49}$$

Similarly, in case of the backward problem, it can be stated that these results, using a constant c_2

$$\sigma^{'2}_{\varepsilon_{p-1}} + c_2 \Delta_p = \sigma^{'2}_{\varepsilon_p} \tag{A.50}$$

and

$$c_2 = -\Delta_p^{'} / \sigma^2_{\varepsilon_{p-1}} \tag{A.51}$$

c_2 must satisfy

$$\begin{bmatrix} \boldsymbol{b}_{p-1} \\ \\ -- \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ -- \\ \\ \tilde{\boldsymbol{a}}_{p-1} \end{bmatrix} = \boldsymbol{b}_p \tag{A.52}$$

As a result, the recursion process consists of Equations A.42, A.44, A.47, A.48, A.50, and A.52. Now let $\gamma_p = -c_1$ and $\gamma_p^{'} = c_2$. These parameters γ_p $\gamma_p^{'}$ are called the forward and backward reflection coefficients. They are also known as partial correlation, or PARCOR, coefficients. The initial conditions of the recursion are: $\mathbf{r}_0 = R_x[1]$, $\boldsymbol{a}_0 = \boldsymbol{b}_0 = 1$, and $\sigma_{\varepsilon_0}^2 = \sigma_{\varepsilon_0}^{'2} = R_x[0]$. Substituting γ_p and $\gamma_p^{'}$ for c_1 and c_2, $\sigma_{\varepsilon_p}^2$ and $\sigma_{\varepsilon_p}^{'2}$ can be expressed as

$$\sigma_{\varepsilon_p}^2 = (1 - \gamma_p \gamma_p^{'})\sigma_{\varepsilon_{p-1}}^2 \tag{A.53}$$

and

$$\sigma_{\varepsilon_p}^{'2} = (1 - \gamma_p^{'} \gamma_p)\sigma_{\varepsilon_{p-1}}^{'2} \tag{A.54}$$

It is convenient to simplify the relationship between the forward and backward forms of the Levinson recursion above. Since $\tilde{\mathbf{R}}_{\boldsymbol{x}} = \mathbf{R}_{\boldsymbol{x}}$, it is known that $\boldsymbol{b}_{p-1} = \boldsymbol{a}_{p-1}$ and $\sigma_{\varepsilon_{p-1}}^{'2} = \sigma_{\varepsilon_{p-1}}^2$. Equations A.42 and A.44 are written as

$$\Delta_p^{'} = \mathbf{r}_{p-1}^T \tilde{\boldsymbol{b}}_{p-1} = \mathbf{r}_{p-1}^T \tilde{\boldsymbol{a}}_{p-1} = \Delta_p \tag{A.55}$$

Apply this result to Equations A.49 and A.51

$$\gamma_p^{'} = \frac{\Delta_p^{'}}{\sigma_{\varepsilon_{p-1}}^2} = \frac{\Delta_p}{\sigma_{\varepsilon_{p-1}}^{'2}} = \gamma_p \tag{A.56}$$

Therefore, the essential equations to compute the recursion process can be rewritten as

$$\gamma_p = \frac{\mathbf{r}_{p-1}^T \tilde{\boldsymbol{a}}_{p-1}}{\sigma_{\varepsilon_{p-1}}^2} \tag{A.57}$$

$$\boldsymbol{a}_p = \begin{bmatrix} \boldsymbol{a}_{p-1} \\ \\ -- \\ 0 \end{bmatrix} - \gamma_p \begin{bmatrix} 0 \\ -- \\ \\ \tilde{\boldsymbol{a}}_{p-1} \end{bmatrix} \tag{A.58}$$

$$\sigma_{\varepsilon_p}^2 = (1 - |\gamma_p|^2)\sigma_{\varepsilon_{p-1}}^2 \tag{A.59}$$

where $p = 1, 2, \ldots, P$ and initial conditions are

$$\boldsymbol{a}_0 = [1]; \quad \mathbf{r}_0 = R_x[1]; \quad \sigma_{\varepsilon_0}^2 = R_x[0] \tag{A.60}$$

