The Music Really Speaks to Me

Real-Time Musical Vocoder

Final Report

18-551 Fall 2009

Group 6

Si (Laura) Cai (scai@andrew.cmu.edu)

Pritish Gandhi (pgandhi@andrew.cmu.edu)

Chris Guida (cguida@andrew.cmu.edu)

TABLE OF CONTENTS

1. ABSTRACT	3
2. THE PROBLEM	3
3. NOVELTY	4
4. SOLUTION	5
5. WHAT IS LINEAR PREDICTIVE CODING OR LPC	6
6. THE LEVINSON DURBIN ALGORITHM TO CALCULATE LPC	9
6.1 THE LEVINSON-DURBIN RECURSIVE ALGORITHM	12
6.2 COMPARISON OF LEVINSON-DURBIN ALGORITHM WITH OTHER ALGORITHMS	13
7. PROCEDURE AND ALGORITHM	15
8. SPECIFICATIONS	19
9. FUNCTION PROFILE TIMES, SPEEDS, LATENCY & OPTIMIZATION	20
10. ERRORS PROBLEMS & THEIR SOLUTIONS	22
11. CODE INFORMATION	24
12. DEMONSTRATION	26
13. FINAL WORK SCHEDULE	27
14.REFERENCES	28

1. Abstract

For our 551 project, we implemented a real-time, musical vocoder on the DSK. We use the term "musical vocoder" to distinguish our project from the other kinds of vocoders; namely, the telecommunications vocoder and the phase vocoder. The primary method we used was linear predictive coding (LPC), which was accomplished using the Levinson-Durbin algorithm.

2. The Problem

A musical effect that has been popular for decades is making an instrument sound as though it is talking. The musical applications for doing this are extremely diverse, but two of the most popular uses are making a human voice sound robotic (see ELO's "Mr. Blue Sky" or Styx's "Mr. Roboto"), and causing one human voice to sound like many (see Infected Mushroom's "Cities of the Future"). One of the earliest methods of achieving this was a "Talk Box", used heavily by Peter Frampton. This device was basically just a speaker with a tube that was inserted into the mouth, so that when the guitar was played, the note of the guitar was transferred to the user's mouth, which could then be shaped by the mouth to form speech-like sound. Following soon after was the analog musical vocoder (Vocal Coder), a device that accomplishes the same thing, except instead of directly shaping the sound with one's mouth, the user speaks into a microphone and the formants from the user's speech are applied to the guitar. **Our project models a musical vocoder on the C67 DSK**. With a vocoder, not only is it possible to make your voice sound like anything (especially when used with a synthesizer), it also can be used to create vocal harmonies when given multiple excitation signals, the goal of 2006's Group 9. Our vocoder uses two inputs, a microphone (for speech) and a line-in (for excitation instrument). LPC is performed on both inputs, the instrument signal is inverse filtered to produce its residue (excitation) signal, which is then used as the input to a (different) filter which models the formant structure of the microphone input. The output of this combination is the sound of the input instrument speaking the words uttered by the user into the microphone.

3. Novelty

There are two main aspects to the novelty of any project:

1. The project or goal itself, and

2. The algorithms/methods used to achieve that goal.

Our project is novel for item #1 in that no other group has used the DSK for the exact application we are using it for; that is, modeling an analog vocoder (combining speech and music components). The project with the closest relation to ours in this respect is the "Sing-Synth" project from 2003's Group 9. Theirs is similar to ours in that it combines two signals (the user's voice and a synthesized musical signal) to produce the output. However, their approach (item #2) is completely different from ours, as they used pitch detection and subtractive synthesis rather than linear prediction. The result is that their method results in only the note being preserved from the user's voice, while in our project, the words the user is saying (the formants) are preserved instead, while the actual note is discarded.

As for item #2, our approach is linear prediction, which has been used in many previous 551 projects. Some of these projects dealt with coding speech signals to transmit them more quickly and efficiently (e.g. the CDMA Modem project from 2000's Group 16), while others have dealt with speech morphing. When taking both item #1 and item #2 into account, these speech morphing projects (e.g. "Hey, stop sounding like me!") are the most similar to ours, since they tried to make a person sound like another person, while we have tried to make a guitar sound like a person. However, our project contains an additional challenge since we are sampling both input signals in real-time, while all of the speech-morphing projects had at most one input and a training set.

It must also be noted that, while our project sounds like it might relate to projects using a "phase vocoder" algorithm (such as 2008's Group 2), it does not. This is because of the two different uses of the word "vocoder": one ("musical vocoder") is a device that combines speech and an excitation signal, which our project aims to model (item #1), while the other, ("phase vocoder") is a computer algorithm (item #2) used for many things, including time scaling and pitch shifting, but not combining different signals together. The

phase vocoder is simply named after the musical vocoder because it works similarly (separating excitation signals from residue signals, processing, then recombining); however, they are quite different because one IS an application (our project) while the other HAS an application. To be even more clear:

Our Project:	**2008 Group 2:**
1. **Goal:** Combine speech and music (**musical vocoder**)	1. **Goal:** Beat detection/synchronization
2. **Method/Algorithm:** LPC using Levinson-Durbin	2. **Method/Algorithm:** STFTs and **phase vocoder**

4. Solution

To obtain an output to sound like the musical instrument talking we separated the formant structure of the speech and then applied the shape of that formant structure to the excitation signal that we formed by generating the residual of the instrument signal [4]. Separating the formant structure is done using Linear Predictive Coding which from now on will be referred to as LPC. Figure 4.1 shows a very simple flow diagram of the process we used to compute the desired output.

