Audio Streaming Sportscaster: Speech Synthesis

Nick Cohen ncc@ **Chris Guerra cguerra@**
Christy Kelly ckelly@ **Christian Rodriguez cpr@**

Group 16 Spring 2000 18-551 Digital and Signal Communication Systems Project Report

Abstract

One of the major complaints of computers today is that they are not capable of human-like interaction. They do not know their user or even if a user is present. One emerging technology that gives computers a way to interact with its user is speech synthesis. Speech synthesis and speech recognition technologies are now standing on the threshold of new ways that computers do things, and give us information. Imagine not only being able to watch your favorite sports team play on the Internet but hear the sport commentary as well. This is only a small example of the benefits that speech synthesis technology can bring to human-computer interaction. Seeing these opportunities in speech synthesis and recognition fields have caused the PC industry to put more money and time into developing and understanding the processes behind speech synthesis.

The goal of this project is to understand and implement a general text-to-speech (TTS) system. This paper gives an in-depth account of the process of developing a simple text-to-speech system using Linear Predictive Coding (LPC) coefficients and other signal processing techniques that operate on a DSP EValuation Module (EVM) in a PC.

1.0 Introduction

Avid sports fan know what it is like to be away from the local broadcasts on the day of a favorite team's game. However, the recent addition of game simulation programs on the Internet such as Gamecast from ESPN, allows fans to watch their favorite teams from anywhere with Internet access. Being relatively new, these programs only offer a play-by-play simulation of the game as well as streaming text commentary. They do not offer any sound options to complement the game, and as most sports fans know, the sounds and excitement of the announcers help create the atmosphere. Without them, these games are simply small dots moving aimlessly on the screen. For the current interfaces to offer sound options, network administrators have to send streaming audio in large sound files over the Internet. This requires high bandwidth. As an alternative to the lack of sound on the Internet, users could send sound information as text,

and the receiver could convert the text to sound on their desktop. This is where a Text-to-Speech system may prove beneficial.

1.1Background of Text-to-Speech

The development of speech synthesizers began in the 1930's with Homer Dudley's invention of the Vocoder. Since then, many developments have been made in creating Text-to-Speech programs. Today, with faster and more robust technology, Text-to-Speech has reached the 90% accuracy level. However, to further enhance current systems, researchers consistently go back to basic speech processing techniques. For this reason, to understand the most recently developed text-to-speech systems, one must first understand the basic signal processing techniques behind text-to-speech systems.

1.2 Signal Processing Text-to-Speech Basics

Signal processing techniques in speech communications are fundamentally based on research about the way human voice is fabricated. The overall implementation of text-to-speech is based on sending different forms of energy through a series of filters that are modeled after cross sections of the human vocal tract. To synthesize speech one must first understand how natural speech is produced.

1.2.1 Process of Human Speech Production

Figure 1.1 is an X-Ray picture of the human vocal tract. One can model the vocal tract as a series of tubes where the length and cross-section can be varied to control the amount of air being passed. Speech is simply the acoustic wave that is radiated from this system when air is expelled from the lungs and the resulting flow of air is perturbed by a constriction somewhere in the vocal tract [1] .

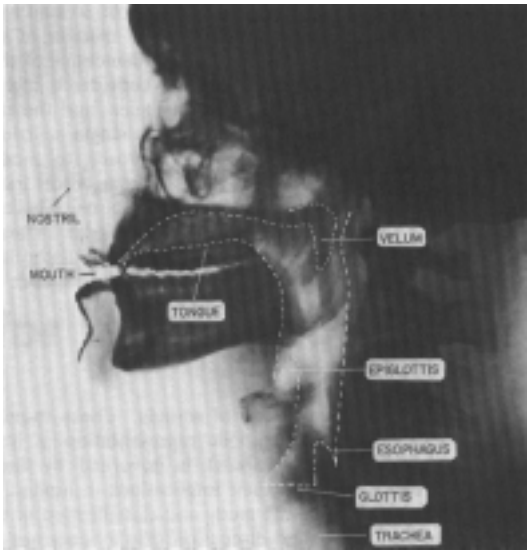


Figure 1.1 X-Ray of Human Vocal Tract

The energy source of the vocal tract system can produce two different inputs: noise or quasi-periodic pulse waveforms. For noise, only a random noise generator and a gain parameter are needed to control the intensity of the excitation. For quasi-periodic pulse waveforms, glottal pulses are used to control the excitation. An impulse train generator can represent glottal pulses spaced by a fundamental period, whose impulse has the shape of a glottal wave. An example of a synthetic glottal wave is seen in figure 1.2. This picture is represented by the equation:

$$g(n) = \_[1-\cos(\pi n/N_1)] \quad \text{for } 0 \le n \le N_1$$
$$= \cos(\pi(n-N_1)/2N_2) \quad \text{for } N_1 \le n \le N_1 + N_2$$
$$= 0 \quad \text{otherwise}$$

N_1 and N_2 are parameters that can be varied [1].

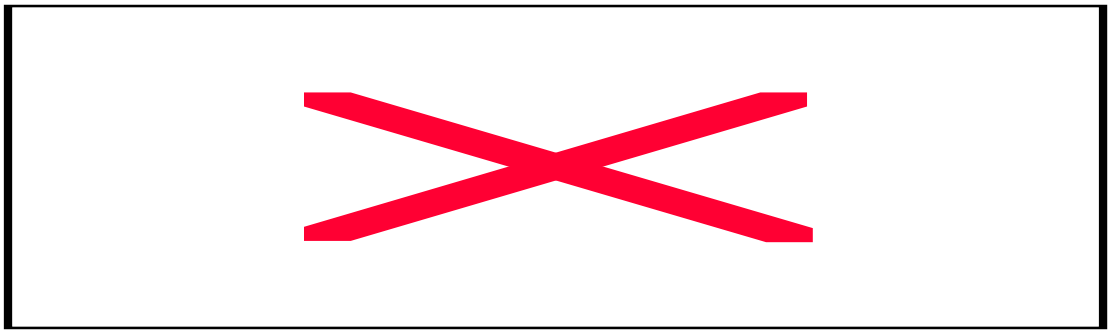


Figure 1.2 Example of Glottal Pulse

The selected input depends on the fundamental characteristics of the language being modeled. Human languages can be described in terms of their fundamental sounds called phonemes. These different phonemes can be classified according to vocal tract positions and

restrictions. The categories are vowels, diphthongs, semi-vowels, and consonants, which consist of nasals, stops, fricatives, affricates, and whisper. (see figure 1.3)

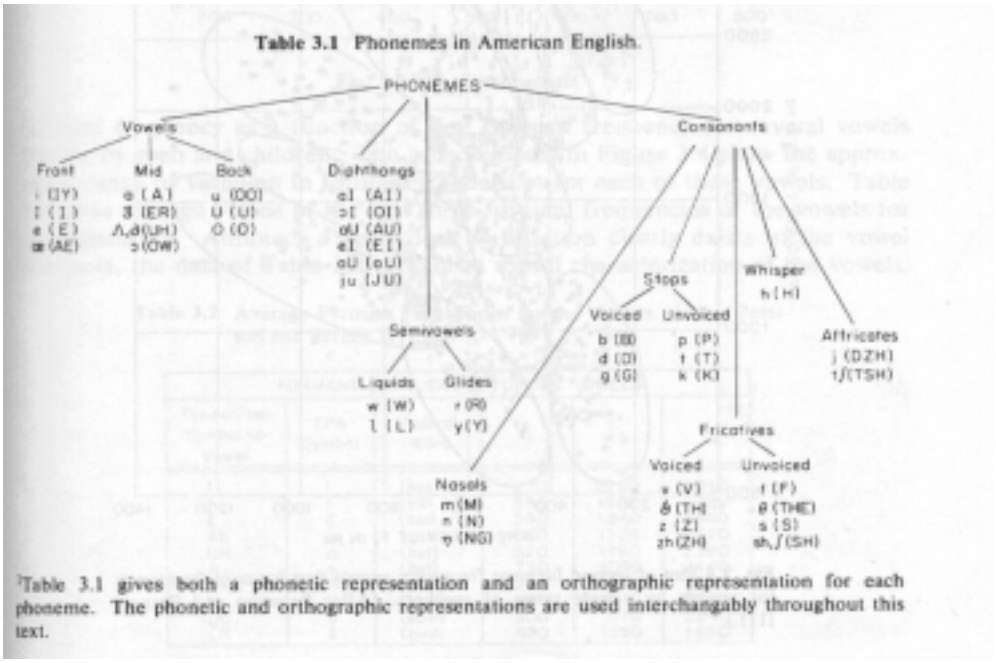


Figure 1.3 Phoneme Hierarchy

Additionally, fixed vocal tract shapes are called continuant sounds, while variable vocal tract shapes are called non-continuants. Phonemes can also be classified by what type of energy passes through the vocal tract. Sending glottal pulses through the vocal tract creates voiced phonemes; sending noise through the vocal tract creates unvoiced phonemes.