From Equations A.58 and A.60, the following facts can be derived;

$$a_p^{(p)} = -\gamma_p \tag{A.61}$$

and

$$0 \leq |\gamma_p| < 1 \tag{A.62}$$

B. APPENDIX B
C PROGRAMS

B.1 Main Program of RELP Vocoder (protoxx.c)

```
/* ====================== proto11.c -- December 11, 2002 ====================== */
/* ==================== This is a main file of the project ==================== */
/*                                                                              */
/*  proto11.c -- calculates LPC coefficients, PARCOR parameters (reflection     */
/*               coefficients), gain, and lattice-filtered residual signals     */
/*               from real-time voice signals for up to 10 seconds and          */
/*               Linear Prediction order up to 10.                              */
/*               It also produces the synthesized voice signal from the         */
/*               PARCOR parameters and their residual signals.                  */
/*                                                                              */
/*  Other files necessary to build this program:                                */
/*                                                                              */
/*   >SOURCE FILES: a_corr.c      -- autocorrelation function                   */
/*                  tans.c        -- transfers data from a buffer to another    */
/*                  LPC_encode.c  -- LPC & PARCOR coefficients and gain         */
/*                  fwd_lattice.c -- forward lattice filter                     */
/*                  inv_lattice.c -- inverse lattice filter                     */
/*   >LINKER COMMAND FILE: proto.cmd                                            */
/*   >HEADER FILE:         proto.h                                              */
/*   >SUPPORT FILES:       c6xdskinit.c                                         */
/*                         vectors_11.asm                                       */
/*                         rts6701.lib @ c:\ti\c6000\cgtools\lib                */
/*                         c6xdsk.h, c6xdskinit.h, c6xinterrupts.h, c6x.h       */
/*         and also        fastrts67x.lib -- from TI's web site                 */
/*                                                                              */
/*  e[n]:     residual signal, n=0,...,BUFFER_2_SIZE-1                          */
/*  a[m]:     LPC coefficients, m=0,...,ORDER                                   */
/*  gamma[m]: PARCOR coefficients, m=0,...,ORDER-1                              */
/*  gn[1]:    gain                                                              */
/*                                                                              */
/********************************************************************************/

#include <stdio.h>
#include <stdlib.h>                /* for calloc                                */
```

```
#include "proto.h"

#pragma DATA_SECTION(indx, "delays");
#pragma DATA_SECTION(buffer_1, "delays");
#pragma DATA_SECTION(buffer_2, "delays");
#pragma DATA_SECTION(x, "delays");
#pragma DATA_SECTION(end_buf, "delays");

#pragma DATA_SECTION(lpc_coeff, "lpc_data");
#pragma DATA_SECTION(parcor, "lpc_data");
#pragma DATA_SECTION(residue, "lpc_data");
#pragma DATA_SECTION(gain, "lpc_data");
#pragma DATA_SECTION(in_buff, "lpc_data");
#pragma DATA_SECTION(out_buff, "lpc_data");
#pragma DATA_SECTION(end_mem, "lpc_data");

int indx;                           /* temporary space for the pointer Ptr_dly  */
float buffer_1[BUFFER_1_SIZE];      /* buffer_1: ring buffer storing input data */
float buffer_2[BUFFER_2_SIZE * 2];  /* buffer_2: buffer for processing data     */
float x[BUFFER_2_SIZE];             /* x: buffer for synthesized data           */
int end_buf = 0;                    /* dummy for checking memory                */

int i = 0;                          /* counter for input samples                */
static int s = 340;                 /* s & t: parameters to store synthesized   */
int t;                              /*  data in out_buff[]                      */
int flag = 0;                       /* to call LPC processing functions         */
                                    /* flag = 0: no call                        */
                                    /* flag = 1: call the functions             */
int status = 0;                     /* status = 0: the first 180 samples        */
                                    /* status = 1: after the first 180 samples  */
                                    /* status = 2: call decoder functions       */
unsigned int * Ptr_dly;             /* pointer for the ring buffer "buffer_1"   */
                                    /* points to the last address               */
unsigned int * Ptr_lat;             /* pointer for dly[] of fwd_lattice         */
int count = 0;

float lpc_coeff[(ORDER + 1) * 50 * REC_TIME]; /* 20ms sampling period is        */
float parcor[ORDER * 50 * REC_TIME];          /* 50 periods in 1 second.        */
float residue[BUFFER_2_SIZE * 50 * REC_TIME]; /* 200 samples/period             */
float gain[50 * REC_TIME];                    /* 1 sample/period                */
float in_buff[(BUFFER_2_SIZE - 40) * 50 * REC_TIME];
float out_buff[(BUFFER_2_SIZE - 40) * 50 * REC_TIME];
                                              /* 160 samples/period             */
float end_mem = 0;                            /* dummy for checking memory      */
```

```
void main()
{
 int T = 0;
 float * a, * gamma, * e;
 float gn[1] = {0.0};