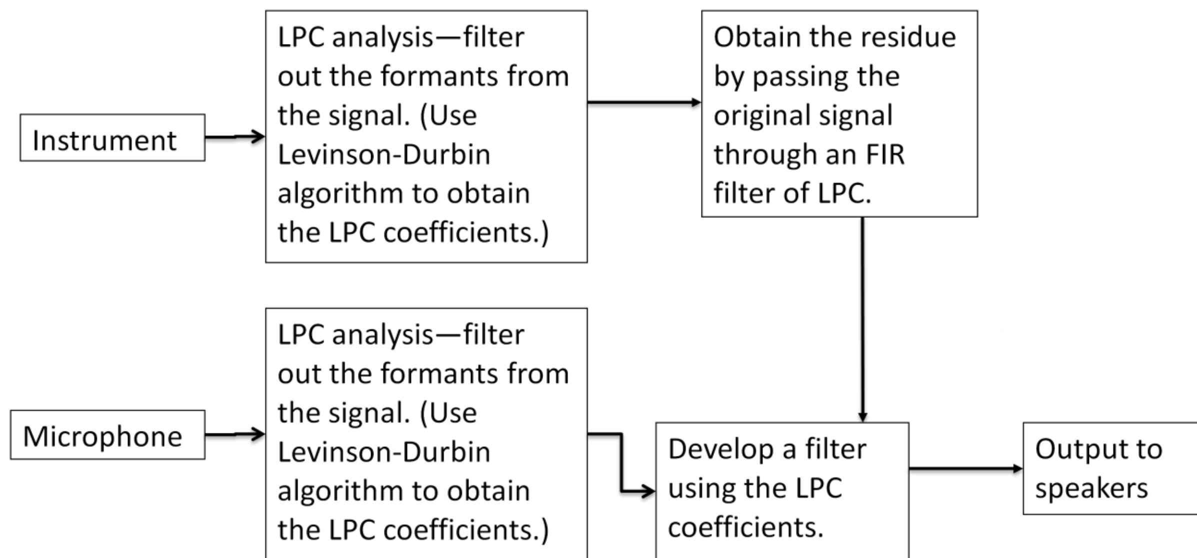


Figure 4.1: Simple flow diagram of our solution

The following steps explain the procedure shown in Figure 4.1. A detailed explanation of this procedure is discussed later once the algorithm is explained.

a) The two inputs are fed into each channel of the line-in and are sampled directly by the ADC of the C67 DSK. A Hamming window is used to operate on the microphone signal so that the coefficients will have all the required information and drastic frequency changes at the beginning and end of each frame due to Gibbs phenomena are avoided. The size of the frame selected is critical to the delay and resolution of speech. This is discussed later.

b) The LPC coefficients for both the inputs are calculated. New LPC coefficients are calculated once every frame

c) The LPC coefficients from the microphone signal are used to develop a filter. The LPC coefficients of the instrument signal are used to generate the excitation signal from the musical instrument.

d) The residue is calculated by passing the instrument signal through an inverse filter developed by the signal's own LPC coefficients.

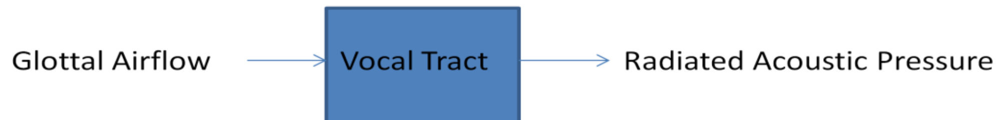
e) This residual signal will function as the excitation signal which is then passed through the filter created by the LPC coefficients of the speech signal.

f) The output thus formed has the characteristic sound of the musical instrument, filtered to sound as though it is talking.

5. What is Linear Predictive Coding or LPC? [1]

The source-filter model of speech production models speech as a combination of a sound source (i.e. the vocal cords), and a linear acoustic filter, the vocal tract (and radiation characteristic). An important assumption that is made in the use of the source-filter model is the independence of source and filter. In such cases, the model should more accurately be referred to as the "independent source-filter model".

- Vocal Tract and LPC model analogy.



- Assumptions :
  - Linearity – The vocal tract is a linear acoustic system
  - Independence – The properties of glottal source and glottal tract are independent.

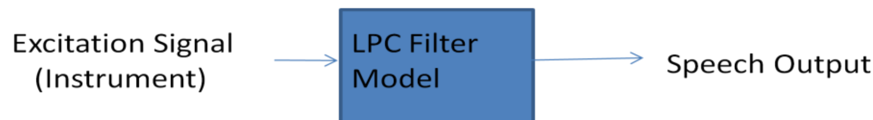


Figure 5.1: Analogy between source filter model for voice production and A digital system using LPC [1]

Linearity is defined mathematically for a system (or mathematical function) that has an independent variable (the input) and a dependent variable (the output). The term describes certain relations between the input and output. Without going into mathematical detail, it can be shown that a hard-walled system of tubes with no sharp bends, extreme constrictions or sharp projections into the flow path (demonstrated as a trumpet horn) is a linear acoustic system for sounds of reasonable amplitude. The vocal tract is a fairly linear acoustic system, if vibration of the softer walls, such as the cheeks or velum can be neglected. Physiological systems are generally non-linear, though a linearity assumption is plausible and is required for our application.

LPC starts with the assumption that a speech signal is produced by a buzzer at the end of a tube (voiced sounds), with occasional added hissing and popping sounds (sibilants and plosive sounds). Although apparently crude, this model is actually a close approximation to

the reality of speech production. The glottis (the space between the vocal folds) produces the buzz, which is characterized by its intensity (loudness) and frequency (pitch). The vocal tract (the throat and mouth) forms the tube, which is characterized by its resonances, which give rise to formants, or enhanced frequency bands in the sound produced. Hisses and pops are generated by the action of the tongue, lips and throat during sibilants and plosives. LPC analyzes the speech signal by estimating the formants and estimating the intensity and frequency of the remaining buzz. This decomposition of speech sounds is done in two parts:

1. A filter function consisting of LPC coefficients
2. A source function which is the excitation signal of a musical instrument in our application but can also be a person's voice or an impulse signal generated by a computer at a particular fundamental frequency.

A vocoder uses the LPC coefficients and re-synthesizes speech by filtering the excitation signal through the speech filter (i.e. filtering 2 through 1).

