Speech Analysis and Modification
A Vowel Detection and Replacement Tool

Dalia El-Shimy Antonio Sánchez
Eric Simpson

Dr. Saeed Gazor

Speech Analysis and Modification

A Vowel Detection and Replacement Tool

April 9th, 2007

Investigators:

Dalia El-Shimy, 489 3440

Antonio Sánchez, 496 7007

Eric Simpson, 482 8602

Supervisor:

Dr. Saeed Gazor

Acknowledgements

We would like to thank Dr. Saeed Gazor for his continued support throughout this project, for being a team member, and for encouraging us to learn and explore on our own.

We would also like to thank all those who keep us inspired, as well as those who provide much needed relief from the seemingly endless amount of work we have had to bear this year.

Abstract

The Speech Analysis and Modification (SAM) System is a vowel detection and replacement software tool. It was developed to assist researchers in the Department of Psychology at Queen's University in their study of psychoacoustics. The program is designed to run on any personal computer with a microphone, running a Windows operating system.

The development of the tool is, in itself, a study of pattern detection in voice data, as well as of digital encoding and filtering techniques. The problem not only required research into the theory of digital signal processing, but also required the study of formants and periodicities in the frequency spectrum of voiced speech. In addition, some understanding of the mechanism of the human ear was necessary to determine the important information contained in speech. Using characteristics of the poles obtained from an autoregressive (AR) model produced by Linear Predictive Coding (LPC), a method was devised to represent each sound as a vector that encompasses its relevant properties. Subsequently, a metric was designed to compare the vectors representing segments of speech to a library of vowels. The metric took into account the power at certain frequencies, estimated from the poles in the AR model. Finally, using these AR filter models, vowel sounds were successfully synthesized from other vowel sounds, while preserving the speaker's tone and volume.

Both the training and main algorithm were first designed and tested in MATLAB. After satisfactory performance, all methods were coded in C++. A user-friendly graphical interface was also created to give the researcher control of all experimental parameters, as well as to display relevant characteristics of the speech under study.

Contents

List of Figures

1 Introduction

1.1 Purpose

The goal of the project is to develop a vowel replacement tool to be used in psychoacoustic studies. When the participant speaks, the algorithm will attempt to detect a specified vowel, and subsequently replace it with another vowel sound. The modified signal will be played back to the subject in real-time, prompting an adjustment to his or her speech. In turn, psychology researchers may make use of collected parameters, such as the reaction time and the number of attempts before correction.

1.2 Motivation

To this day, the problem of speech recognition has not fully been solved. While the human brain is able to interpret words spoken by different people, with different accents, and different volume levels quite easily, designing a robust system to attempt the same tasks has proved to be extremely challenging. The reason is that the waveforms produced for a certain sound, even when spoken by the same person and with the same tone, look very different. An example of this is shown in Figure 1.1. It is believed that in order to improve these systems, further studies must be conducted to see how the brain perceives and interprets different sounds. This knowledge can be applied to speech coding and compression algorithms, in order to send only the important information contained in the human voice.

1.3 Objectives

The main objective of this project is to design a tool in C++ that can accurately detect two vowel sounds, 'AH' and 'EE', exchange these vowels in real-time, and play them back to a user. The detection of vowels is much simpler than attempting to design a full speech recognition system; vowels are highly periodic, and one can exploit this property to distinguish them.

Since the tool is intented to function in real-time, the algorithms designed must be very efficient, and relatively simple. There is therefore a trade-off between complexity and accuracy. SAM is designed to have a maximum delay of 16 milliseconds, corresponding to 128 samples speech sampled at 8 kHz. Thus, the algorithms are designed with a complexity to maximize accuracy, while maintaining this time constraint.

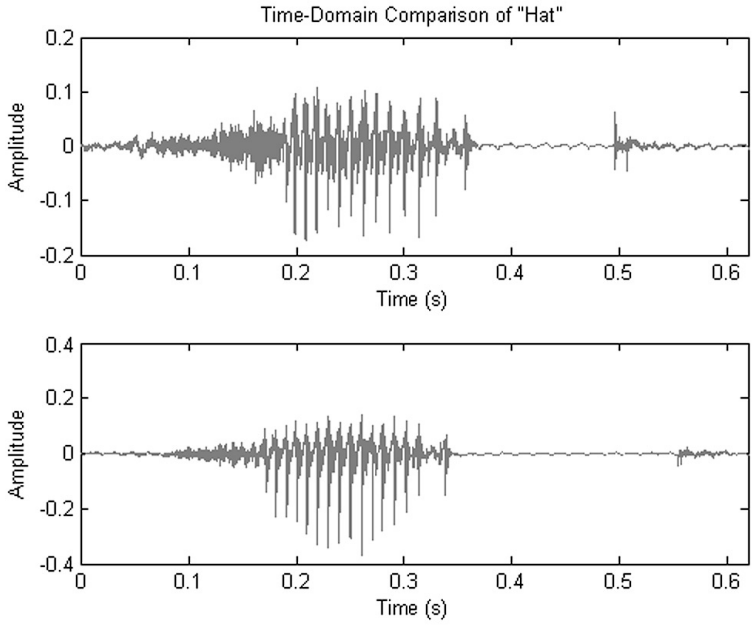


Figure 1.1: The time-domain samples are plotted for two different instances of the word "hat". The sounds are indistinguishable by ear, but the waveforms look quite different.

A secondary objective is to create a friendly user-interface that allows one to customize the detection parameters, such as which vowels should be detected, which ones should be replaced, and which vowels they should be replaced with.

1.4 System Overview

The proposed system model for the core of tool is shown in Figure 1.2. The first stage is the system training mode that attempts to characterize the speaker's voice. After training, the tool will enter its normal running mode. Here, the voice processor will first convert the analog signal from the microphone into a digital form that can be processed. The digital signal, divided into fixed-length segments, is sent along two paths: the first is sent to a spectral analyzer to determine the frequency content of the sound, and the second leads to a short delay routine that will compensate for the time required to process the signal. If the spectral analyzer detects one of the predefined vowel sounds in a segment based on frequency content, it will signal the vowel replacement routine to perform an exchange. The replacement routine will select the specified alternative vowel from the library, and subsequently synthesize the

new speech signal. If the spectral analyzer does not detect a target, the system will transmit the delayed original signal. The digital output will then be converted to an analog signal that is fed to the speakers.