It is important to categorize phonemes to determine which input and filter must be used to produce the phoneme sounds. For example, voice sounds require glottal pulses as an input while unvoiced sounds are produced by static noise as input. Continuant phonemes can be modeled with static filters, while non-continuant phonemes will require dynamic filters that change with time. Nasal sounds will require filters with both poles and zeros while other sounds can be typically modeled with all pole filters.

At different points along the vocal tract, the cross-sectional area and length varies. These variations contribute to the resonant frequencies for each sound. Through extensive studies, researches have shown how different parts of the vocal tract affect each sound. For example, sound can be altered by the velum (in the passage from the throat to the nose), which when opened adds a nasal quality to the sound. Constricting parts of the vocal tract with the tongue and/or lips creates differences in the vowels and voiced consonants.

1.2.2 Text-to-Speech Models

Using basic signal processing techniques, there are three common ways to create artificial speech. One method, concatenation uses prerecorded speech and combines it to make whole words. An advantage to this method includes highly accurate speech, but at the expense of memory.

Another technique uses what are called formant frequencies. This technique assumes that the vocal tract is a series of lossless tubes where each tube has a resonant or formant frequency. This method requires taking the first three formants of the given phoneme and applying them to an all pole filter model. The formant frequency technique creates respectable output for some voiced phonemes. However, for sounds like the nasals, which more strictly follow a zero-pole model, formant frequencies give poorer results.

Finally, this project uses LPC coefficients to model each phonetic waveform. This model is similar to the formant frequency technique in that it uses an all-pole filter model to create the phoneme waveforms. However, it is more accurate than formant frequencies because it attempts to predict what each waveform should be based on a series of previous inputs. Since LPC coefficients offer a means to implement a general TTS system and they may help maximize performance and memory, it was deemed appropriate to develop this implementation. LPC coefficients are described in better detail in section 3.2.

2.0 Our Text-to-Speech System

For our Text-to-Speech system, we wanted to create the most realistic system within the allotted time and parameters of this course. Our Text-to-Speech system operates under the following parameters.

1. LPC Coefficients will be used to represent the speech production model.
2. The sampling rate will be 16000 Hz, which is the average rate for higher quality systems.
3. The system uses glottal pulses as an energy source for voiced phonemes.
4. The system uses Rayleigh distributed noise for unvoiced phonemes.
5. The system uses the CMU dictionary to parse words into phonemes. Any word not found in the dictionary is ignored.
6. Input will be prompted on the PC. The program will deal with numbers, but not with complicated symbols and punctuation, i.e., it removes punctuation at the beginning and end of words.
7. Input is limited for polysyllabic words. Only words with syllable information added to the CMU dictionary will be used.

3.0 Implementing a Text-to-Speech System

To create a basic text-to-speech system one needs to create a mathematical model for the human vocal tract. The implemented approach requires extracting LPC coefficients for each

phoneme in the language, modeling the vocal tract with an all-pole filter model (cascade IIR filter) composed of these coefficients, and applying blending techniques to smooth out the discontinuities between individual phones.

3.1 LPC Coefficients

LPC analysis is used in a variety of signal processing fields, and it has many applications in the field of speech processing. It starts with the assumption that speech can be modeled by creating an all-pole filter using the resonant, (formant) frequencies. Although there are a lot of different ways to extract these formant frequencies, LPC is one of the more commonly used since it provides extremely accurate estimates of speech parameters, and is relatively efficient for computation.

3.1.1 Coefficient Extraction

The LPC coefficients represent the specific locations of the poles for the filter model. These coefficients come from trying to predict the speech sample as a linear combination of previous samples. This can be modeled by the equation:

$$\hat{u}(L+1) = -u(L) - a(2)u(L-1) - a(3)u(L-2) - \cdots - a(n+1)u(L-n)$$

u(L+1) is the estimate of the next sequence value and n is the prediction order. The coefficients are outputted in vector form a=[1 a(2) … a(n+1)]. From here, the LPC computes the least-squares solution, which leads to the predictor coefficients, r=[r(1) r(2) … r(n+1)]^T from the matrix

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(n)^* \\ r(2) & r(1) & \ddots & \vdots \\ \vdots & \ddots & \ddots & r(2)^* \\ r(n) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

This solution can be found using pre-programmed functions to extract LPC coefficients that can be found on numerous websites. [2]

To obtain the LPC coefficients, Matlab scripts extracted voice samples from the TIMIT database. This database contains wave files with spoken sentences, which are divided into groups from different regions of the United States. The scripts extracted from a small subset of the database including two to three speakers and a dialect known as “Army Brat”. People who have lived throughout the U.S. characterize this dialect. From these wave files, the scripts extracted a waveform for each phoneme. The LPC function in MATLAB helped extract the first 12 LPC coefficients of each phoneme. These are used to represent the resonant frequencies of each phoneme for the IIR filter design.

3.1.2 Implementation

The LPC coefficients can be used to create an all-pole filter design to model a loss-less tube model. The transfer function is as follows:

$$H(z) = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}} = \frac{1}{A(z)}$$

However, this is a very simple version of a filter. For a better quality filter, one can create a cascade IIR filter by calculating the Bi-Quad coefficients from the LPC coefficients.

3.2 Creating individual phoneme waveforms using IIR filters

IIR filters will be used to implement the speech production model.

3.2.1 IIR Filter Design

IIR filters were implemented using a cascade design. To do so, the LPC coefficients for each phoneme must be converted to Bi-Quad coefficients which will be used in the cascaded IIR filter. The a_k are used in the filter as shown is figure 3.1, below.

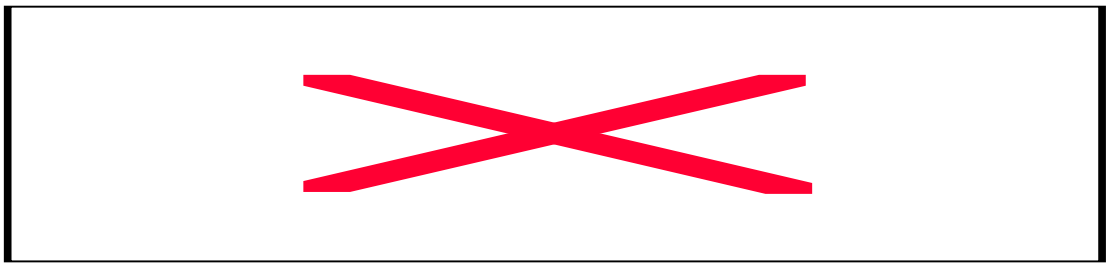


Figure 3.1 Cascade Diagram of IIR Filter

Since each phoneme has its own unique set of coefficients, there will be a unique filter for each phoneme. This design can be represented by the transfer function:

$$\mathbf{H(z) = \frac{1}{\prod_{k=1}^{6} (1 - a_{1k} z^{-1} - a_{2k} z^{-2})}}$$

3.2.2 Phoneme Waveform Design

To create each phoneme, one must pass either a series of glottal pulses or noise through the IIR filter created with the Bi-Quad coefficients. Whether the input is noise or glottal pulses depends on the classification of the phoneme as stated in section 1.2.1.