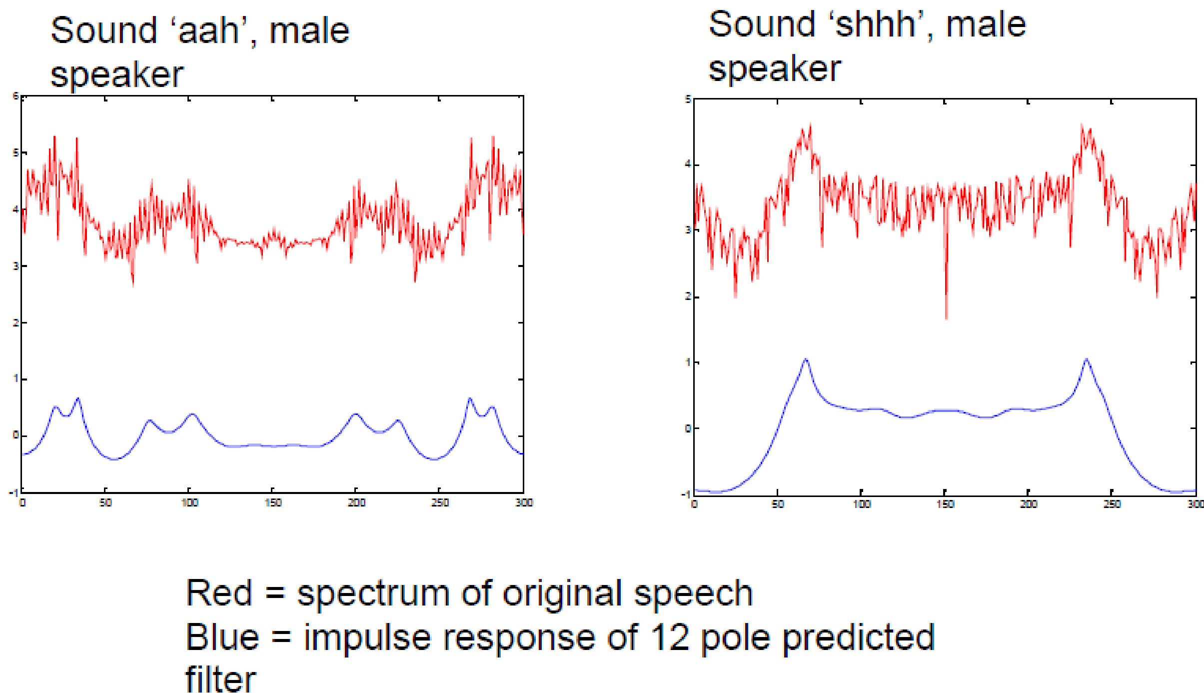


Figure 5.2: Shows how LPC depicts the filter response of the original signal [2]

6. The Levinson Durbin Algorithm to calculate LPC [3]

There are three types of filter design modeling that can be used.

1. The *moving average* (MA) model has zeros but not poles:

   $H(z) = B(z)$

2. The *autoregressive* (AR) model has poles but not zeros:
   $H(z)= G/A(z)$
3. The third type of model has both poles and zeros and is called the *autoregressive moving average* (ARMA) model:
   $H(z)= B(z)/A(z)$

Of the three types of filter design by modeling, the all-pole AR model is the most commonly used, largely because the design equations used to obtain the best-fit AR model are simpler than those used for MA or ARMA modeling. Serendipitously, the all-pole model also has the ability to describe most types of speech sounds quite well, and for that reason we have used it in our vocoder.

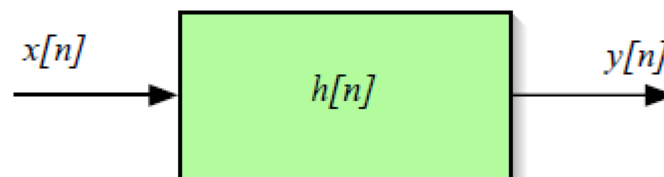


Figure 6.1: The Autoregressive Moving Average (All Pole) Model

The transfer function for an all-pole filter model can be given by:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{G}{1 - \sum_{k=1}^{P} \alpha_k^{(P)} z^{-k}} \qquad (1)$$

Where H(z), Y(z) and X(z) are the z-transforms of the filter, output and input respectively. Gain is denoted by G, $\alpha_k^{(P)}$ denotes the LPC coefficients of a Pth order model.

Since the representation is not orthogonal, all the coefficients change when the order of the model (P) changes.

The inverse z-transform of this function would yield:

$$y[n] = Gx[n] + \sum_{k=1}^{P} \alpha_k^{(P)} y[n-k] \qquad (2)$$

Now we want to obtain a filter transfer function of the form in Eq(1) to an arbitrary desired filter transfer function, $H_d(z)$. This is done by minimizing the average square error between the magnitude of the frequency response of the desired filter $H_d(e^{j\omega})$ and all the all-pole filter that is obtained $H(e^{j\omega})$.

$$\xi^2 \equiv \frac{1}{2\pi} \int_{-\pi}^{\pi} \left| H(e^{j\omega}) - H_d(e^{j\omega}) \right|^2 d\omega \qquad (3)$$

Applying Parseval's theorem to Eq. (3) we obtain,

$$\xi^2 = \sum_{n=0}^{N-1} (h[n] - h_d[n])^2 \qquad (4)$$

Since h[n] is the system's response to the unit sample function δ[n], we obtain from Eq. (2),

$$h[n] = G\delta[n] + \sum_{k=1}^{P} \alpha_k^{(P)} h[n-k] \qquad (5)$$

And

$$\xi^2 = \sum_{n=0}^{N-1} \left(\left(G\delta[n] + \sum_{k=1}^{P} \alpha_k^{(P)} h[n-k] \right) - h_d[n] \right)^2 \qquad (6)$$

For a particular model order P we solve for each α_k by writing Eq. (6) with a different internal dummy variable, obtaining the derivative of ξ^2 with respect to α_k and setting the result to zero.

$$\frac{d\xi^2}{d\alpha_k} = \sum_{n=0}^{N-1} \left(\left(G\delta[n] + \sum_{l=1}^{P} \alpha_l^{(P)} h[n-l] \right) - h_d[n] \right) h[n-k] = 0 \qquad (7)$$

$$\sum_{n=0}^{N-1} \left(G\delta[n]h[n-k] + \sum_{l=1}^{P} \alpha_l^{(P)} h[n-l]h[n-k] \right) = \sum_{n=0}^{N-1} h_d[n]h[n-k] \qquad (8)$$

Since the system is causal G will not enter into the solution. Hence our final form is,

$$\sum_{l=1}^{P} \alpha_l^{(P)} \sum_{n=0}^{N-1} h[n-l]h[n-k] = \sum_{n=0}^{N-1} h_d[n]h[n-k] \qquad (9)$$

Using the auto-correlation function symbol $\Phi[m]$ this Eq boils down to,

$$\alpha_1\phi[0] + \alpha_2\phi[1] + \alpha_3\phi[2] + \alpha_4\phi[3] = \phi[1] \qquad (21)$$

$$\alpha_1\phi[1] + \alpha_2\phi[0] + \alpha_3\phi[1] + \alpha_4\phi[2] = \phi[2]$$

$$\alpha_1\phi[2] + \alpha_2\phi[1] + \alpha_3\phi[0] + \alpha_4\phi[1] = \phi[3]$$

$$\alpha_1\phi[3] + \alpha_2\phi[2] + \alpha_3\phi[1] + \alpha_4\phi[0] = \phi[4]$$

Assuming here that P=4 (4 predictor coefficients).