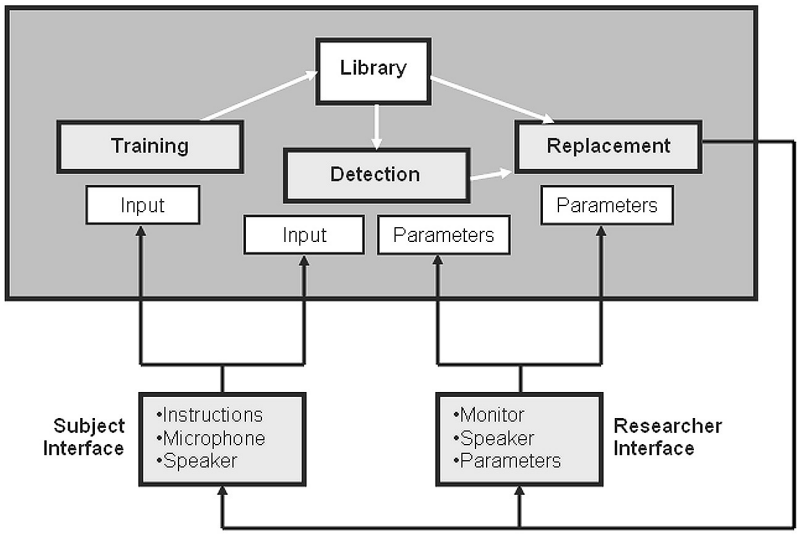


Figure 1.2: Overview of software tool

2 Background

2.1 Psychoacoustics

The main application for the Speech Analysis and Modification project is to aid research in the field of psychoacoustics. Psychoacoustics can be defined as the study of the relationship between physical sounds and the brain's interpretation of them. One current area of research in psychoacoustics investigates the interaction between a person's speech and his or her auditory system. For such studies, a device is needed to slightly alter a participant's speech in order to observe the brain's reaction upon hearing an unexpected sound. Through comparison, the auditory system acts as a natural feedback system to the brain's speech center. Based on the information received, adjustments to various speech parameters, such as word pronunciation and volume, can be made accordingly.

2.2 The Human Ear

The human ear acts like a natural filter and frequency analyzer. A diagram of the human ear is shown in Figure 2.1. The coiled-shaped cochlea in the ear is filled with a fluid that surrounds tiny hair cells. When the ear is stimulated by sound waves in the frequency range of 20 Hz to 20 kHz, the hair cells are forced to vibrate. They convert this motion into electrical signals that are transmitted to the primary auditory neurons in the brain. These neurons act like analog-to-digital converts, creating action potentials (voltage spikes) which are sent down the auditory nerve to the brain, which interprets them and distinguishes between sounds.

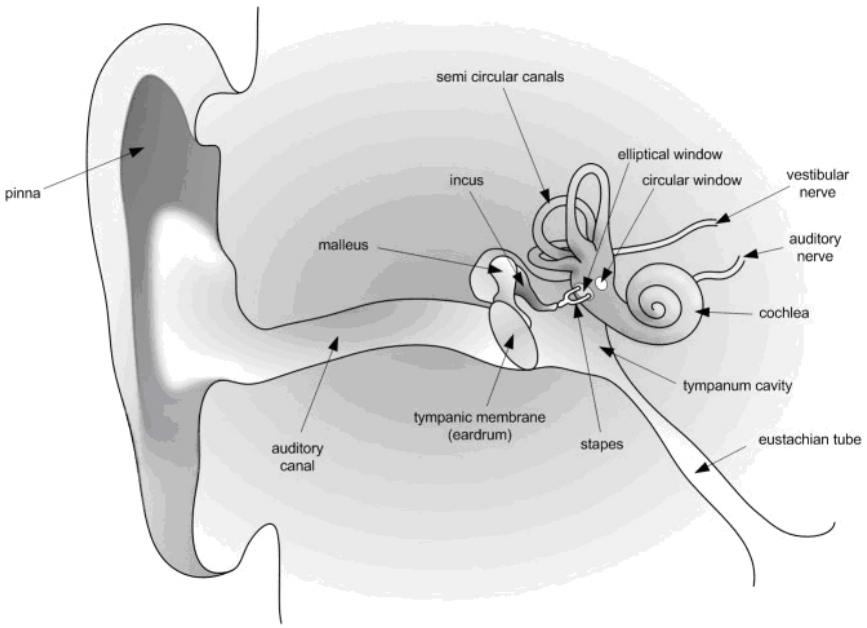


Figure 2.1: Diagram of the human ear [3]

2.3 Vowel Formants

Speech is produced when air is forced through the vocal chords in the voice box, causing them to vibrate at a certain frequency. This frequency is known as the fundamental frequency, and determines the pitch of the voice. Various cavities in the throat, mouth and nose act as resonators, amplifying the vibrations.

Vowels are generated by holding the resonators in a certain configuration, and allowing the escape of sound through an open mouth. This differs from the formation of consonants, which are produced by obstructing or constricting air-flow. Therefore, vowels are periodic signals, and their harmonic power spectrum can be examined.

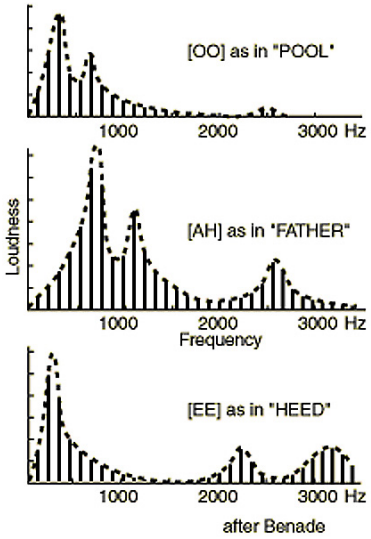


Figure 2.2: The vowel formants are the frequencies at which the power peaks [2]

The amplitude of the harmonic spectrum is greatest near resonant frequencies, known as formants, which are produced by the vocal tract and are specific to each vowel [2]. These formants remain constant even when voice pitch changes. A shift in the fundamental frequency only modifies the spacing between harmonics. Thus, one can distinguish between different vowels, regardless of their tone. This property is essential to vowel recognition.

For further information about the workings of the ear and vocal tract, refer to a text on human physiology, such as [5].

3 Theory and Design

Most of the digital processing theory in this report is based on Proakis and Manolakis [4]. Gersho and Gray [1] also provide a useful reference for predictive coding.