3.3 Blending techniques

After creating the filters for each of the phonemes, various filtering techniques can be applied to produce more intelligible speech. For our system, two stages of blending were applied.

First, modifications were made to the individual phoneme waveform. Later, neighboring waveforms are blended to improve the transition between them and obtain discrete syllable sounds.

3.3.1 Word Construction

Creating a word takes five steps. First, select the proper energy source (glottal pulse train or noise). Second, find the correct glottal or noise parameters to best represent each sound and scale the phone by the appropriate amount (1 for vowels, .6 for glottal generated consonants, .3 for noise generated consonants). Third, apply accurate decays to create the pitch variations of each phoneme. Next apply a low pass filter to all noise generated phonemes to soften the sounds. Finally one must blend two neighboring phonemes to smooth transitions.

3.3.1.1 Choosing Phoneme Parameters: Voiced Phonemes

As mentioned before, passing an energy source through an IIR filter designed from LPC coefficients that represent each phoneme creates a phoneme waveform. Depending on the phoneme, either glottal pulses or noise are passed through the filter. In our filter design, there were four parameters that varied the glottal pulse, and two to vary the noise. A simple mathematical representation of a glottal pulse is in the equation:

$$
\begin{aligned}
g(n) &= \_[1-\cos(\pi n/N_1)] && \text{for } 0 \le n \le N_1 \\
&= \cos(\pi(n-N_1)/2N_2) && \text{for } N_1 \le n \le N_1 + N_2 \\
&= 0 && \text{otherwise}
\end{aligned}
$$

where N_1 and N_2 are parameters that can be varied. The length of the sound can be varied by how many glottal pulses are put in a row. The spacing between each glottal pulse also affects the sound. These four parameters can be varied to produce different results. Common parameters to model a glottal pulse energy source are choosing N1 = 90 N2 = 44, #samples = 400, and spacing = N1+N2+100. With these values, we created the phonemes necessary for the word evolution. The graph and sound sample are below.

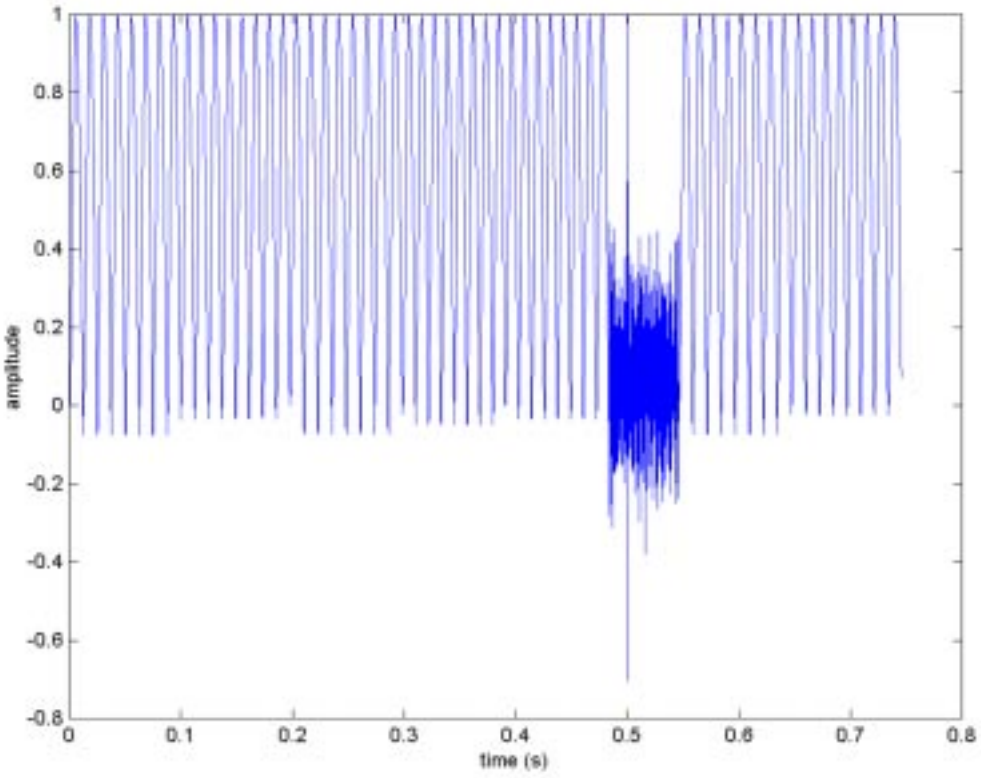


Figure 3.2 Evolution using fixed glottal parameters

Through trial-and-error, it was determined that manipulating the various parameters gives different results. Varying N1 makes the sound more or less nasal, while changing N2 affects the damping in the sound. For some phonemes, which are composed of quick sudden bursts of air, fewer glottal pulses produce more accurate sounds. A table of our parameters that we found most suitable for each sound is attached in Appendix A. The result of using these customized parameters for our word evolution is seen below.

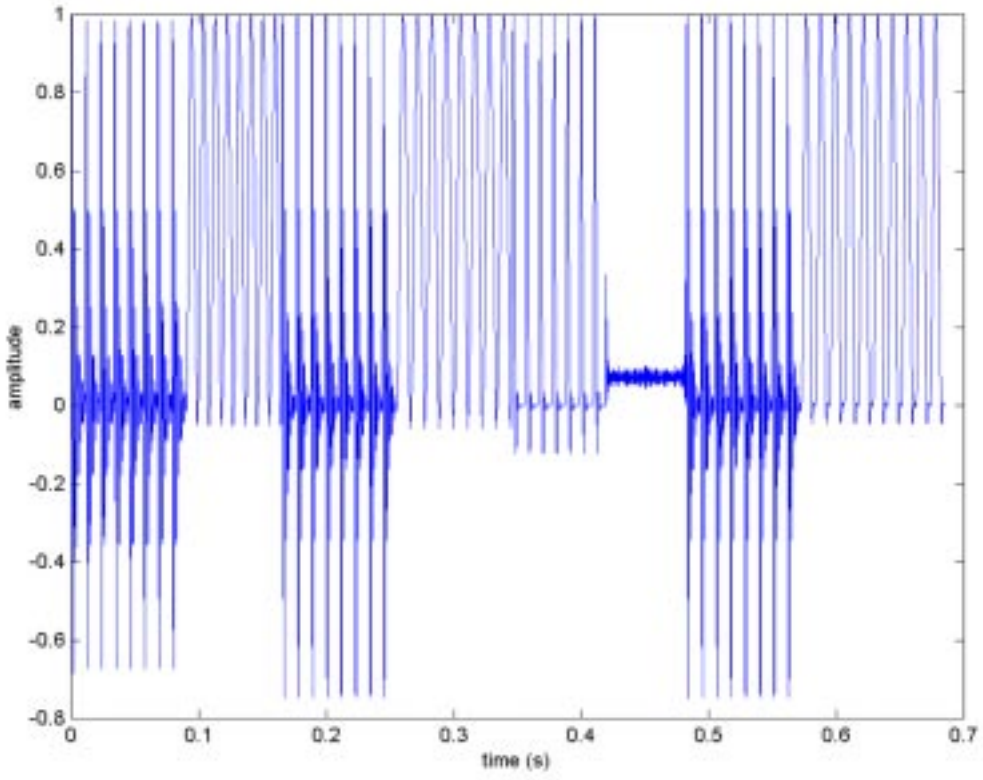


Figure 3.3 Evolution using Customized Parameters

3.3.1.2 Choosing Phoneme Parameters: Unvoiced Phonemes

Using noise as an input from the energy source creates the unvoiced phonemes. An accurate way to represent noise is by an uncorrelated Gaussian distributed noise sequence. This creates different uniform distributed data, which is better than most built in noise generators, where noise is produced in a non-uniform manner. From this data comes the Rayleigh distributed data, which is represented by the equation below.

$R_n = \text{sqrt}(-2\sigma^2 \ln(1-A_n))$

A_n is each value in A, which is a uniformly-distributed sequence in the interval [0 1). σ^2 is the variance. Our values for each phoneme are listed in the chart in Appendix A.