 *Ptr_dly = 20;

/* ===================== initialize buffers (optional) ======================= */
 for (T = 0; T < BUFFER_1_SIZE; T++)
    buffer_1[T] = 0;
 for (T = 0; T < BUFFER_2_SIZE*2; T++)
    buffer_2[T] = 0;
 for (T = 0; T < (ORDER + 1) * 50 * REC_TIME; T++)
    lpc_coeff[T] = 0;
 for (T = 0; T < ORDER * 50 * REC_TIME; T++)
    parcor[T] = 0;
 for (T = 0; T < BUFFER_2_SIZE * 50 * REC_TIME; T++)
    residue[T] = 0;
 for (T = 0; T < 50 * REC_TIME; T++)
    gain[T] = 0;
 for (T = 0; T < (BUFFER_2_SIZE - 40) * 50 * REC_TIME; T++)
 {
    out_buff[T] = 0;
    in_buff[T] = 0;
 }

/* ============================ main routine ================================= */
 comm_intr();                              /* init DSK, codec, McBSP           */
 while(1)                                  /* infinite loop                    */
 {
    a = (float *)calloc(ORDER+1, sizeof(float));
    gamma = (float *)calloc(ORDER, sizeof(float));
    e = (float *)calloc(BUFFER_2_SIZE, sizeof(float));

    while(flag == 1)
    {
        LPC_encode(buffer_2, a, gamma, gn); /* obtain the coefficients         */
        fwd_lattice(buffer_2, gamma, e);    /* obtain residue signal           */
        encode(a, e, gamma, gn);            /* encode into files               */
        inv_lattice(e, gamma, x);           /* reproduce synthesized signal    */
        for (t = 0; t < 160; t++)           /* store the signal in out_buff[]  */
            out_buff[s + t] = x[t+20];
        s = s + 160;
        flag = 0;                           /* set flag to zero                */
        count++;
```

```
        if (count == (50 * REC_TIME)-1)
        {
            flag = 2;
            status = 2;
        }
    }
    free(a);
    free(gamma);
    free(e);
 }
}
/* ======================= the end of main routine ============================ */

/* ======================= interrupt service routine ========================== */
interrupt void c_int11()
{
 short sample_data;
 static int y = 0;
 static int z = 0;

 sample_data = input_sample();               /* new input data                  */
 buffer_1[*Ptr_dly] = (float)sample_data;    /* store input data into buffer    */
 if (y < ((BUFFER_2_SIZE - 40) * 50 * REC_TIME)+20)  /* store whole input data  */
    in_buff[y++] = (float)sample_data;               /* into in_buff            */

/*                                                                              */
/*      The following diagram illustrates how the buffer_1, incoming            */
/*      samples and  each processing period are related to                      */
/*                                                                              */
/* samples          0                    160     180            320     340    */
/* Prd1     |.......|======================|-------|              :       :     */
/* Prd2     :       :              |-------|======================|-------|     */
/* Prd3     :       :              :       :       :      |-------|===========  */
/*          :       :              :       :       :      :       :       :     */
/*        b_1[0]    :          b_1[160]    :    b_1[200]  :     b_1[64]   :     */
/*               b_1[20]                b_1[180]       b_1[44]         b_1[84]  */
/*                                                                              */
/*     @ The first 20 spaces of buffer_1 in the first period are zeros          */
/*     @ There are 20 samples for overlap at the front and the rear of each     */
/*       period.                                                                */
/*       [=====]: needed samples, [.....]:  zeros, [-----]: overlap             */

 i++;  /* increment counter */

/* ======================= The first 180 samples ============================== */
```

```
 if (status == 0 && i == 180) /* the very first processing period */
 {
    trans(Ptr_dly, buffer_1, buffer_2);  /* transfer data into buffer_2         */
    flag = 1;                            /* call the functions                 */
    status = 1;                          /* change status to 1                 */
    i = 0;                               /* reset i to 0                       */
 }
/* ================== The end of the first 180 samples ======================= */

/* ======================= After the 180th sample ============================== */
 else if (status == 1 && i == 160)       /* process every 160 samples (20ms)    */
 {
    trans(Ptr_dly, buffer_1, buffer_2);  /* transfer data into buffer_2         */
    flag = 1;                            /* call the functions                 */
    i = 0;                               /* reset i to 0                       */
 }

 if (z < ((BUFFER_2_SIZE - 40) * 50 * REC_TIME))
     output_sample((int)out_buff[z++]);

/* increment pointer Ptr_dly to the last address of ring buffer                 */
 *Ptr_dly = (*Ptr_dly + 1) % BUFFER_1_SIZE;

 return;                                 /* return from ISR                     */
}
/* ================= the end of interrupt service routine ==================== */

void encode(float a[], float e[], float gamma[], float g[])
{
    short j;
    static int k = 0, m = 0, p = 0, q = 0;

    for (j = 0; j < ORDER + 1; j++)
        lpc_coeff[k + j] = a[j];
    k = k + ORDER + 1;
    for (j = 0; j < ORDER; j++)
        parcor[m + j] = gamma[j];
    m = m + ORDER;
    for (j = 0; j < BUFFER_2_SIZE; j++)
        residue[p + j] = e[j];
    p = p + BUFFER_2_SIZE;
    gain[q] = g[0];
    q = q + 1;
}
```