3.1 Linear Predictive Coding

Linear Predictive Coding (LPC), a widely used technique in speech coding and recognition applications, generates a filter model for data based on linear prediction coefficients. More technically, the speech data is modeled as an autoregressive process, and the parameters of the innovation filter are used to encode the data. The process assumes two excitation functions: an impulse train containing the fundamental frequency that accounts for 'voiced' speech, and random white noise that accounts for the 'unvoiced' speech [4]. Using the Levinson-Durbin algorithm for LPC of order p, a set of coefficients $\{a(i), 1 \leq i \leq p\}$ is generated. Subsequently, the filter model produced is that of an all-pole filter in the form:

$$H(z) = \frac{1}{1 + \sum_{i=1}^{p} a(i) z^{-i}} \tag{3.1}$$

3.2 Sampling

Each analog speech signal $s_a(t)$ must first be sampled at regular intervals to obtain the discrete-time signal:

$$s(k) = s_a(kT), \tag{3.2}$$

The samples are taken at integer multiples of the sampling period T.

According to the Sampling Theorem, in order to avoid frequency aliasing, the analog speech signal must be sampled at a frequency greater than twice the highest frequency contained within its spectrum. Thus, the sampling frequency F_s must satisfy

$$F_s = \frac{1}{T} \geq 2F_{\max} \tag{3.3}$$

Since typical human speech does not contain significant spectral energy above 4000 Hz, a sampling frequency of 8000 Hz is used to generate the discrete-time signal $s(k)$.

3.3 Sound Representation

When an LPC of order twelve ($p = 12$) is applied to a given sound sample, the coefficients $\{a(i), 1 \leq i \leq p\}$ obtained by the Levinson-Durbin algorithm could potentially be used to characterize a sound block. However, analysis shows that the roots of the polynomial

generated by these coefficients is more relevant in understanding the nature of the sound.

The formants in speech data are located at the frequencies at which the power spectrum peaks, as in Figure 2.2. For a white noise input, the Power Spectral Density, S, of the output of the filter can be calculated using the following equation:

$$\begin{aligned} S(\omega) &= \sigma_{\mathrm{w}}^2 H(e^{j\omega}) H(e^{-j\omega}) \\ &= \sigma_{\mathrm{w}}^2 |H(e^{j\omega})|^2 \end{aligned} \tag{3.4}$$

where $H(z)$ is the innovation filter generated by LPC. For an impulse input, the output of the filter is simply $H(z)$. Thus,

$$S(\omega) = |H(e^{j\omega})|^2 \tag{3.5}$$

The denominator of the transfer function can be factored into a product involving its poles, $\{\rho_i\}_{i=1}^p$, in the complex plane, and so the magnitude response can be calculated as follows:

$$\begin{aligned} H(z) &= \frac{1}{\prod_{i=1}^{p} (1 - \rho_i z^{-1})} \\ H(e^{j\omega}) &= \frac{1}{\prod_{i=1}^{p} (e^{j\omega} - \rho_i) e^{-j\omega}} \\ |H(e^{j\omega})| &= \frac{1}{\prod_{i=1}^{p} |(e^{j\omega} - \rho_i)|} \end{aligned} \tag{3.6}$$

The denominator in this last expression is the product of all distances of poles to a point on the unit circle at angle ω. A smaller distance to the unit circle results in a larger magnitude response. Thus, by combining Equation 3.6 with Equations 3.5 and 3.4, it can be seen that there is a direct correspondence between the square distance of poles to the unit circle and the height of peaks in the frequency power spectrum at a given frequency. These peaks occur at frequencies related to an angle in the complex plane by the equation:

$$f = \frac{\omega F_s}{2\pi} \tag{3.7}$$

Thus, it was concluded that the entries of a vector representing a sound sample would be more appropriately formed by the poles of its filter model.

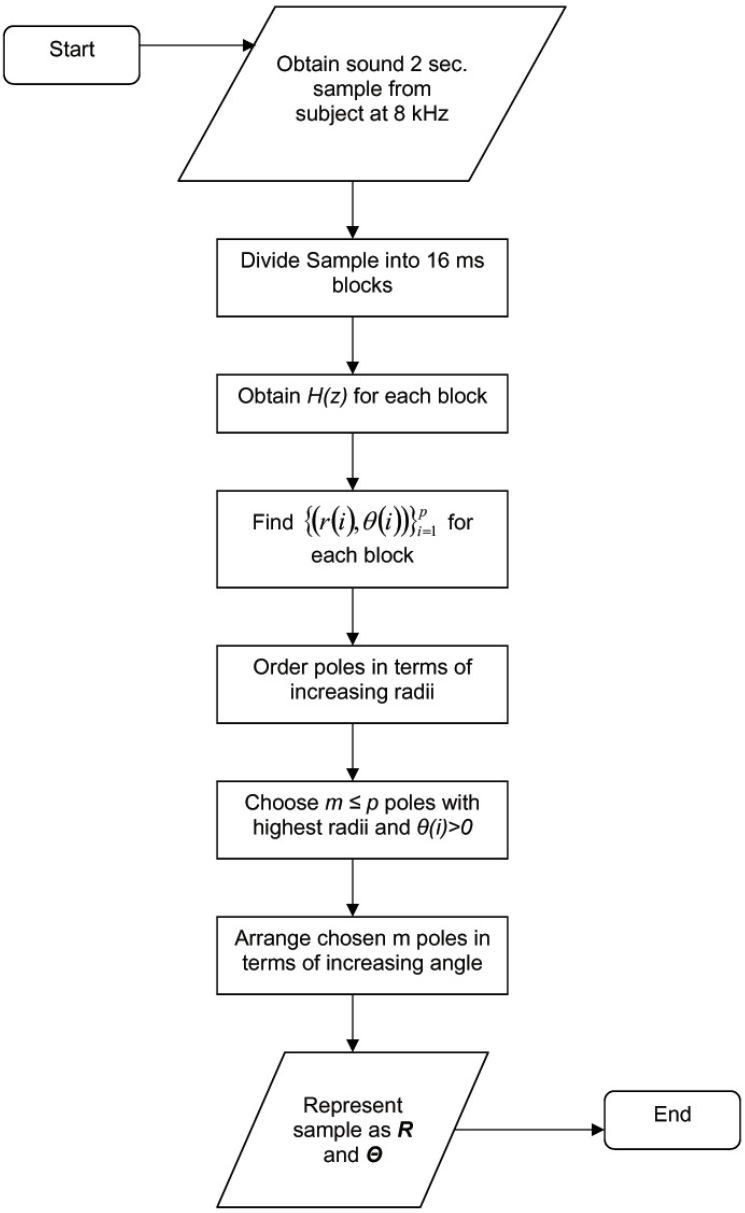


Figure 3.1: This flowchart demonstrates how to form a vector from a block of sound samples