3.3.2 Phoneme Decay

Generally, the phoneme rises in the beginning and decays at the end. To make our phonemes adjust to this rise and decay, we applied a Hann window (named after Julius von Hann, incorrectly referred to as a Hamming window in MATLAB for each phoneme. A Hann window is represented by the equation below.

$$H(x,\tau,\alpha) = \begin{cases} \alpha + (1-\alpha)\cos(\pi\frac{x}{\tau}) & |x| < \tau \\ 0 & \text{else} \end{cases}$$

The windows size was selected such that it was the same as the length of the phoneme. The effects of this to the word evolution are seen below.

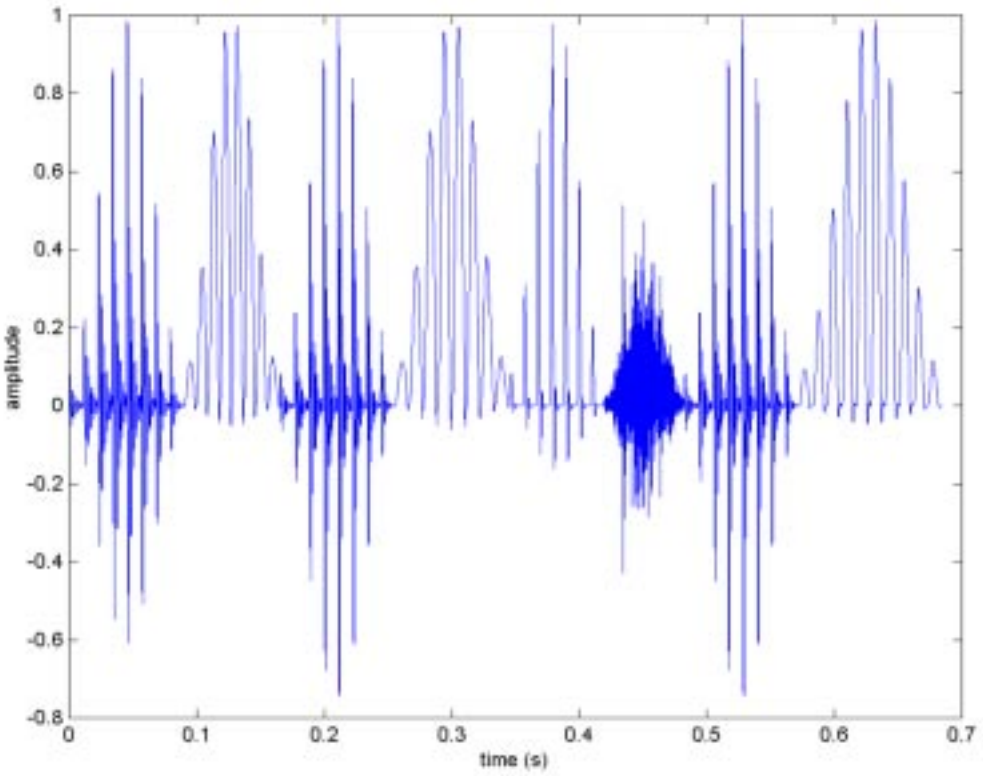


Figure 3.4 Evolution after applying a Hann Window

3.3.3 Problematic Sounds and Solutions

Although the above techniques applied to the phoneme waveforms create better sounding words, there are still some phonemes that needed special attention to gain more accurate results. These special cases comes from the fact that in human speech there are more factors creating audible sounds than those that can be modeled by the above parameters. For example, some sounds contain a combination of both glottal pulses and noise to produce the sound. Also, many of the unvoiced fricatives vary in harshness (compare the sound of the phoneme /t/ to /s/) . These two issues were addressed in fixing some of our phonemes.

3.3.3.1 The Unvoiced Fricatives

When blending voiced with unvoiced fricatives, we found that the harshness of the Rayleigh noise input was causing an overpowering static. After some experimentation we found

that applying a low pass filter removed a lot of the static noise, creating a smoother, more understandable sound. The difference that this makes can be heard in our word evolution.

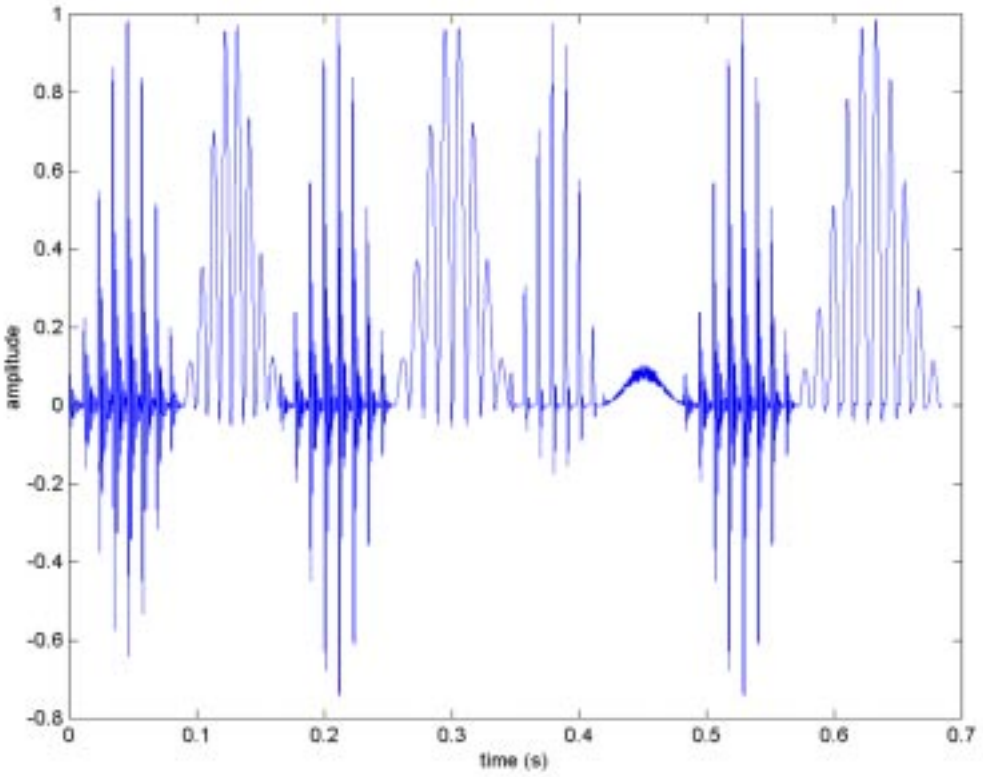


Figure 3.5 Evolution using Low Pass Filter on Noise Phonemes

3.3.3.2 Special phonemes: /b/ and /jh/

For the phonemes /b/ and /jh/ we found that applying our normal filtering techniques did not help in the understandability of these phonemes. For that reason, we resorted to different implementations created specifically for these two phonemes.

For /b/ we downloaded the actual waveform that we originally used to calculate the LPC coefficients, and applied FFT signal processing techniques to observe the input response needed for the /b/. From this, we realized that the /b/ phoneme consists of a short burst of glottal pulse, followed by some noise. Also, the sound is quick and does not require a long duration. Creating a specific input response for this letter created better output.

For /jh/ the same technique was applied. We modeled the input response as a series of glottals that contained small N_1 and N_2 coefficients followed by some noise, with a smaller variance (σ^2). This created a better result as well.

3.3.4 Blending

For our project, we compared two different techniques for blending. The first was Convolution, which a PhD student, Mike Gunia, suggested to use. Our second technique was First Derivative Matching.

3.3.4.1 Convolution

The theory behind the convolution method is that by blending the frequency content of two phonemes mimics the anticipation of human coarticulation when two adjacent phonemes are spoken[8]. Coarticulation is the human tendency to include part of the next phoneme in the phoneme one is currently speaking. To use convolution to model this effect the following equation can be used:

$$s'(n)=S\ a\ FILTER\{s(n)\}b\ s(n-h\ )$$

where s(n) is our original speech signal. Using this technique, we achieved terrible results. The graph and example sound is below. Clearly, convolution did not produce the desired results.