B.2 Function trans (trans.c)

```
/*      trans.c -- "trans" function transfers latest 200 samples in          */
/*                 ring buffer "buff_1" to processing buffer "buff_2".       */
/*                 The first half of the "buff_2" are all zeros to           */
/*                 calculate autocorralation of samples because it           */
/*                 is requirement of the function "a_corr".                  */
/*                                                                           */
/*     [0]                                      [BUFFER_2_SIZE*2-1]          */
/*      |<--- BUFFER_2_SIZE --->|<--- BUFFER_2_SIZE --->|                     */
/*      |------- zeros ---------|------- samples -------|                     */
/*                              OLDEST <---------- NEWEST                    */
/*                                                                           */

#include "proto.h"

void trans(unsigned int * ptr, float buff_1[], float buff_2[])
{
    short j;
    for(j = 0; j < BUFFER_2_SIZE; j++)
        buff_2[(BUFFER_2_SIZE*2 - 1) - j] = buff_1[(*ptr - j) % BUFFER_1_SIZE];
}
```

B.3 Function LPC_encode (LPC_encode.c)

```
/*  LPC_encode.c -- to calculate Linear Predictive Coding coefficients,      */
/*                  PARCOR parameters and gain of input voice signals        */
/*                  using Levinson Recursion Algorithm                       */
/*                  For farther reference, see Therrien page 428             */

/*  s[]:        input signal                                                 */
/*  a[]:        LPC coefficients                                             */
/*  gamma[]:    PARCOR coefficients (reflection coefficients)                */
/*  g:          gain                                                         */
/*  Rx[]:       allay of an autocorrelation                                  */
/*  r[]:        r[p] is defined as transpose of [Rx[1] Rx[2] ... Rx[p]]      */
/*  rev_a[]:    reversal of a[]                                              */

#include <math.h>              /* for "sqrt/sqrtdp" from FastRTS Library     */
#include <ieeed.h>             /* for "_divd" from FastRTS Library           */
#include "proto.h"

void LPC_encode(float s[], float a[], float gamma[], float gain[])
{
    float Rx[BUFFER_2_SIZE], r[ORDER], rev_a[ORDER+1];
    float power;
```

```
    float ra=0;
    int m, n, p;

/* ================     Autocorralation of input signal     ================ */
    a_corr(Rx, s, BUFFER_2_SIZE, BUFFER_2_SIZE);

/* ====================     Levinson Recursion          ==================== */

/* ====   initial conditions & initialization   ==== */
    a[0] = 1;
    rev_a[0] = 0;
    power = Rx[0];
    for (m = 1; m <= ORDER; m++)
    {
        a[m] = rev_a[m] = 0;
        r[m-1] = Rx[m];
        gamma[m-1] = 0;
    }
/* ================================================ */

    for (n=0; n<ORDER; n++)
    {
        for (p=0; p<=n; p++)
            ra += (double)r[p]*(double)a[n-p];
/*                                          | a[n-1] |                   */
/*                                          | a[n-2] |                   */
/*      ra(n) = [r[0] r[1] ... r[n-2] r[n-1]] * |  :     |                   */
/*                                          | a[1]   |                   */
/*                                          | a[0]   |                   */

        gamma[n] = (double)ra/(double)power;     /* "actual" gamma[n] is     */
                                                 /* gamma[n+1]               */
        for(p=0; p<=n; p++)                /* make a reversal matrix of a[] */
            rev_a[p] = a[n-p];
        for(p=1; p<=n+1; p++)
            a[p] = a[p]-((double)gamma[n]*(double)rev_a[p-1]);

/*            |a_p-1[0]  |          |    0     |                            */
/*            |a_p-1[1]  |          |----------|                            */
/*      a_p = |    :     | - gamma_p|a_p-1[p-1]|                            */
/*            |a_p-1[p-1]|          |    :     |                            */
/*            |----------|          |a_p-1[0]  |                            */
/*            |    0     |          |a_p-1[0]  |                            */