Keeping in mind that the algorithms are to be implemented in real-time, the vectors were further refined. By examining the placement of poles in the complex plane for various samples of vowel sounds, it was noted that certain poles had a high variance for a particular sound, while others seemed to remain in a specific area. In general, those that moved a lot were furthest away from the unit circle, and had little impact on the overall shape of the frequency spectrum. This was exploited to reduce the size of the vectors representing certain vowels, which, in turn, reduced the overall complexity of the detection process. Also, since the coefficients in the AR innovation filter are real-valued, all poles are either real or appear in complex conjugate pairs. Thus, only poles in the upper-half complex plane need to be considered. The following algorithm was subsequently designed in order to obtain vectors representing a given sound sample:

1) For a given sound block, apply LPC decomposition to obtain a set of complex poles in Cartesian coordinates
$$\{\rho(i)\}_{i=1}^{p}$$

2) Represent each pole in polar coordinates to obtain the new set
$$\{(r(i), \theta(i))\}_{i=1}^{p}$$

3) Arrange the poles in order of increasing radius

4) Choose the $m \leq p$ poles with highest radii and strictly positive angles

5) Arrange the m chosen poles in terms in terms of increasing radii, yielding the new set of poles $\{\rho'(i)\}_{i=1}^{m}$, and form the vectors:
$$\begin{bmatrix} r'(1) \\ r'(2) \\ \vdots \\ r'(m) \end{bmatrix}, \begin{bmatrix} \theta'(1) \\ \theta'(2) \\ \vdots \\ \theta'(m) \end{bmatrix} \tag{3.8}$$

3.4 Training the System

Since the precise formant structures for certain vowels vary from person to person, the designed system has a 'training' phase, in which the library of vectors representing the

different sounds is populated. This ensures a higher accuracy in detections, and customizes the filters used for the modified speech production, resulting in a more realistic output.

In addition, including a training phase in the tool allows for more flexibility; the person conducting the study is able to dynamically customize or add vowel sounds.

3.4.1 Averaging

In order to accurately characterize a vowel sound from a particular subject, several sound samples must first be obtained. Performing the Levinson-Durbin algorithm on each of these blocks will result in a different set of poles, which must subsequently be averaged. However, the probability density function of the pole locations is unknown. It is assumed, however, that there is such a density for each particular vowel, and that the pole locations obtained from two different blocks of data are uncorrelated. Thus, the first and second statistical moments of each pole are estimated by taking the arithmetic average and sample variance of each pole location. If enough training sets are obtained, the Weak Law of Large Numbers (WLLN) justifies this approximation.

According the WLLN, the arithmetic mean of the pole locations converges to the statistical mean in probability. For example, if the radius and phase of a certain pole as are modeled as random variables R and Θ respectively, and it is further assumed that the distribution of the pole location at time i is independent and identically distributed to the pole location at time j (for $i \neq j$), then by the WLLN:

$$\lim_{N\to\infty} P\left\{\left|\frac{1}{N}\sum_{i=1}^{N} R(i) - \mu_r\right| \geq \epsilon\right\} = 0$$
$$\lim_{N\to\infty} P\left\{\left|\frac{1}{N}\sum_{i=1}^{N} \Theta(i) - \mu_\theta\right| \geq \epsilon\right\} = 0$$

where μ_r and μ_θ are the statistical averages for magnitude and phase respectively. Therefore, taking a large number of pole samples in the training process allows for the approximation of the first statistical moment for each pole location of a given vowel using the WLLN.

Similarly, in order to calculate the variance for each pole location, the second statistical moment is needed. Again, using the WLLN, the arithmetic average of the square of the radius and phase for each pole converges to the second statistical moment in probability, provided that sufficiently many sample poles are obtained in the training process.

3.4.2 Populating The Library

The following algorithm is used to create each library entry:

1) Break a given sound segment of length L seconds into N 16 ms blocks

2) Perform the Levinson-Durbin algorithm on each block to obtain N sets of LPC coefficients:

$$\{a_n(i)\}_{i=1}^{p}, \quad n = 1 \dots N$$

3) Using the N samples for each LPC coefficient, find its average

$$\bar{a}(i) = \frac{1}{N} \sum_{n=1}^{N} a_n(i), \quad i = 1 \dots p \tag{3.9}$$

4) Find the roots corresponding to each polynomial formed by a set of LPC coefficients (or the poles of each filter model), and convert these into polar form to obtain the set:

$$\{(r_n(i), \theta_n(i)\}_{i=1}^{p}, \quad n = 1 \dots N$$

5) Using the N samples for each pole, calculate:

- The arithmetic mean of the magnitude:

$$\bar{r}(i) = \frac{1}{N} \sum_{n=1}^{N} r_n(i), \quad i = 1 \dots p$$

- The sample variance of the magnitude:

$$\sigma_r^2(i) = \frac{1}{N} \sum_{n=1}^{N} (r_n(i) - \bar{r}(i))^2, \quad i = 1 \dots p$$

- The arithmetic mean of the angle:

$$\bar{\theta}(i) = \frac{1}{N} \sum_{n=1}^{N} \theta_n(i), \quad i = 1 \dots p$$