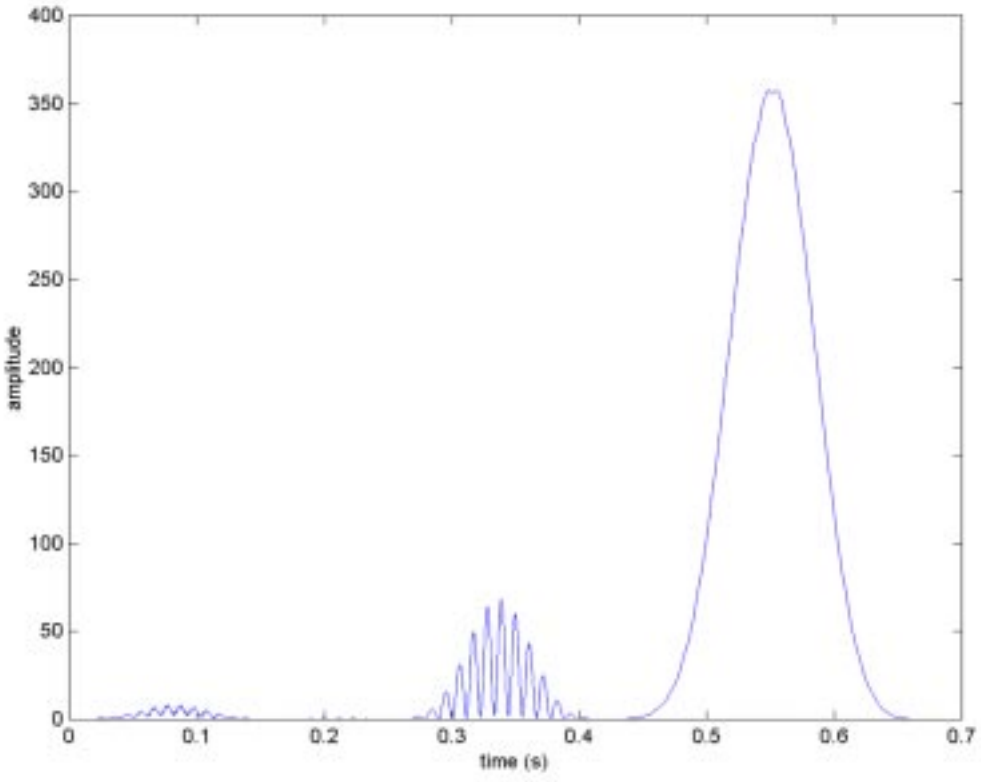


Figure 3.6 Evolution after applying Convolution

3.3.5.2 First Derivative Matching

This procedure is useful for combining two adjacent waveforms. It operates under the assumption that a continuous waveform such as speech has a continuous first order derivative.

The idea behind First Derivative Matching is that if you can find the place where the two derivatives are most nearly equal (shift until the difference between the derivatives is minimized) you have found the place where the slopes of the two waveforms are closest. By shifting the second of the two signals by the amount that minimized the difference between the derivatives, one finds the point where the transition between the two phones is smoothest. A difference equation calculates the first derivative for two adjacent waveforms. One problem that arises is how far to look into each waveform before too much of a waveform is superimposed onto another. If too large a shift is made, then the two phones completely overlap, creating a jumbled sound. We found that by limiting the shift to a range of 200 to 700 (12.5 ms to 43.75 ms @ 16 kHz) samples worked best. A sample of first derivative matching applied to the word evolution is below.

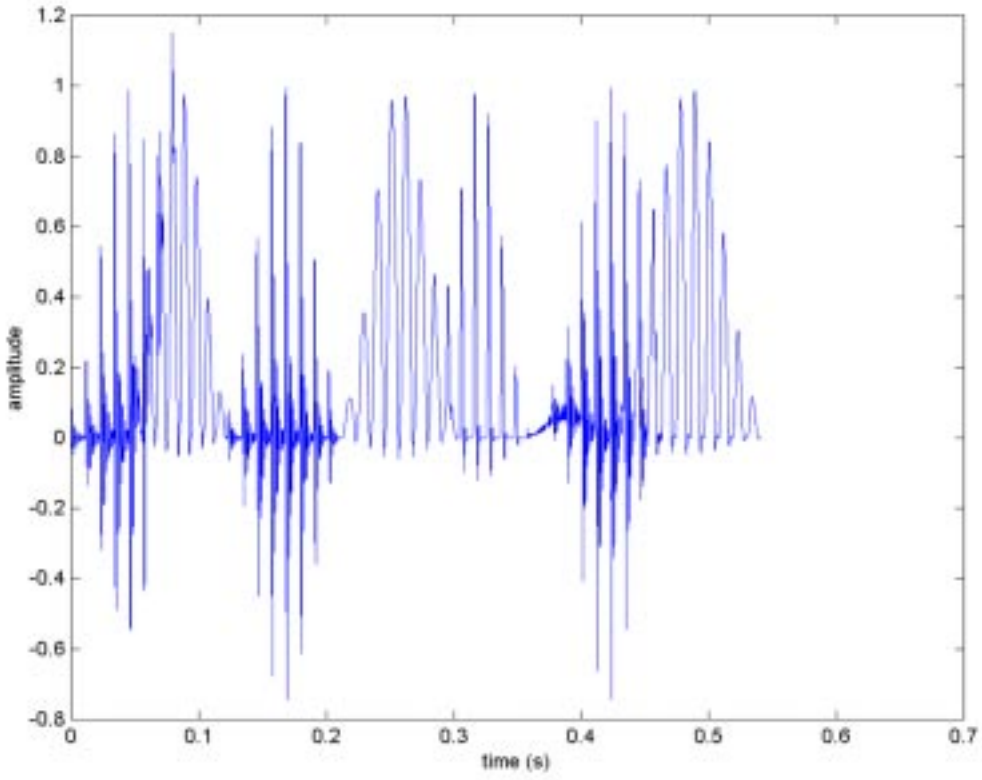


Figure 3.7 Evolution after applying First Derivative Matching

3.3.6 Scaling Phones

As a final touch, weights are applies to each phone to make them sound more like they would in a real word. Consonants are almost always weaker than vowels, so all glottal consonants were scaled by .6. All noise-generated consonants were scaled by .3. The results of scaling in the word evolution are show below.