/*                                          @ a[0] is always ONE        */
```

```
        power = (double)power*(1 - ((double)gamma[n]*(double)gamma[n]));
        ra = 0;
    }
    gain[0] = sqrtdp((double)power);            /* from FastRTS Library     */
}
```

B.4 Function a_corr (a_corr.c)

```
/*  NAME                                                                    */
/*      a_corr                                                              */
/*                                                                          */
/*  USAGE                                                                   */
/*      This routine has the following C prototype:                         */
/*                                                                          */
/*      void a_corr                                                         */
/*      (                                                                   */
/*          float       r[],                                                */
/*          float       x[],                                                */
/*          int         nx,                                                 */
/*          int         nr                                                  */
/*      )                                                                   */
/*                                                                          */
/*      r[nr]   : Output array.                                             */
/*      x[nr+nx]: Input  array.                                             */
/*                Must be word aligned.                                     */
/*      nx      : Length of autocorrelation.                                */
/*      nr      : Number of lags.                                           */
/*                                                                          */
/*  DESCRIPTION                                                             */
/*      This routine performs an autocorrelation of an input vector         */
/*      x. The length of the autocorrelation is nx samples. Since nr        */
/*      such autocorrelations are performed, input vector x needs to be     */
/*      of length nx + nr. This produces nr output results which are        */
/*      stored in an output array r.                                        */
/*                                                                          */
/*      The following diagram illustrates how the correlations are          */
/*      obtained.                                                           */
/*                                                                          */
/*      Example for nr=8, nx=24:                                            */
/*      0       nr                  nx+nr-1                                 */
/*      |-------|---------------------|  <- x[]                            */
/*      |       |---------------------|  -> r[0]                           */
/*      |      |---------------------|   -> r[1]                           */
/*      |     |---------------------|    -> r[2]                           */
/*      |    |---------------------|     -> r[3]                           */
```

```
/*      |   |----------------------|      -> r[4]                          */
/*      |  |----------------------|       -> r[5]                          */
/*      | |----------------------|        -> r[6]                          */
/*                                                                         */
/*      Note that x[0] is never used, but is required for padding to make  */
/*      x[nr] word aligned.                                                */
/*                                                                         */
/*    *** THIS CODE IS A MODIFIED VERSION OF TI's DSPLIB FUNCTION ***      */
/*    *** "DSP_autocor".                                          ***      */
/*                                                                         */

void a_corr(float r[], float x[], int nx, int nr)
 {
    int i,k;
    double sum;
    for (i = 0; i < nr; i++)
    {
        sum = 0;
        for (k = nr; k < nx+nr; k++)
            sum += (double)x[k] * (double)x[k-i];
        r[i] = (float)sum;
    }
 }
```