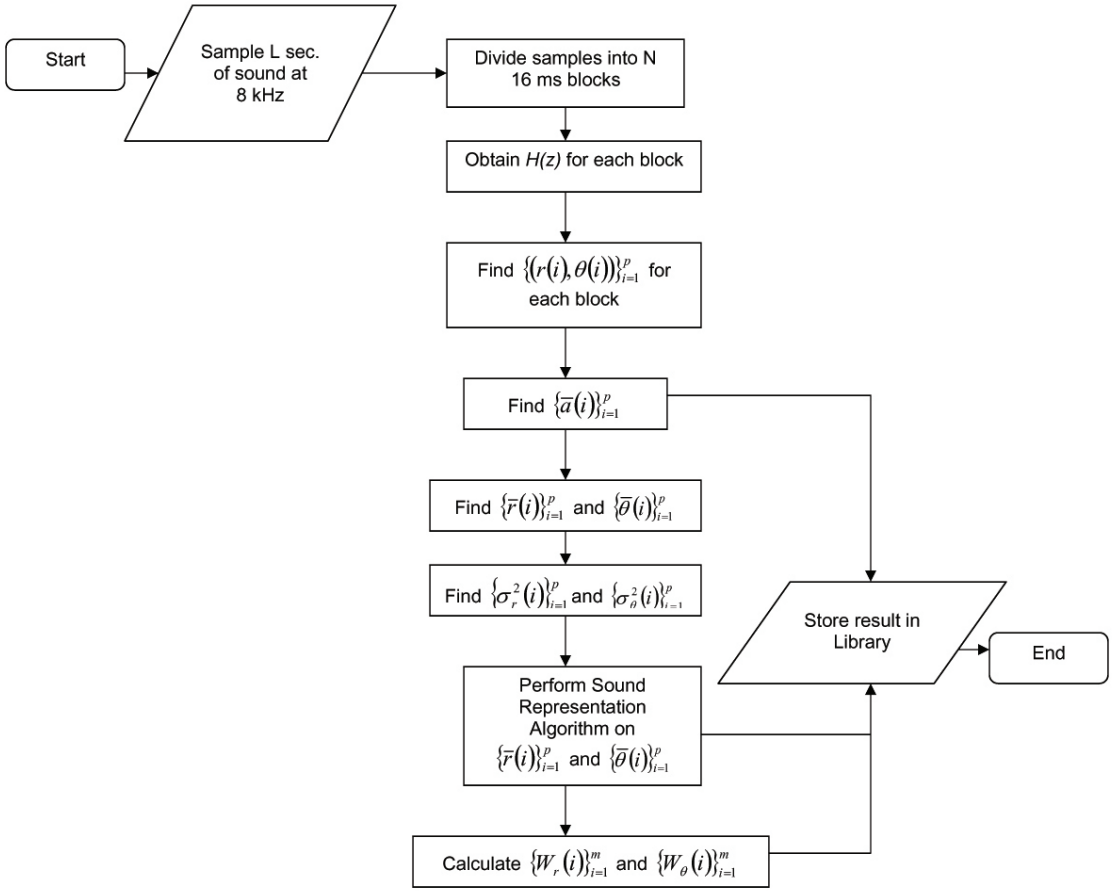


Figure 3.2: This flowchart demonstrates how to form a vowel for the library

- The sample variance of the angle:

$$\sigma_\theta^2(i) = \frac{1}{N}\sum_{n=1}^{N}(\theta_n(i) - \bar{\theta}(i))^2, \quad i = 1 \dots p$$

6) Using the new set $\{(\bar{r}(i), \bar{\theta}(i))\}_{i=1}^{p}$, use the algorithm proposed in Section 3.3 to form the vectors:

$$\vec{R} = \begin{bmatrix} \bar{r}'(1) \\ \bar{r}'(2) \\ \vdots \\ \bar{r}'(m) \end{bmatrix}, \quad \vec{\Theta} = \begin{bmatrix} \bar{\theta}'(1) \\ \bar{\theta}'(2) \\ \vdots \\ \bar{\theta}'(m) \end{bmatrix}$$

7) Calculate the logarithmic total power[1] P of the new set of poles:

$$\begin{aligned} P &= \log\left(\prod_{i=1}^{m}\frac{1}{(1-|\bar{\rho}'(i)|)^2}\right) \\ &= \log\left(\prod_{i=1}^{m}\frac{1}{(1-\bar{r}'(i))^2}\right) \\ &= -2\sum_{i=1}^{m}\log\left(1-\bar{r}'(i)\right) \end{aligned} \tag{3.10}$$

8) For each of the m poles, calculate the weights to be used in the weighted difference as follows:

$$\begin{aligned} W_r(i) &= \frac{\log\left(\frac{1}{(1-|\bar{\rho}'(i)|)^2}\right)}{mP(\sigma_r^2(i)+\delta)} \\ &= \frac{-2\log\left(1-\bar{r}'(i)\right)}{mP(\sigma_r^2(i)+\delta)} \end{aligned} \tag{3.11}$$

$$W_\theta(i) = \frac{-2\log\left(1-\bar{r}'(i)\right)}{mP(\sigma_\theta^2(i)+\delta)} \tag{3.12}$$

These Weighting Factors are designed to account for the direct correspondence between distance of poles to the unit circle in the complex plane, and the height of peaks in the

[1]Technically, this is not a 'total power'. This value will later be used as a normalization constant in a weighted average. The maximum contribution to power of a pole occurs at a frequency related to the angle of that pole by equation 3.7. By considering only these maximum power contributions, we are taking the 'total power' to be a product of the contributions of each pole.

frequency power spectrum. The factor δ was introduced to ensure that no division by zero occurs in case the sample variance of any of the poles or angles is nil.

9) Finally, create a library entry for the sound sample that includes the vectors:

$$\vec{R} = \begin{bmatrix} \bar{r}'(1) \\ \bar{r}'(2) \\ \vdots \\ \bar{r}'(m) \end{bmatrix}, \vec{\Theta} = \begin{bmatrix} \bar{\theta}'(1) \\ \bar{\theta}'(2) \\ \vdots \\ \bar{\theta}'(m) \end{bmatrix}, \vec{W}_r = \begin{bmatrix} W_r(1) \\ W_r(2) \\ \vdots \\ W_r(m) \end{bmatrix}, \vec{W}_\theta = \begin{bmatrix} W_\theta(1) \\ W_\theta(2) \\ \vdots \\ W_\theta(m) \end{bmatrix}, \vec{A} = \begin{bmatrix} \bar{a}(1) \\ \bar{a}(2) \\ \vdots \\ \bar{a}(p) \end{bmatrix}$$

3.4.3 Pole Alignment

In previous derivations, it was assumed that the poles of the autoregressive model had some order that allowed them to be labeled $\rho(i),\ i = 1 \ldots p$. However, this is not the case. Complex numbers have no natural order like the integers or reals, so a method was required to arrange them in such a way that sets of these poles could be averaged together. Several techniques were attempted, including arranging poles by angle, by magnitude, and by aligning a new set of poles to a previous set, based on a nearest neighbour (NN) approach.

The problem with arranging poles by angle is that when two poles have relatively close angles but very different magnitudes, they will switch order when the angles cross each other, resulting in large errors when averaging the ordered sets. Arranging by magnitude was even worse, since there is much less separation in magnitude than there is in angle. An algorithm based on NN therefore developed. The question then became what to use for a base set to align to.