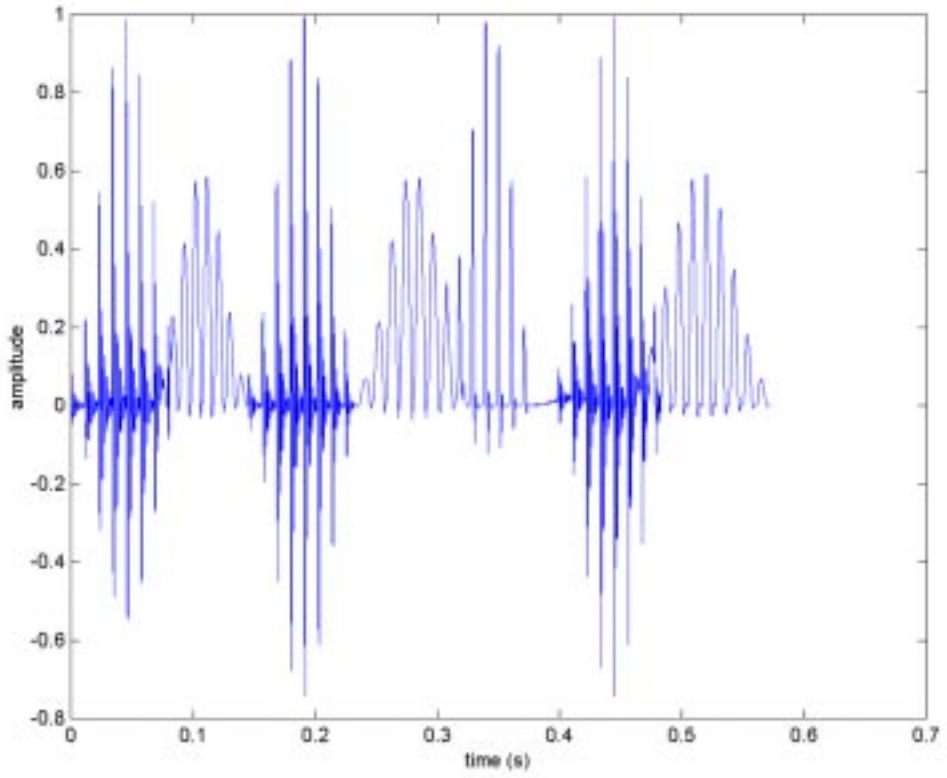


Figure 3.8 Evolution after scaling phonemes

3.4 Methods for Improving Sounds

Problems arise when attempting to find a general model for any individual phoneme. Phonemes have different sounds depending on their location in a sentence. One way to come up with better sounding words would be to come up with different models that depend on a phoneme's location in a word. Other phonemes, like the /b/ sound have different stages in their sound. Energy is collected and released suddenly with a softer period following. In such cases, one could improve performance by selecting two sets of LPC coefficients. One from the portion of the sound where the initial burst is being released, and the second from decaying portion. Another trick might be attempting to classify more phonemes. Some sounds, like the /s/, take in different characteristics in depending on where they are in the word. Classifying two different /s/ phonemes could improve the sound of words.

It seems that using multiple sets of LPC coefficients will yield more accurate sounds. While this may be the case, that this project seeks a general implementation of a TTS systems. In limiting each phoneme to one set of coefficients it is asserted that only one filter model for each phoneme is necessary. Additionally, if many sets of coefficients are used, one approaches the

point where they are storing the same amount of data that actually exists in a phoneme. This suggests that concatenation of phonemes might be attractive in some or even all cases.

4.0 System Level Implementation

Our system's structure is seen in figure 4.1. The user inputs the text at a prompt. The PC then looks up the phoneme content of each of the words entered and sends their phoneme breakup to the EVM board. The EVM then applies our filtering techniques and blends the phonemes into words, which it then outputs through the codec.

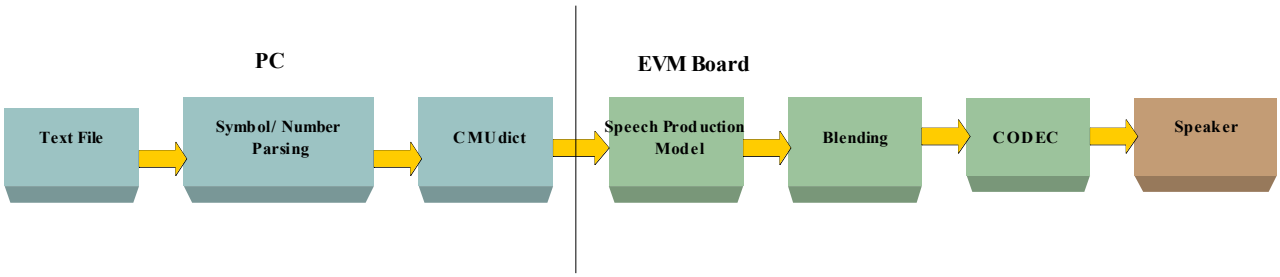


Figure 4.1 System Diagram

4.1 PC Component

The PC receives input from a prompt, looks up phoneme data for each word, and sends the phoneme sequence to the EVM. We use the CMU dictionary from speech.cs.cmu.edu as a source for word to phoneme data. The CMU dictionary contains about 127,000 words and their comprising phonemes in a text format. The database is about 3.5MB. Our program reads through the dictionary and builds a hash table for quick lookups. The load process takes about five seconds. When the dictionary is loaded, the PC reads input and looks up each word, then communicates with the EVM to send the phoneme sequence. The PC strips punctuation and converts decimal numbers into their spelled-out form.

We found that blending by syllable rather than by word improves quality quite a bit. The CMU dictionary does not contain syllable information, however, so we decided to insert it as periods between phonemes. It would have been impossible to insert syllable information for all of the words in the dictionary, but we did insert it for specific words that we wanted to demonstrate.

We had hoped to implement lookups on the EVM, since in that case the PC would only have to send text to the EVM. However, when we considered ways of doing that it became more complicated than we expected. The hash table could not be sent directly since it uses linked lists and thus pointers and arbitrary memory locations. The text file could be sent to the EVM

where the board builds its own hash table, but that would be a waste of memory, since a copy of the 3.5MB database would have to be stored while the EVM builds the lookup table. A more realistic way to do it would be to build the database into a contiguous format that somehow facilitates quick lookups, and then send that database to the EVM. It would have taken us time to implement this, and since we had a working hash table we decided to leave word lookups on the PC.

4.2 PC EVM Communication

PC and EVM communication in our project is not particularly complicated. When the EVM is ready for input, it requests a transfer over the HPI from the PC. The PC prompts the user, looks up the comprising phonemes, and sends those as integer representations to the EVM.

4.3 Speed

We were rather surprised by how long it takes the EVM to generate speech. This becomes less surprising when one considers how many floating point math functions are called. For voiced phonemes, the glottal pulse function involves one cosf call for each input sample. For noise functions, there is one rand and one log call for each input sample. Further, each phoneme is windowed by a Hann function, which also involves a cosf function for each sample. To test speed, we tried the example "the evolution of five fifty one." Without any optimizations, it took an average of 22.6 million cycles per syllable. If one assumes a cycle time of 40ns, this is 0.90s per syllable - slower than real-time. To reduce this, we took note of the fact that all of the unvoiced phonemes use the same sigma value for noise. We modified the program to generate an array of noise when it loads, and use that array over and over rather than generate it each time with rand and log. After this modification, the example took 17.5 million cycles per syllable.

To further test this optimization, we used the example "sally sells seashells by the seashore," which involves the "s" and "sh" phonemes, both unvoiced, to great extent. Before optimization, the program took an average of 36.0 million cycles to generate each syllable. After optimization, it took 25.4 million.

Glottal pulses could not be optimized the same way because each phoneme has different glottal parameters. There would not be enough on-chip memory to store pre-generated input for all of the phonemes, but it could be stored in DRAM. We did not experiment to see how much faster pre-generating all input and storing it in DRAM would be than a cosine call.

Another speed impediment we noticed was that the math library was off-chip. We moved the libraries off-chip so that our program, which turned out to be about 32k, would fit. Each sample required either two or three calls to off-chip math functions. We tried moving the

math library, rtslib, back on chip but found that we were 64 bytes too large to fit on chip. After a witch-hunt for stray printf statements failed to get rid of 64 bytes, we decided to move our noise pre-generation function, which is only called once at program load, off chip. Moving rtslib back on-chip improved things considerably. "The evolution of five fifty one" took an average of 6.68 million cycles per syllable, and "sally sells seashells by the seashore" took an average of 9.39 million. 9.39 million cycles is about equivalent to 0.375s, much closer to the time it would actually take to speak a syllable.

We could also pre-generate Hann windows. There are some phoneme lengths that are common between similar phonemes. For example, many of the vowel phonemes are the same length, 1440 samples. We could achieve time savings by uses pre-generated Hann windows for the most common lengths. We did not have time to properly implement this. We expect that it would produce results similar to, but not as spectacular as pre-generating noise, because it would be saving only a single cosine call rather than a rand and a log call.

4.3 Memory Allocation

Our program uses on-chip memory as much as possible. Our own code came to be about 39k of on-chip program memory. We could not fit library functions into on-chip space, so we moved them into board memory. Our goal was to keep as much data in on-chip memory as possible. We used global variables while working on syllables and phonemes. A limitation of this is that a syllable must be shorter than a defined length - we used 8000, or half a second. This short length created some problems for us. If we did not insert syllable information for words, the program treated the whole word as a single syllable. Many words are longer than half a second, and these words would crash the program. We were not too worried about this, because we were assuming that in a final implementation, all of the words would be split by syllable. Nevertheless, we had spare on-chip data memory, and a longer syllable buffer would have prevented this problem in many cases.