It can be written in the form of a vector such as,

$$\begin{bmatrix} \phi[0] & \phi[1] & \phi[2] & \phi[3] \\ \phi[1] & \phi[0] & \phi[1] & \phi[2] \\ \phi[2] & \phi[1] & \phi[0] & \phi[1] \\ \phi[3] & \phi[2] & \phi[1] & \phi[0] \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} \phi[1] \\ \phi[2] \\ \phi[3] \\ \phi[4] \end{bmatrix} \quad (22)$$

This is of the form:

$\boldsymbol{R}\alpha = \boldsymbol{P}$

where ***R*** is a P X P matrix of autocorrelation coefficients, α is a P X 1 vector of the $\{\alpha k\}$, and ***P*** is a P X 1 vector of autocorrelation coefficients. This equation is known as the *Wiener-Hopf equation.*

A direct solution to the Wiener Hopf equation can be obtained by pre-multiplying both sides by the inverse of ***R***:

$\alpha = \boldsymbol{R}^{-1} \boldsymbol{P}$

The inversion of the ***R*** matrix can be accomplished by Gaussian elimination and other similar techniques,

which are in $O(N^3)$computational complexity. In our project, however, a simpler solution known as Levinson-Durbin recursion is used because the correlation matrix ***R*** is *Toeplitz*; all the matrix elements of each diagonal, major and minor, are identical. Exploiting this symmetry the Levinson-Durbin recursion has a complexity of $O(N^2)$.

6.1 The Levinson - Durbin recursive Algorithm:

Levinson-Durbin recursion provides for $\{\alpha k\}$ a faster solution for in the system of equations for situations in which the matrix on the left side of the equation is a Toeplitz matrix. In our application, the $\{\alpha k\}$ represent the autocorrelation coefficients of the random process y[n]. The solution $\{\alpha k\}$ are the Pth - order predictor coefficients for the best-fit linear predictive model that transforms a white random process x[n] into a random process that has autocorrelation coefficients phi according to the equation.

$$y[n] = \sum_{k=1}^{P} \alpha_k^{(P)} y[n-k] + Gx[n]$$

The equations of the Levinson-Durbin recursion, which are used to compute the corresponding reflection coefficients and LPC parameters are:

$$E^{(0)} = \phi[0]$$

$$k_i = \frac{\left\{\phi[i] - \sum_{j=1}^{i-1} \alpha_j^{(i-1)} \phi[i-j]\right\}}{E^{(i-1)}}, \text{ calculated for } 1 \le i \le P$$

$$\alpha_i^{(i)} = k_i$$

$$\alpha_j^{(i)} = \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)}, \text{ for } 1 \le j \le i-1$$

$$E^{(i)} = (1-k_i^2)E^{(i-1)}$$

The coefficients $\{k_i\}$ for $1 \le i \le P$ are referred to as the *reflection coefficients.* They constitute an alternate specification of the random process y[n] that is as unique and complete as the LPC predictor coefficients $\{a_k^{(P)}\}$. The reflection coefficients are not required in our application and so they are discarded in our project but they have to be computed since they are required to calculate the next LPC coefficients.

If the magnitude of the reflection coefficients $|k_i|$ is less than 1 for $1 \le i \le P$, all of the roots of the polynomial

$$A(z) = 1 - \sum_{k=1}^{P} \alpha_k^{(P)} z^{-k}$$

will lie inside the unit circle. This means that if $|ki| < 1$, the resulting filter H(z) will be stable. It can be shown that deriving $\{k_i\}$ the in the fashion described above using Levinson-

Durbin recursion guarantees that $|k_i| < 1$. **This means that the Levinson Durbin Algorithm guarantees that the system is always stable.**

6.2 Comparison of Levinson-Durbin (LD) Algorithm with other algorithms:

LD v/s Cholesky Decomposition: The Cholesky decomposition is a method used to find the inverse of a matrix which has Hermitian Symmetry. However the computational complexity of the Cholsky decomposition is O(N3) as compared to the LD algorithm which is O(N2) since LD exploits the fact that LPC analysis has Toeplitz Symmetry.

LD v/s Schur Decomposition: The Schur decomposition states that every square matrix **A** is unitarily similar to an upper triangular matrix **T** with $\mathbf{A}=\mathbf{U}^H\mathbf{T}\mathbf{U}$. This method is also slower than the LD algorithm.

LD v/s Widrow-Hoff (Least mean square or LMS) algorithm: The Widrow algorithm is not a linear prediction algorithm but an adaptive filter technique that can be used to predict the filter structure similar to LPC. However the Widrow-Hoff algorithm does not guarantee minimum phase systems and stability. Since LD creates a minimum phase system (all poles and zeros lie within the unit circle) so it is always stable and even its inverse is always stable. It is important that the system is minimum phase since we use the inverse filter to find the residual. Although the Widrow-Hoff method is a more elegant and accurate method of prediction, it is considerably slower than the Levinson-Durbin Algorithm. Table 6.2.1 shows the comparison of clock cycles per frame for a 180 sample frame size with speech sampled at 16kHz. [6]

Block	Number of Clock Cycles	Execution Time
Autocorrelation	105372	1.756ms
Levinson-Durbin	25075	0.418ms
LMS	310985	5.18ms
Modified signed LMS	371663	6.19ms
FIR	12423	0.21ms
IIR	13637	0.23ms
Overall using LMS	337045	5.62ms
Overall using modified signed LMS	397723	6.63ms
Overall using Levinson-Durbin	156507	2.61ms

Table 6.2.1: Comparison of the overall time taken by the Widrow-Hoff and Levinson Durbin Algorithm [6]

7. Procedure and Algorithm

Input: As we stated earlier, we needed to input two signals to the C67 DSK simultaneously. One was from the microphone and the other the musical instrument. For this we used the stereo line-in input of the DSK. Since the line-in is a stereo input we fed both, the microphone and the instrument, to each individual channel of the line-in of the DSK. The microphone was fed into the right channel while the instrument into the left channel. (The reason why we couldn't use both the line-in and the mic-in simultaneously to sample the two inputs is discussed in later Section 10) Since the microphone and instrument signals are implicitly mono there was no loss of information by feeding them mono into the DSK (i.e 1 on each channel of the stereo input). Pre-amplifiers were required before feeding the signals into the line-in since the microphone and instrument (guitar in our demo) signals are of the order of a few millivolts. They thus have to be amplified to ~1 volt before inputting them to the line-in of the DSK.