It was noted that the problem of alignment only arises when the transfer function is factored into its poles; the coefficients themselves do have a natural order. In the training set, the LPC coefficients for the N blocks of 16 ms (as described in Section 3.4.2) are available. An initial estimate of the average pole placements was thus calculated by first averaging these N sets of AR coefficients, as in Equation 3.9, and finding the roots of this averaged polynomial. The following iterative process was then developed to refine the average poles:

1) Initialize a counter $k = 0$, and a number of iterations K

2) Initialize the average poles by calculating the average LPC coefficients, $\bar{a}(i)$, and factoring the corresponding polynomial to obtain the set: $\mathcal{P}^{(0)} = \{\bar{\rho}^{(0)}(i)\}_{i=1}^{p}$

3) Initialize a set of distance variances $\mathcal{V}^{(0)} = \{\sigma^2_{\mathcal{P}}\}_{i=1}^{p}$ with values of 1. This is to be used to remove outliers, and will be the variance in Euclidean distance from each pole to the base set

4) For each set of LPC coefficients, arrange the poles such that the i^{th} pole is closest to the i^{th} base pole in $\mathcal{P}^{(k)}$. If two poles are closest to the same base pole, count the further one as an outlier. If a pole is further than $\sigma^2_{\mathcal{P}}(i) \in \mathcal{V}^{(k)}$ away from its nearest neighbour, then count it as an outlier

5) Update $\mathcal{P}^{(k+1)}$ and $\mathcal{V}^{(k+1)}$ by averaging the new aligned poles and calculating the sample variance of the Euclidean distance for each pole

6) If $k < K$, $k := k + 1$, and go to step 4

Using this algorithm, the alignment only typically took 2 iterations to converge. An example of average poles obtained using this algorithm is shown in Figure 3.3.

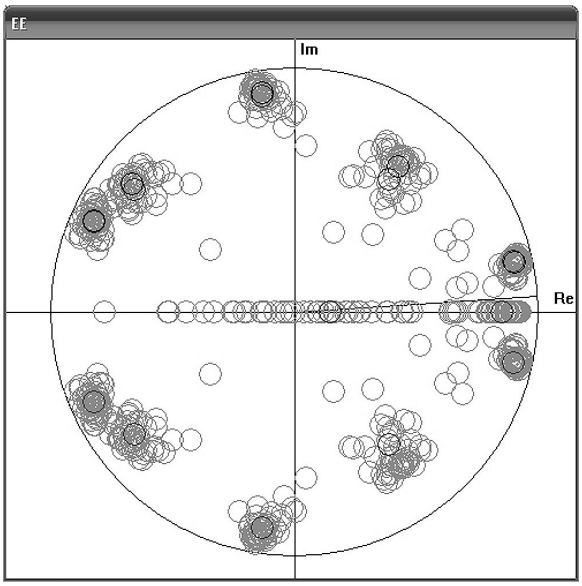


Figure 3.3: Demonstration of pole alignment results. Aligned poles are black, initial estimates from averaged LPC coefficients are in dark grey, and all poles from the training set are in light grey. Note that trained poles only appear i the upper-half complex plane

3.5 Sound Detection

3.5.1 Difference Function Criteria

In order to detect sounds, a suitable difference function is required to compare vector forms of speech segments to those in the library. It is constructed using the following criteria:

1) It should depend on the difference between magnitudes and difference between angles of poles

2) Deviations from poles that are closer to the unit circle should be weighted more than deviations from poles that are further from the unit circle

3) Deviations from poles that have a smaller variance (in both magnitude and angle) should be weighted higher than those with higher variances

The first criteria is justified since a change in angle represents a change in frequency of a formant, and a change in magnitude represents a change in power at the formant frequency.

It has previously been asserted that the closer poles are to the unit circle in the filter model, the higher the peaks are in the output power spectrum. Thus, the poles that are closer to the unit circle have a larger impact on the location of the formants. Deviations away from these poles have a larger impact on the shape of the power spectrum than those that are farther away from the unit circle, and so these deviations should be weighted higher. This justifies the second criteria.

While examining speech data for specific vowel sounds, it was observed that some poles vary their placement in the complex plane more than others. Several poles often stay in a small, fixed region for a particular vowel sound. This implies that those poles that remain relatively fixed have more of an impact on the production of a certain vowel sound, and so deviations away from these poles should be weighted higher than deviations away from poles with higher placement variances. Thus, the variances of magnitude and angle should be considered in the metric.

3.5.2 Initial Difference Function

An initial difference function, $d(\cdot)$, was developed to incorporate the criteria listed in 3.5.1:

$$d(\mathbf{P_1}, \mathbf{P_2}) = \sum_{i=1}^{m} (1-\alpha) \frac{\max(\bar{r}_1'(i), \bar{r}_2'(i))(\bar{r}_1'(i) - \bar{r}_2'(i))^2}{\sigma_r^2(i) + \delta} + \alpha \frac{(\bar{\theta}_1'(i) - \bar{\theta}_2'(i))^2}{\sigma_{\theta,i}^2 + \delta} \tag{3.13}$$

where $\mathbf{P_1}$ and $\mathbf{P_2}$ are two sets of poles, consisting of magnitudes and angles,

$$\mathbf{P_1} = \begin{bmatrix} \bar{r}'_1(1) & \bar{\theta}'_1(i) \\ \bar{r}'_1(2) & \bar{\theta}'_1(2) \\ \vdots & \\ \bar{r}'_1(m) & \bar{\theta}'_1(m) \end{bmatrix}, \quad \mathbf{P_2} = \begin{bmatrix} \bar{r}'_2(1) & \bar{\theta}'_2(i) \\ \bar{r}'_2(2) & \bar{\theta}'_2(2) \\ \vdots & \\ \bar{r}'_2(m) & \bar{\theta}'_2(m) \end{bmatrix} \tag{3.14}$$

The max(·) term ensures criterion two is satisfied, and dividing by the variance ensures criterion three is satisfied, normalizing the data. The δ term was added to prevent dividing by zero, since real poles often had an angular variance of zero. The α term allows for more of an emphasis to be placed on a difference of angle (frequency) or a difference in magnitude, depending on whether it is greater than or less than $\frac{1}{2}$.

However, the initial difference function did not perform satisfactorily. When comparing a library entry of the sound 'EE' against a two second sample of that vowel, both spoken by the same subject, the difference function detected only certain instances of the sound. Nonetheless, by means of additional testing, the metric was subsequently refined to eventually meet the detection requirements.