The syllable is built on-chip, and when complete it is transferred to a buffer for the whole input in board DRAM. This buffer holds 10 seconds of 16000Hz stereo. Since the length is so short, we decided trying to optimize this with DMA would not be worthwhile. The codec transmit ISR reads from this buffer. With our design, the DRAM buffer is only accessed twice - once for writing, once for reading, so the slowness of the DRAM should not play much of a role.

4.4 Limitations to our system

Even though the program should be able to convert any text to speech, there are some limitations to it.

4.4.1 CMU Dictionary

The CMU dictionary contains over 100,000 words found in the English Dictionary and their phoneme breakup. Since this gives the PC the needed phonemic breakup of the words, if a word is not found in the dictionary, the program cannot output it. Since our pre-selected demo consists of certain names not included in the dictionary, those were manually added.

4.4.2 Syllable Limitations

Blending should only be used on phonemes within the same syllable. If all phonemes within a word are blended together, smearing occurs in the speech, and outputs poorer results. For this reason, we need to differentiate between different syllables. Since the CMU dictionary does not contain syllable information within its phoneme breakup, it became necessary to add syllable separations within the dictionary to prevent the program from blending all phonemes through neighboring syllables. Since this would be tedious and unnecessary to go through the entire CMU dictionary, this information was added only to words used in the demo. For this reason, if one inputs a polysyllabic word in the input that has not been adjusted, the word will not contain breaks within different syllables.

5.0 Testing

We applied many different words of various difficulties to our system. We chose words that contained different combinations of voiced and unvoiced phonemes, different syllable lengths, and words of various speaking complexity. Because of limitations with regard to the lack of syllable information in the CMU dictionary, many multiple syllable words could not be tested, since this requires manual alteration of the dictionary. Overall, approximately 60 words were tested on the system.

5.1Demo

Our demo consists of two parts. The first part models the application that we chose for our text-to-speech system: streaming text from Gamecast. At the prompt we entered pre-selected text from a variety of sports games found in Gamecast.

The second part of our demo allows the user to input any stream of text that they choose at the prompt as long as it is within the limitations of our program**.**

5.2 Performance

The program does not produce anything that one would call natural sounding speech, but it is intelligible, albeit a very stereotypical "computer" voice. It does help quite a bit to know

what word the program is trying to say. Adding greater pauses between syllables and words has made them more distinguishable. One of the greatest impediments to the ultimate quality is the fact that the system uses LPC to model phonemes. Other speech processing techniques could probably produce better results. Each of the steps taken, including: customizing glottal parameters, blending, windowing, and low pass filtering unvoiced phonemes, greatly improved the quality of the output, as demonstrated in our presentation.

5.3 Results

As stated above, some words were recognizable, and others were not. We also concluded that most of the problems stem from the fact that we used LPC coefficients to describe our different phonemes. Since LPC is a prediction analysis, combining information from all previous parts of the waveform to produce the next output, phonemes that consisted of large variation of the vocal tract posed some difficulty. We found that phonemes such as the vowels, where their sound is constructed from a fixed vocal tract that does not vary much over time, were better sounding than stops, which are formed by building up pressure behind a constriction in a vocal tract and then suddenly releasing the pressure. Phonemes such as stops require a great variance from the initial waveform shape to that at the end of the waveform. For this reason, it proves difficult to represent the variable phonemes with LPC coefficients.

6.0 Conclusion

Overall, we feel that we accomplished our two goals successfully. First, we feel that we created a well working system using LPC coefficients. We applied many different types of blending techniques as well as adjustments to create respectable computerized speech considering that we were working with LPC coefficients. We also feel through our research, development, and implementation of different filtering techniques we gained insight into the signal processing techniques behind speech synthesis.

6.1 Improvements/Future Work

Since no text-to-speech system was implemented in this course before, we feel that there are many improvements that can be made to our system.

Building a good speech synthesizer is a very difficult task. We knew this beginning the project, but actually trying it helped us realize that more clearly. If we had looser time restraints, there are a number of things we could do to improve the program. One impediment we had to deal with was the limited quality of LPC. There are better speech processing techniques such as Cepstral, which would yield better output. We could also split phonemes up into even

smaller units. Several filters rather than one could better represent many phonemes. For example, "b" begins with building up voicing, then releasing it suddenly. There is a short rumble before we release the air. A "b" sound can be cut at the beginning to sound exactly like a "p," which does not have the buildup. In fact, our program produced a "p"-like sound rather than a "b." That and many other phonemes could be better represented by several filters representing different parts.

Most research today focuses on concatenation - taking real speech samples and blending them together, rather than trying to generate them with LPC. Rather than using a transfer function to generate sounds, we could use real speech samples. It would take quite a bit of time to properly gather samples and generate the diphones (transitions between phonemes), but there is a good potential for excellent results.

We had hoped to have time to include pitch in our final program. The CMU dictionary contains stress information for words, and there are general rules we could apply, such as emphasize the beginning of the sentence and then die off. Adjusting pitch would have made our program sound less computerized.

7.0 References

1. LR Rabiner & RW Schafer, *Digital Processing of Speech Signals*
2. http://www.mathworks.com/access/helpdesk/help/toolbox/dspblks/lpc.shtml -LPC coefficients explanation of autocorrelation method
3. http://ee.mokwon.ac.kr/~jspark/rtp/speech/node38.html -LPC coefficient equations
4. http://asylum.sf.ca.us/pub/u/howitt/lpc.tutorial.html - LPC coefficients and speech
5. http://www.speech.cs.cmu.edu - CMU Dictionary
6. The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus Training and Test Data NIST Speech Disc CD1-1.1
7. Jonathan Allen, *From Text to Speech:The MITALK System*
8. http://www.gerc.eng.ufl.edu/STUDENTS/GuniaM/test1.htm - Blending techniques
9. Sullivan, Tom. Ph.D. Carnegie Mellon University
10. Oppenheim, Alan V, Alan S Willsky. Signals and Systems 2nd edition. Prentice Hall, New Jersey. 1997.
11. Sklar, Bernard. Digital Communications: Fundamentals and Applications. Prentice Hall, New Jersey 1988.
12.