The two signals were sampled at a **rate of 16kHz**. Since all speech information is present between about 500-3500 Hz, we could do with a sampling rate of 8 kHz to accurately capture all of the vocal frequencies but since one of our inputs was a musical instrument we decided to select a higher sampling rate to accommodate higher frequencies which would be generated by the instrument.

Once the inputs were sampled by the ADC (analog-to-digital converter) of the DSK they were stored in a circular buffer. A circular buffer is a buffer which is only a fixed length long (fixed memory locations). Once the buffer is fully accumulated the pointer will move back to the first location of the buffer and will start replacing the old data. Figure 7.1 shows the working of a circular buffer.

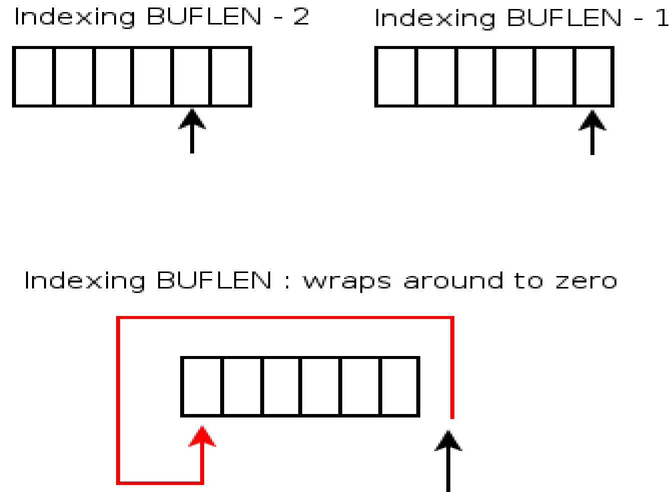


Figure 7.1: Functioning of a Circular Buffer

Each piece of data stored in the buffer is a 32-bit unsigned integer value with the lower 16 bits holding the data of the right channel (microphone) and the upper 16 bits holding the data of the left channel (instrument). We split these signal's data into two separate floating point buffers to process them individually. Since the Levinson-Durbin recursive algorithm works on floating point we had to type cast them to floating point values. ***Processing:*** All the processing that was done in our project was done on the DSK. We did not need to use the computer for any part of the processing since the memory and processing power of the DSK was sufficient for our application. The clock cycles and time taken to run these functions are described in Section 9.

The operations or processing was broken up into 6 steps.

a) ***Windowing:*** Just processing on a short frame of data assumes a rectangular window. This means that the entire input signal is multiplied by a rectangular function which is 1 for duration of the frame and 0 otherwise. Using a rectangular window on the input causes Gibbs phenomena to occur on the frame. Gibbs phenomena are the ringing artifacts that occur during processing as a result of the abrupt cutoffs at the beginning and end of the frame. The method to avoid this is to window the signal choosing an appropriate window function. In our application we choose a Hamming window since its frequency response is more tapering in the higher frequencies and so it filters the higher

frequencies better. The length or size of the window function is defined by the frame size. A hamming window is defined by the expression,

$$w(n) = 0.54 - 0.46\ \cos\left(\frac{2\pi n}{N-1}\right)$$

where n is the length of the window (frame size in our case).

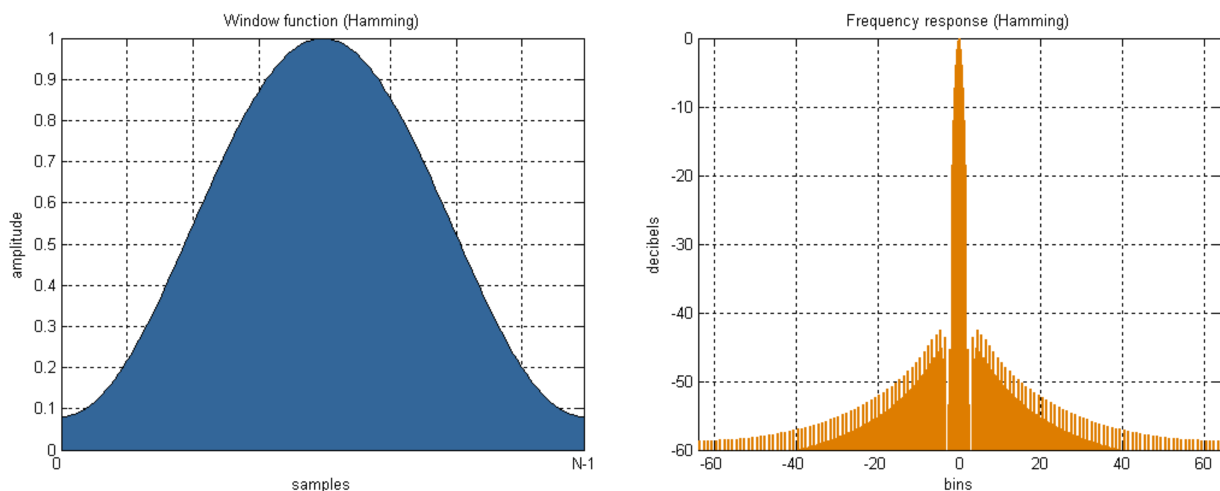


Figure 7.2: The Hamming window and its frequency response

b) ***Autocorrelation*** *[5]*: To calculate the LPC coefficients using the Levinson-Durbin algorithm, P+1 autocorrelation values of the signal are required (where P is the order of the LPC), as shown previously in Section 6. Autocorrelation can be calculated by shifting the copy of the signal over itself and summing up all the overlapping values of both functions over n (where n are samples). It is computed using the following expression.