3.5.3 Refined Difference Function

In order to better characterize the position of the poles, the Weighting Factors described in Section 3.4.2 were developed. In addition, through the process of testing and refining the difference function, it was determined that the detection of sound samples could be accomplished more effectively through comparison of angles and radii separately. Subsequently, the refined difference functions were formed:

$$d_r(\mathbf{P_1}, \mathbf{P_2}) = \sum_{i=0}^{m} W_r(i)(\bar{r}'_1(i) - \bar{r}'_2(i))^2 \tag{3.15}$$

$$d_\theta(\mathbf{P_1}, \mathbf{P_2}) = \sum_{i=0}^{m} W_\theta(i)(\bar{\theta}'_1(i) - \bar{\theta}'_2(i))^2 \tag{3.16}$$

where $\mathbf{P_1}$ and $\mathbf{P_2}$ are defined as in Equation 3.14.

With these difference functions, it is now possible to represent the difference between two sets of poles. In order for a detection to occur, these two distances between the incoming data and a vowel in the library must be less than some thresholds ϵ_r and ϵ_θ. The problem then became how to set these thresholds. Through experimentation, it was revealed that the

optimal thresholds were dependent on which library vowel the incoming data was compared to. To compensate, the training phase was modified to determine appropriate thresholds for each vowel, and the weights were modified to normalize these thresholds to one for every vowel. Letting the set of poles for the k^{th} vowel in the library be represented as $\mathbf{P_v}(k)$, the following steps were included in the training algorithm:

1) Determine all distances, $\{d_r(k,i)\}_{i=1}^N$ and $\{d_\theta(k,i)\}_{i=1}^N$, of the k^{th} vowel to the training data using the previously defined weights, where N is the total number of 16 ms blocks in the training data, as before

2) Compute the arithmetic mean and sample variance of these sets, $\bar{d}_r(k)$, $\sigma^2_{d,r}(k)$, $\bar{d}_\theta(k)$, and $\sigma^2_{d,\theta}(k)$

3) Set $\epsilon_r(k) = \bar{d}_r(k) + 2\sigma_{d,r}(k)$, and $\epsilon_\theta(k) = \bar{d}_\theta(k) + 2\sigma_{d,\theta}(k)$

4) Redefine the weights from Equations 3.11 and 3.12 for the vowel as:

$$W'_r(k,i) = \frac{-2\log\left(1-\bar{r}'_v(k,i)\right)}{\epsilon_r(k) m P(\sigma^2_{v,r}(k,i)+\delta)} \tag{3.17}$$

$$W'_\theta(k,i) = \frac{-2\log\left(1-\bar{r}'_v(k,i)\right)}{\epsilon_\theta(k) m P(\sigma^2_{v,\theta}(k,i)+\delta)} \tag{3.18}$$

By setting the thresholds to be equal to the mean distance plus twice the standard deviation of the distances, we ensure that most of the training data results in a positive detection, but also keep a tight bound so that there are few false detections using this scheme. In the detection mode of the program, the new difference functions are used:

$$d_r(\mathbf{P},\mathbf{P_v}(k)) = \sum_{i=0}^{m} W'_r(k,i)(\bar{r}'(i)-\bar{r}'_v(k,i))^2 \tag{3.19}$$

$$d_\theta(\mathbf{P},\mathbf{P_v}(k)) = \sum_{i=0}^{m} W'_\theta(k,i)(\bar{\theta}'(i)-\bar{\theta}'_v(k,i))^2 \tag{3.20}$$

where $\mathbf{P}$ is the set of poles from the incoming sound segment. The difference functions are now constructed such that for the k^{th} vowel, a detection occurs when $d_r(\mathbf{P},\mathbf{P_v}(k)) < 1$ and $d_\theta(\mathbf{P},\mathbf{P_v}(k)) < 1$. If two sets of poles from the library satisfy this, then the one with the smallest product $d_r(\mathbf{P},\mathbf{P_v}(k)) \cdot d_\theta(\mathbf{P},\mathbf{P_v}(k))$ is selected.

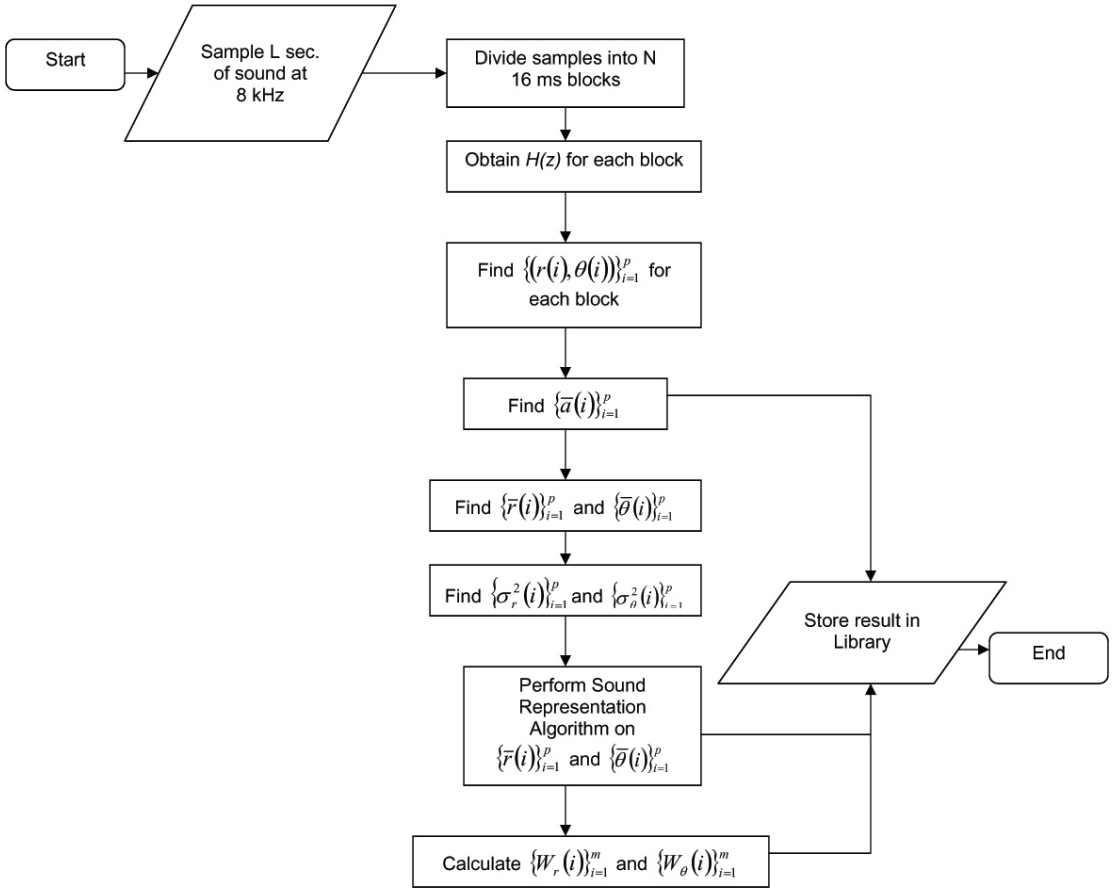


Figure 3.4: This flowchart outlines the steps involved in detecting a vowel

3.6 Sound Replacement

Performing the Levinson-Durbin algorithm on a speech segment generates a set of LPC coefficients, which in turn can be used to form an estimate of a sample $x(n)$ using the past p samples:

$$\hat{x}(n) = \sum_{i=1}^{p} a(i)x(n-i)$$

Subsequently, the error between the original signal and its estimate can be calculated as:

$$\begin{aligned} e(n) &= x(n) - \hat{x}(n) \\ &= x(n) - \sum_{i=1}^{p} a_i x(n-i) \end{aligned}$$