B.5 Function `fwd_lattice` (`fwd_lattice.c`)

```
/*  fwd_lattice.c -- to calculate a prediction error signal through        */
/*                   a prediction error FIR lattice filter using           */
/*                   PARCOR coefficients                                   */

/*  s[]:        input signal                                               */
/*  gamma[]:    PARCOR coefficients (reflection coefficients)              */
/*  e[]:        prediction error or residual signal of an input            */

#include "proto.h"

void fwd_lattice(float s[], float gamma[], float e[])
{
    int m, n, p;
    float dum;                      /* dummy                              */
    float dly[BUFFER_2_SIZE];       /* delay sequence & ring buffer       */
    extern unsigned int * Ptr_lat;  /* pointer for dly[]                  */
    *Ptr_lat = 0;                   /* set initial value of Ptr_lat       */

/* ========================================================================= */
```

```
/*     store values of input sequence s[] into e[]                          */
/*     & store values of one sample delay of s[] into dly[]                 */
/*     the first value of delay, dly[0] must be always zero                 */
/*                                                                          */
/* ### size of s[] is BUFFER_2_SIZE times two, first half is zeros ###      */

    for (m = 0; m < BUFFER_2_SIZE-1; m++)
    {
        e[m] = s[BUFFER_2_SIZE + m];
        dly[*Ptr_lat + m + 1] = s[BUFFER_2_SIZE + m];
    }
    e[BUFFER_2_SIZE-1] = s[2*BUFFER_2_SIZE-1];
    dly[*Ptr_lat] = 0;
/* ======================================================================== */

/* ===================== Lattice Filter Loop ============================== */
/*                                                                          */
/*        |E_p[n] |   |   1     -gamma_p|   |E_p-1[n]   |                  */
/*        |       | = |                 | * |           |                  */
/*        |Eb_p[n]|   |-gamma_p    1    |   |Eb_p-1[n-1]|                  */
/*                                                                          */
/*                                                                          */
/*                    E_1[n]       E_2[n]                                   */
/*   x[n] *--+-->-+-->-+--->---+-->-+--->------+-->-+------> E_p[n]         */
/*           |     \  /         \  /           \  /                         */
/*           |      \/ -g_1      \/ -g_2  ...    \/ -g_p                    */
/*           |      /\ -g_1      /\ -g_2         /\ -g_p                    */
/*           |     /  \         /  \           /  \                         */
/*           +-[]-+-->-+---[]--+-->-+--[]------+-->-+------> Eb_p[n]        */
/*            1/z         1/z          1/z                                 */
/*                    Eb_1[n]      Eb_2[n]                                  */
/*                                                                          */
/*     x[n]:    input sequence                                              */
/*     E_p[n]:  prediction error of order p - residue sequence              */
/*     Eb_p[n]: backward prediction error of order p - delay sequence       */

    for (n = 0; n <ORDER; n++)
    {
        for (m = 0; m < BUFFER_2_SIZE; m++)
        {
            dum = e[m];
            p = (*Ptr_lat + m) % BUFFER_2_SIZE;        /* for dly[]         */
            e[m] = dum - (double)gamma[n]*(double)dly[p];
            dly[p] = -1*(double)gamma[n]*(double)dum + dly[p];
        }
```

```
        *Ptr_lat = (*Ptr_lat + BUFFER_2_SIZE - 1) % BUFFER_2_SIZE;
        dly[*Ptr_lat] = 0;
    }
}
```