$$R(j) = \sum_{n} x_n x_{n-j}.$$

c) ***Levinson-Durbin Recursion*** *[5]*: The Levinson-Durbin recursive algorithm as was discussed in Section 6 is used to compute the P order LPC coefficients using the autocorrelation values that were computed earlier. This algorithm calculates P coefficients from P+1 autocorrelation values iteratively. Along with the LPC coefficients

the algorithm also computes the reflection coefficients which are not required to form the all-pole filter but are needed to predict the next LPC coefficients. It also returns the deviation or error between the actual frequency response and the predicted response for each frame. The error obtained for each frame is to be divided by 10000^2 since it has to be normalized lie between 0-1. Since the ADC of the DSK gives values between -10000-10000 instead of 0-1.

d) ***Residual**[4]**:*** The original signal of the instrument contains a filter shape of the instrument (characteristic of the instruments body, pick-ups, electrical filter due to line inductance and shunt capacitance). The residual of the signal is obtained by separating the original signal of the instrument from its formant structure. This is done by finding the formant shape using LPC analysis and then passing the original signal on the instrument through an inverse filter generated by its own LPC coefficients (i.e. an all-zero or FIR filter).This residue is used as an excitation signal which is shaped by the speech formants. Using the original signal directly as the excitation would lead to an output signal containing both the filter shape of the instrument and the speech which would distort the output. It is important to take care of the initial conditions while applying a filter to any signal. Here since P LPC coefficients are used the order of the filter is P. Thus the instrument signal variable has P initial values to compute the output of the first sample. So the size of the instrument variable we used was frame size + P and at the end of every frame the first P samples of the instrument were overwritten with the P last input values of that frame which were the initial conditions of the next frame. So the first actually input sample was the Pth sample of the input variable since the previous P samples were just initial conditions.

e) ***Filter:*** To obtain the desired output finally the residual of the instrument has to be passed through an all-pole (autoregressive) filter developed using the LPC coefficients of the speech signal. This applies the shape of the speech formants to the residual. The function implemented is a simple difference equation of the form

$\text{output}[n] = \text{input}[n] + \Sigma\ \alpha_k\ \text{output}[n-k]$

where the summation k ranges from 0 to P (order of LPC coefficients computed) for the nth sample. It is important to take care of the initial conditions while applying a filter to any signal. Here similar to the residual function, to take care of the initial conditions the output variable was frame size + P samples long and at the end of every frame the first

P values of the output variable were overwritten by the last P samples of that frame which became the initial conditions for the next frame. So the first actually output sample was the Pth sample of the output variable since the previous P samples were just initial conditions.

f) ***Output****:* The final stage is the output. The output is then stored in an output buffer which can be sent to the line-out. One output and one playback buffer was used to avoid overwriting of old data if the processing speed is faster than the output speed. This technique is called the ping-pong technique by which processing of the next frame continues while the DSK outputs the values of the previous frame.

8. Specifications

a) **Sampling Rate:** As discussed earlier the sampling rate we choose was 16kHz. Though 8kHz would have been sufficient to model the speech signal without any loss of information. However since the other input was a musical instrument we decided to use a higher sampling rate.

b) ***Frame Size:*** *Variable*

The frame size was variable to set to any value that gives the best output. However selection of an optimal frame size is important. The frame size should be as small as possible to avoid large delay between the input and the output since each frame is outputted only after one complete frame is processed. Opposed to this a smaller frame size also reduces the information content that describes the LPC coefficients for that frame. So smaller the window, less accurate or less resolved are the LPC coefficients.

Thus a window size between 250-600 samples proves to be the good fit considering these limitations since a human ear cannot distinctively identify a delay of 25-32ms.

c) ***Frame Rate:*** For a sampling rate of 16kHz and a frame size of 512 samples the frame rate = sampling rate/frame size = 31.25hz. In other words in every 1/31.25 = 32ms a new frame is loaded.

d) ***Output Data Rate:*** The DSK outputs 16bits (1 sample) every 0.000625sec. So in 1 sec it outputs 256kb/s or 32kB/s.

e) ***Code Size:*** ONCHIP_PROG = 51,360 bytes

ONCHIP_DATA = 52,579 bytes

SDRAM = 65,536 bytes

9. Function Profile Times, Speeds, Latency and Optimization

The following profile times in Table 9.1 are for a frame size of 512 and 32 predictor coefficients.

Function Name	Clock Cycles	Time
Hamming Window	114,823	0.505 ms
Autocorrelation	69,004	0.3 ms
Levinson-Durbin	17,981	0.0791 ms
Residual	244,843	1.07 ms
Filter	267,745	1.17 ms

Table 9.1: Measured clock cycles and time taken for each function

Table 9.1 shows that the total processing time for all the functions per frame was 2.6696ms. Including the for loops in the main to copy data into the input variables and the output back into the playback buffer the total processing time per frame was about 5ms. The time taken to load one frame of 512 samples at a sampling rate of 16kHz is 32ms. This shows that the processing is real fast and thus could be implemented in real time.

To improve the performance, we manually unrolled the innermost loops of the nested for-loops in the filter and residual functions. That is, replace the small inner loop with the necessary number of multiplications and additions, directly referencing the array locations to be used. In the original nested for-loop, the innermost loop is performed 32 times. Since the number of iterations of the loop is quite large, we unrolled part of the loop so it is only performed 8 times. This aids the compiler in optimization.

Function Name	Clock Cycles	Time
Residual	135,713	0.597 ms
Filter	123,099	0.5416 ms

Table 9.2: Measured clock cycles and time taken for each function

Table 9.2 shows that after unrolling the loops for the two functions, residual and filter, they both were around 100,000 cycles faster than the original functions. It is thus evident that by manually unrolling the loops, the speed of both functions increased significantly. To

improve the speed of critical operations, we enabled code compiler to perform optimizations. It helped to generate parallel code where possible. To enable optimization, we set the Opt Level to “File(-o3)”, Program Level Opt. to “No Ext Func/Var Refs (-pm – op2)” and Interlisting to “Opt/C and ASM (-s)”. In the Feedback menu, we check the option box for Generate Optimizer Comments (-os). This will allow the software to pipeline schedules instructions from a loop so that multiple iterations of the loop can execute in parallel. From the assembly file the optimizer produces, we were able to obtain how many iterations of the last loop not manually unrolled are run in parallel.