Appendix A:
Phoneme Data spreadsheet

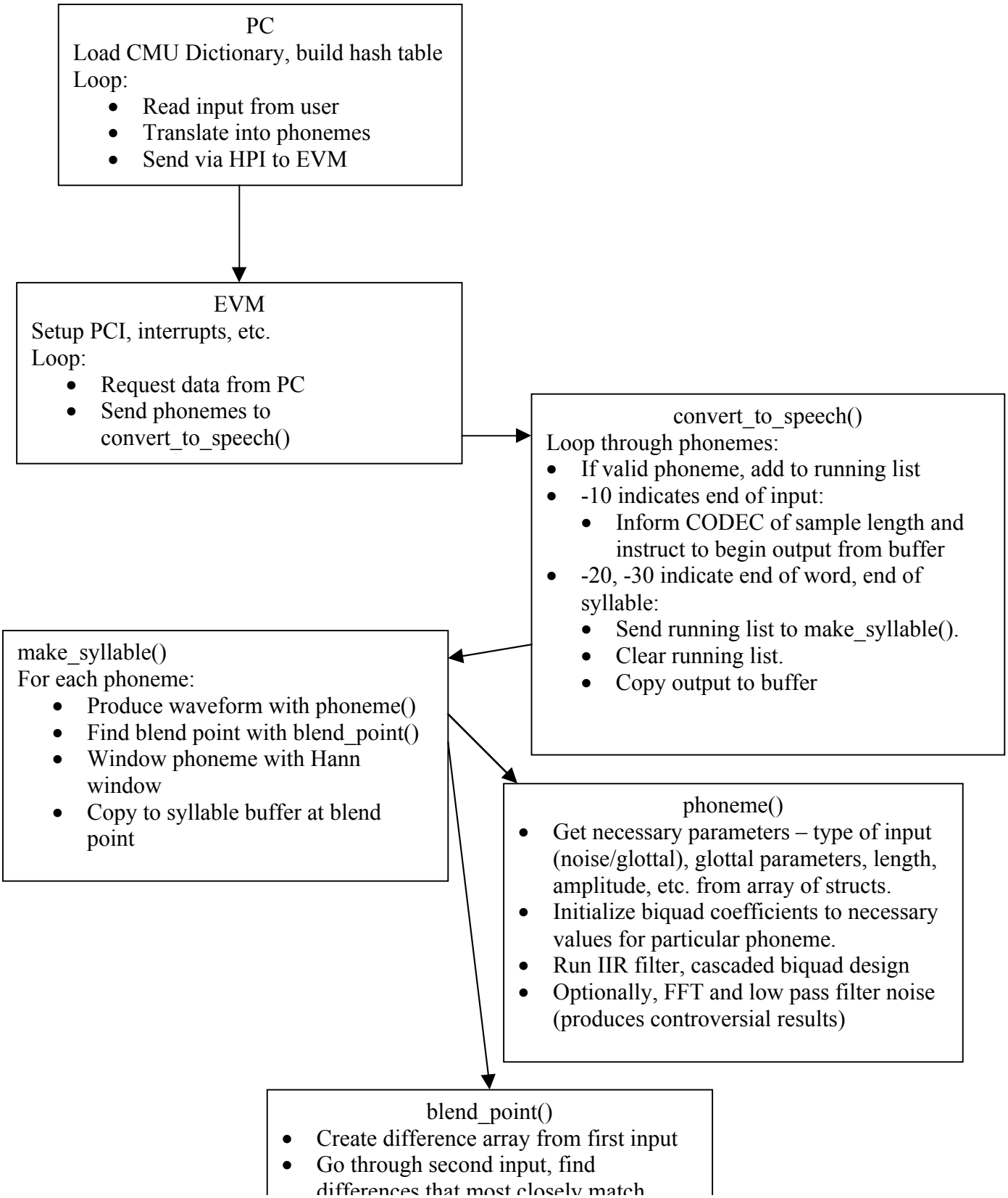
Phoneme	word	N1	N2	GP/ NOISE	Length Parameter	#glottal	length	Special Notes
AA	odd AA D	6	12	GP	180	8	1440	
AE	at AE T	10	2	GP	180	8	1440	
AH	hut HH AH T	10	2	GP	180	8	1440	
AO	ought AO T	12	12	GP	180	8	1440	
AW	cow K AW	12	2	GP	180	8	1440	
AY	hide HH AY D	70	55	GP	180	8	1440	
B	be B iy			NOISE	201	1	201	inp(1) = 1.0660, inp(2:201) = Rayleigh(.0303, 200)
CH	cheese CH IY Z			NOISE	1000	1	1000	Rayleigh(1,1000)
D	dee D IY	20	60	GP	180	8	1440	
DH	thee DH IY	160	0	GP	180	8	1440	
EH	Ed EH D	2	60	GP	180	8	1440	
ER	hurt HH ER T	20	2	GP	180	8	1440	
EY	ate EY T	180	0	GP	180	8	1440	
F	fee F IY			NOISE	750		0	take fft. Ifft 1:150 zeros 1:850)
G	green G R IY N	30	40	GP	180	8	1440	
HH	he HH IY			NOISE	800	1	800	take fft. Ifft 1:150 zeros 1:850)
IH	it IH T	25	140	GP	180	8	1440	
IY	eat IY T	45	140	GP	180	12	2160	
JH	gee JH IY	2	2	GP & NOISE	180		0	gpt(2,2,180,10) + Rayleigh(20*.0331, 180*10)
K	key K IY	0	0	NOISE	1000	1	1000	take fft. Ifft 1:150 zeros 1:850)
L	lee L IY	80	1	GP	180	8	1440	
M	me M IY	60	45	GP	180	10	1800	
N	knee N IY	60	45	GP	180	10	1800	
NG	ping P IH NG	60	45	GP	180	10	1800	
OW	oat OW T	100	70	GP	180	8	1440	
OY	toy T OY	50	80	GP	180	8	1440	
P	Pee P IY			NOISE	1000	1	1000	take fft. Ifft 1:150 zeros 1:850)
R	read R IY D	35	15	GP	180	10	1800	
S	sea S IY			NOISE	900	1	900	take fft. Ifft 1:150 zeros 1:850)
SH	she SH IY			NOISE	1000	1	1000	take fft. Ifft 1:150 zeros 1:850)
T	tee T IY			NOISE	320	1	320	take fft. Ifft 1:150 zeros 1:850)

TH	theta TH EY T AH			NOISE	800	1	800	take fft. Ifft 1:150 zeros 1:850)
UH	hood HH UH D	35	35	GP	190	8	1520	take fft. Ifft 1:150 zeros 1:850)
UW	two T UW	20	50	GP	170	7	1190	
V	vee V IY	60	35	GP	150	8	1200	
W	we W IY	30	15	GP	160	8	1280	
Y	yield Y IY L D	50	15	GP	160	8	1280	
Z	zee Z IY			NOISE	800	1	800	take fft. Ifft 1:150 zeros 1:850)
ZH	seizure S IY ZH ER			NOISE	700	1	700	take fft. Ifft 1:150 zeros 1:850)

Contains parameters used for each phoneme

Appendix B:

Code Organization with Specific Examples



```
void convert_to_speech(int *p, int *buf) {
  /* convert text to speech, word by word */
  /* put results for each word into buf */
  /* output on codec */
  int s_l,i,j,k,buf_idx=0;
  int sample;
  int phone,phones[10],phones_count=0;

  for(i = 0; ; i++) {
    phone=p[i];
    if(phone == -30 || phone == -20) {  /* flush syllable */
      s_l = make_syllable(phones,phones_count);
      phones_count = 0;
      while(buffer_len > buffer_idx);
      for(j=0; j < s_l; j++) { /* copy syllable to buffer */
        sample = (int)10000*syllable[j];
        sample = (sample & 0xFFFF) | (sample << 16);
        buffer[buf_idx++] = sample;
      }
      for(k=0;k<1000;k++) buffer[buf_idx++] = 0;
    } else if(phone == -10) {  /* end of text - output */
      buffer_len = buf_idx;
      buffer_idx = 0;
      break;
    } else {
        phones[phones_count++] = phone;
     }
  }
}



int make_syllable(int *p, int count) {
  int phone_code, l;
  int i,j,s_l=0,b;

  if(!pdat_init) {
    init_p_data(pdat);
    pdat_init = 1;
  }
  if(!noise_pregen) {
    pregen_noise();
    noise_pregen = 1;
  }
  for(i = 0; i < SYLBUFF_LEN; i++) syllable[i] = 0;

  for(i = 0; i < count; i++) {
        phone_code = p[i]/10;
    l = pdat[phone_code].l;
    phoneme(phone, pdat, 10*phone_code);
    b = blend_point(syllable, phone, s_l, l);
    window(phone, W_HANN, l);
    s_l += l - (s_l - b);
    for(j = 0; j < l; j++)  syllable[b++] += phone[j];
  }
  return s_l;
}
```

```
float diff[MAX_SHIFT-MIN_SHIFT];

int blend_point(float *p1, float *p2, int l1, int l2) {
  int i,max,min_i=0;
  float min_diff = 100000;
  float d;

  if(l1 < MIN_SHIFT || l2 < MIN_SHIFT) return l1;
  if(l1 < MAX_SHIFT || l2 < MAX_SHIFT) {
    max = (l1 < l2) ? l1-1 : l2-1;
  } else max = MAX_SHIFT;

  for(i = l1 - max; i < l1 - MIN_SHIFT; i++) {
    diff[i] = p1[i+1] - p1[i];
  }
  for(i = 0; i < max - MIN_SHIFT; i++) {
    d = p2[i+1] - p2[i] - diff[i];
    d = (d < 0) ? -d : d;
    if(d < min_diff) {
      min_diff = d;
      min_i = i;
    }
  }

  return l1 - min_i - MIN_SHIFT;
}
```