B.6 Function inv_lattice (inv_lattice.c)

```
/*  inv_lattice.c -- to synthesize random signal through an inverse          */
/*                   lattice filter using PARCOR coefficients                */
/*                   residual signal is as an input                          */

/*  x[]:        output signal (synthesized random signal)                    */
/*  gamma[]:    PARCOR coefficients (reflection coefficients)                */
/*  residue[]:  prediction error or residual signal of an input              */

#include "proto.h"

void inv_lattice(float residue[], float gamma[], float x[])
{
/*  f[]: buffer to store p-th order prediction errors                        */
/*  g[]: buffer to store p-th order backward prediction errors               */
    int m, n;
    short fx_res;
    float f[ORDER+1], g[ORDER+1];

    for(m = 0; m <= ORDER; m++)     /* initialize f[] and g[] to zeros       */
    {
        f[m] = 0;
        g[m] = 0;
    }

/* ================ Inverse Lattice Filter Loop ============================ */
/*                                                                          */
/*       E_p-1[n] =            E_p[n]   + gamma_p * Eb_p-1[n-1]             */
/*       Eb_p[n]  = -gamma_p * E_p-1[n] +           Eb_p-1[n-1]             */
/*                                                                          */
/* w[n] =           E_p-1[n]           E_1[n]             E_0[n] =          */
/* E_p[n] *-->-+-->-*--->---------+-->-*--->----+-->-*--->---+--> x[n]      */
/*             \  /              \  /          \  /        |               */
/*           g_p \/       ...   g_2 \/        g_1 \/         |              */
/*          -g_p /\            -g_2 /\       -g_1 /\         |              */
/*              /  \   1/z         /  \   1/z    /  \   1/z  |              */
/*             +-<--*---<---------+-<--*---<----+--<-*---<---+              */
/*            Eb_p[n]            Eb_1[n]      Eb_1[n]                       */
/*                                                                          */
```

```
/*     w[n]:    residual signal (input sequence)                              */
/*     x[n]:    output sequence                                               */
/*     E_p[n]:  prediction error of order p - residue sequence                */
/*     Eb_p[n]: backward prediction error of order p - delay sequence         */

    for(m = 0; m < BUFFER_2_SIZE; m++)
    {
        fx_res = (short)residue[m];  // fixed point version
        fx_res = fx_res * R_sift_2;
        fx_res = fx_res * L_sift_2;
        f[ORDER] = (float)fx_res;

        /*f[ORDER] = residue[m];*/
        for(n = ORDER; n >= 1; n--)
        {
            f[n-1] = f[n] + (double)gamma[n-1]*(double)g[n-1];
            g[n] = -1*(double)gamma[n-1]*(double)f[n-1] + g[n-1];
        }
        g[0] = f[0];
        x[m] = f[0];
    }
}
```

B.7 Header file proto (proto.h)

```
/* proto.h -- constants and declarations for protoXX.c                        */

#define BUFFER_1_SIZE 512       /* buffer size for input data                 */
#define BUFFER_2_SIZE 200       /* buffer size for processing data            */
                                /* 2.5m(overlap) + 20ms + 2.5ms(overlap)      */
#define ORDER 8                 /* Linear Prediction order up to 10           */
#define REC_TIME 2              /* recording time (in second up to 10)        */

#define R_sift_0 1
#define R_sift_1 0.5
#define R_sift_2 0.25
#define R_sift_3 0.125
#define R_sift_4 0.0625
#define R_sift_5 0.03125
#define R_sift_6 0.015625
#define R_sift_7 0.0078125
#define R_sift_8 0.00390625
#define R_sift_9 0.001953125
#define R_sift_10 0.0009765625
#define R_sift_11 0.00048828125
```

```
#define R_sift_12 0.000244140625
#define R_sift_13 0.0001220703125
#define R_sift_14 0.00006103515625

#define L_sift_0 1
#define L_sift_1 2
#define L_sift_2 4
#define L_sift_3 8
#define L_sift_4 16
#define L_sift_5 32
#define L_sift_6 64
#define L_sift_7 128
#define L_sift_8 256
#define L_sift_9 512
#define L_sift_10 1024
#define L_sift_11 2048
#define L_sift_12 4096
#define L_sift_13 8192
#define L_sift_14 16384

//#define REC 250                /* # of recording periods 50 periods/sec   */

/* prototypes */
void trans(unsigned int * ptr, float buff_1[], float buff_2[]);
void a_corr(float r[], float x[], int nx, int nr);
void LPC_encode(float s[], float a[], float gamma[], float gain[]);
void fwd_lattice(float s[], float gamma[], float e[]);
void encode(float a[], float e[], float gamma[], float g[]);
//void decode(float gamma[], float e[]);
void inv_lattice(float residue[], float gamma[], float x[]);
```