Function	# of iterations run in parallel
autocorrelation()	4 iterations
hamming()	not qualified since there is a call
levinson()	4 iterations for the first loop; 2 iterations for the second loop
residual()	1 iteration for the first loop; 7 iterations for the second loop
filter()	1 iteration for the first loop; 7 iterations for the second loop
main()	3 iterations for the first loop; 4 iterations for the second loop; 7 iterations for the third loop

Table 9.3: Compiler optimization (Level 3)

It can be seen from Table 9.3 that the in residual and filter function, there are 7 iterations found in parallel for the second loop. This number is quite strange since there are only 5 parallel iterations found in the assembly file, and as the assemble code indicated that the compiler is only using 2 registers during these iterations. This could be a result of the compiler freely reordering the associative floating-point operations and mistaking the number of iterations. (TI Optimizing Compiler User’s Guide, section 3.9) It could also be a result of the assembly statements that attempt to interface with the C/C++ environment or access C/C++ variables have unexpected results. (TI Optimizing Compiler User’s Guide, section 3.10)

In the main function, there are also 7 iterations found in parallel for the last loop. Although the number is a little bit high, it is at least conceivable, since the loop is used for exporting data to playback buffer.

10. Errors, Problems & their Solutions

Processing time and playback buffer fix:

We initially wrote our code to process the each frame entirely between the last sample of the previous frame and the first sample of the incoming frame, because we assumed (naively) that the length of 1 sample (1/16000 of a second = 14,062.5 cycles) would be plenty of time to do this processing. We later found, after profiling our code, that processing a frame actually takes around 500,000 cycles or so, depending on the values used for "FRAMESIZE" (number of samples per frame) and "P_MAX" (order of the LPC filter).

So instead of performing all of our processing between frames, we switched to processing each frame while playing back the previous frame. Thus, we gave ourselves the length of one frame to complete the processing, rather than just the length of one sample. We achieved this by using Group 3's double-buffer playback method.

There are two playback buffers, playback1 and playback2. Playback1 is filled while playback2 is played back by the xmitISR() function. Then, when xmitISR() reaches the end of the frame, the pointers to the buffers are swapped, and playback2 gets filled up while playback1 is being played back. This method ensures that playback is always smooth, as long as processing is done faster than the frame takes to finish playing back. Again, we took this ingenious solution from Group 3, who generously offered it to us when we were trying to figure out a way to process a frame during the previous frame's playback instead of between frames.

This method introduces a latency equal to the length of 1 frame, which is 32ms for a 512-sample frame. This negligible latency is necessary so that our processing code can finish completely before its output is played back.

Error gain:

The Levinson-Durbin algorithm works by finding a filter that introduces the smallest mean squared error between the predicted values and the actual values. This small error introduces a gain into the filter when it is represented using the LPC coefficients. If this gain is not taken into account, the output of the vocoder is highly distorted.

We fixed this by storing the filter gain in a variable (after it is returned from the levinson() function) and multiplied this variable with the output for every frame. This reduced the distortion in the output.

Receiving and processing two audio signals simultaneously:

At first we thought that the DSK, having two audio inputs (MIC-IN and LINE-IN), was capable of processing these two inputs simultaneously. Realizing that there was nothing in the CODEC documentation about this, we had to come up with a different solution.

The first idea we had was to take a sample in from the MIC-IN line, then quickly switch the ADC to process the LINE-IN input, and take a sample from LINE-IN, and continue switching back and forth between MIC-IN and LINE-IN at 16 kHz. This would have produced satisfactory results because it would have enabled us to take in both inputs more or less simultaneously. However, it seems that the ADC does not fully switch from one input to another fast enough (1/32000 of a second) for this method to work, and the result was that both signals were bleeding into the other signal's input buffer.

We then realized that we could simply use the LINE-IN input for both signals. LINE-IN, being a stereo input, can receive two mono signals simultaneously, and this was perfect for our project since a microphone and an instrument both produce mono signals. The problem with this solution was that both the instrument signal and the microphone signal were what is referred to as "mic-level" signals (a few millivolts), while the LINE-IN jack is designed for "line-level" signals (around 1 volt). Thus, our input signals needed to be amplified by a couple of orders of magnitude before they would be of any use.

At first we tried to solve this problem using microphone amplifiers from Radio Shack, but these proved to be too noisy and not loud enough. So instead we simply used our desktop workstation as a microphone pre-amp (many computers with sound cards have a setting to output directly from the microphone) to boost the signal to line level, and this produced wonderful results. For the guitar, we at first used a second computer as the amplifier, then switched to a real guitar amplifier (since that's what they are for) which was already in the possession of one of our members at the time.

Testing with the correct waveforms:

Formant filters (the type of filter that we are using) work by shaping the power spectrum of the upper frequency bands of an input. Therefore it was impossible to use a sine wave as the test signal for the instrument input, since a sine wave, having no frequency components other than its fundamental, is not affected by the application of such a filter. For this reason, it is also impractical to use a square wave or a triangle wave for testing, since these also have much lower energy in the upper frequency bands than in the fundamental. Once we began using a pulse wave (a signal with lots of high-frequency energy) as the test excitation signal, our test results made much more sense. (The source-filter model for a human voice uses a pulse wave as the excitation, so this was the obvious choice).

11. Code Information

Vocoder using LPC:

Code type: Matlab code

Source: Given to us by Prof. Richard Stern (CMU)

Description: This Matlab code takes a speech waveform from a file and shows how LPC analysis is used to separate the formant structure of speech and re-synthesize it using a pulse waveform generated by the code itself. It uses overlapping to obtain better speech resolution (clarity) and has a frequency variable which can be changed (100hz, 200hz etc) to set the frequency of the pulse wave which is used as the excitation signal. It can be observed that by changing the frequency of the pulse wave the pitch of the speech at the output changes accordingly, regardless of the original speech signal. The characteristic voice of the person speaking is lost and the output sounds robotic (the characteristic of a pulse waveform). This code was not written to work in real-time but it processed a stored sound file in the wave format.

Instrument LPC:

Code type: Matlab code

Source: We wrote the code ourselves.

Description: Using Prof. Stern's Matlab code as a reference, we made our own Matlab code modeling our vocoder. Two files, a speech file and an instrument file, are loaded into Matlab. The code performs LPC analysis similar to Prof. Stern's code on the speech signal but that is extended to perform LPC analysis on the instrument signal too. The residual is then computed by passing the instrument signal through an inverse filter generated by its own LPC coefficients. This residual is used as an excitation signal to re-synthesize speech. In this code the frame size and the number of LPC coefficients are variable. These can be changed accordingly to observe how the output varies with different frame sizes and number of LPC coefficients. This can be use to find the optimum specifications to produce a desirable output. This code is also not real-time.

Hamming Window:

Code type: C code

Source: We wrote this code ourselves

Description: This function applies the Hamming window defined by the following equation,

$$w(n) = 0.54 - 0.46\ \cos\left(\frac{2\pi n}{N-1}\right)$$

Autocorrelation:

Code type: C code

Source: Online [5]

Description: This function computes the P+1 autocorrelation values of a signal of length frame size where P is the order of predictor coefficients that are used to describe the formant structure.

Levinson-Durbin:

Code type: C code

Source: Online [5]

Description: The Levinson-Durbin function computes the LPC coefficients from the autocorrelation values that were calculated earlier using the Levinson-Durbin recursive algorithm that was discussed earlier. Along with the predictor coefficients it also gives the reflection coefficients and the error or each frame. In our application since we do not use

the reflection coefficients we discard them. However they are calculates since they are used to predict the next coefficients.

Residual:

Code type: C code

Source: We wrote this code ourselves

Description: This function calculates the residual of the instrument by passing the original signal through an inverse filter (FIR filter) generated using the LPC coefficients of the original instrument signal itself.

Filter:

Code type: C code

Source: We wrote this code ourselves

Description: This function applies the AR (all-pole) filter to the excitation signal (residue of instrument) to obtain the output.

12. Demonstration

In the demonstration, the instrument we selected was an electric guitar which was plugged into a guitar amplifier whose output was fed into the DSK. We demonstrated the functioning of the vocoder by singing into the microphone and simultaneously playing the guitar. The output was fed into a speaker system that played it out loud. At first we demonstrated the clarity of the speech and its recognition in the output. We then varied the number of predictor coefficients (P) and showed that when P is too small the speech formants are not well defined and so the output speech is less recognizable. On increasing the predictor coefficients (P) the speech in the output got clearer. However, when P was increased too much with a small frame size the time taken for processing increased for greater P and so the output had some distortion, which sounded like whispers. Next we varied the frame size to show how having a larger frame size caused more delay since the latency was 1 frame since at one time an entire frame is processed and then outputted altogether. By reducing the frame size too much, the information to form the filter was too little and so the resolution of the predictor coefficients reduced.

13. Final Work Schedule

Since we did not find any papers online that exactly described our application we had to first study papers which described the use of LPC for other applications such as voice compression and transmission in telecommunication. We then studied the MATLAB code that we got from Prof. Stern. This gave us a basis for understanding the algorithm, as well as a convenient way to test different settings. We then re-wrote that code in MATLAB to suit our project's purposes, then re-wrote that code in C and made it run in real-time on the DSK.

Week	Task	Person
October 5	Studied Prof. Stern's MATLAB code	Pritish, Laura
	Wrote MATLAB code for our application	Pritish, Chris
October 12	Wrote C code to implement the Hamming window, residue and filter functions	Pritish
	Implemented C code that we found online for autocorrelation and Levinson-Durbin recursion	Pritish
October 19	Test inputting 2 signals to the DSK simultaneously	Chris, Laura
	Run parallel inputs together in real-time without any processing	Laura
October 26	Implemented C code for the functions on the DSK and compared them to the results obtained by MATLAB	Laura, Pritish
November 9	Adjusted C code for the DSK to process the signals and run them in real-time	Chris
November 16	Debugged the code to run in real-time without delay with correct processing	Chris, Pritish
	Tested using excitation signal as produced from a function generator	Everyone
November 23	Improved code by adding error correction and reduced output distortion and improved speech clarity	Everyone
	Profiled functions, optimized code and manually unrolled loops to improve processing speed	Everyone

Table 13.1: Final Work Schedule

14. References

[1] The Source-Filter Model Lives, a paper by Martin Rothenberg at the voice foundation 37th Annual Symposium.

http://www.rothenberg.org/source-filter-lives/Source-Filter-Lives-paper-as-presented5.pdf

-This paper explains under which conditions the source-filter model can be used for voiced speech and how the vocal tract can be digitally modeled using LPC.

[2] Bob Beauchaine. A Simple LPC Vocoder.

http://www.bme.ogi.edu/~ericwan/EE586/Students/Vocoder.pdf

-Describes vocal tract modeling using LPC and separation of formant structure from the signal.

[3] http://cnx.org/content/m10482/latest/

-Theory of LPC analysis and Synthesis

[4] Nelson Lee, Zhiyao Duan and Julius O. Smith III. Excitation Signal Extraction for Guitar Tones.

https://ccrma.stanford.edu/~nalee/Publications_files/icmc07-1.pdf

-This paper describes the different methods that are used for extracting an excitation signal from a guitar signal.

[5] http://www.musicdsp.org/showone.php?id=137

-We got the C code for autocorrelation and Levinson Durbin recursion from this webpage. It is a repository of various functions and algorithms that can be used for music processing.

[6] Towards A Real-Time Implementation of Loudness Enhancement Algorithms on a Motorola DSP 56600- Adnan H. Sabuwala

http://etd.fcla.edu/UF/UFE0000602/sabuwala_a.pdf

-Page 53 talks about the comparison in clock cycles, between the Widrow-Hoff and Levinson-Durbin Algorithm. This paper also talks about the advantages of using Widrow-Hoff algorithm for telecommunication as a replacement for Levinson-Durbin recursion. In addition it contains assembly code for Levinson Durbin, FIR & IIR filters and the LMS adaptive algorithm.

[7] A.O.Afolabi, A.Williams and Ogunkanmi Dotun. Development of text dependent speaker identification security system.

http://medwelljournals.com/fulltext/rjas/2007/677-684.pdf

-This paper shows how cepstral analysis can be used in speaker identification in security systems.

[8] www.data-compression.com

-This website is a repository of programs and papers on data-compression. They have various codes for LPC such as LPC-10, MELP 1.2 etc. Though we did not use any code from this website since the LPC code was written for telecommunication, it is a useful page to understand LPC and other compression methods for audio and image processing

[9] http://www.youtube.com/watch?v=_D54kHes_cI

-Commercial application of the vocoder effect that we developed.