Taking the z-transform results in:

$$\begin{aligned} E(z) &= (1 - \sum_{i=1}^{p} a_i z^{-i})X(z) \\ &= \frac{1}{H_1(z)}X(z) \end{aligned} \tag{3.21}$$

where $H_1(z)$ is the filter model corresponding to the LPC coefficients for the data.

From this, one can see that the error can be constructed from the data and the inverse filter, $1/H_1(z)$. The error signal is a combination of white noise and periodic impulses with a period equal to the tone of the speaker. To re-synthesize the original vowel sound, this error signal is sent back through the filter model, resulting in the set of output samples $y(n)$:

$$\begin{aligned} Y(z) &= H_1(z)E(z) \\ &= H_1(z)\frac{1}{H_1(z)}X(z) \\ &= X(z) \\ y(n) &= x(n) \end{aligned} \tag{3.22}$$

In order to synthesize a separate vowel sound, instead of feeding the error signal through the filter model $H_1(z)$, the error signal is sent through the model of the replacing sound, $H_2(z)$. The principle behind this is as follows: if the replacing vowel sound was analyzed instead, and the inverse filter applied to produce an error signal $\tilde{e}(n)$, then $\tilde{e}(n)$ would have

the same distribution as $e(n)$ since it would also contain white noise and the same tone information. By using this method, the periodic impulses in the error signal $e(n)$ cause the new vowel sound to mimic the tone of the original sound data, making the output sound realistic.

Given the original model, $H_1(z)$, and the replacement vowel's model, $H_2(z)$:

$$H_1(z) = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}}, \quad H_2(z) = \frac{1}{1 - \sum_{i=1}^{p} b_i z^{-i}}$$

The modified output, $y(n)$, can be generated from the input signal in the following manner:

$$\begin{aligned} Y(z) &= H_2(z)E(z) \\ &= H_2(z)\frac{1}{H_1(z)}X(z) \\ &= \frac{1 - \sum_{i=1}^{p} a_i z^{-i}}{1 - \sum_{i=1}^{p} b_i z^{-i}}X(z) \\ y(n) &= x(n) - \sum_{i=1}^{p} a_i x(n-i) + \sum_{i=1}^{p} b_i y(n-i) \end{aligned} \tag{3.23}$$

This form of an FIR filter can be easily implemented using the difference equation (3.23).

3.7 Graphical User Interface

An important aspect of the Speech Analysis and Modification project is the design of a Graphical User Interface (GUI) to facilitate the use of the algorithm. Two interfaces were created: one for the training phase, and one for full-running mode.

3.7.1 Training GUI

The Training Mode GUI allows the user to obtain sound samples from the subject, and store them as library entries to be used as reference for detection.

The interface can be seen in Figure 3.5. The 'Record' button allows the user to acquire the training samples. After using 'Play' to test for its quality, the user may then accept

or reject the data by clicking on the appropriate button. Once the sound is accepted or rejected, the user can then decide to record more samples, or to continue to the next step of labeling the vowel sound. When 'Add Vowel' is clicked, this label appears in the list box. In addition, a plot illustrating the position of all training poles as well as the averages for the library is shown.

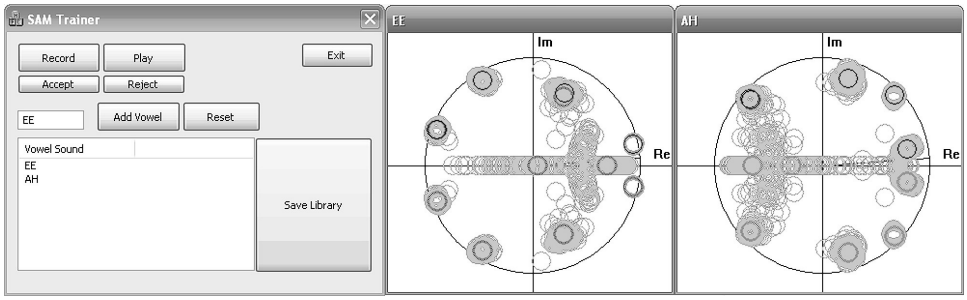


Figure 3.5: The 'Training Mode' user interface. It allows the researcher to obtain various sound samples from a user in order to build a library of vowel sounds

If the user gauges the variance of the poles from their average to be acceptable, then the sample is suitable for use as reference, and may be stored as a library entry as described in Section 3.4.2 by clicking on 'Save Library'. However, the user may also restart the whole process by clicking on 'Reset' if he or she is if unsatisfied with the results.

3.7.2 Running Mode GUI

The Running Mode GUI, seen in Figure 3.6 is comprised of six windows: 'Main', 'Waveform', 'AR Estimate', 'FFT', 'Poles of the AR Model' and 'SAM'. By clicking on the appropriate button in the Main window, the user may start or stop the full Running Mode of SAM. The user can also load the library file corresponding to the subject under test, and by using the drop-down boxes, decide which vowels should be detected, and what they should be replaced with. Finally, the 'Noise Threshold' slide bar allows the user to adjust detection according to the surrounding noise level; increasing the threshold will aid in reducing false detections. The 'Waveform' window displays the amplitude of the subject's speech input as a function of time, while 'AR Power Estimate' illustrates the signal's Power Spectral Density in the frequency domain based on the autoregressive model. 'FFT' displays the signal's Fast Fourier Transform, and 'Poles of AR Model' shows the location of the complex poles associated with the LPC filter model of each speech signal. Finally, the 'SAM' window

displays the tool's logo with a speech bubble indicating which vowel sound has been detected, if any.

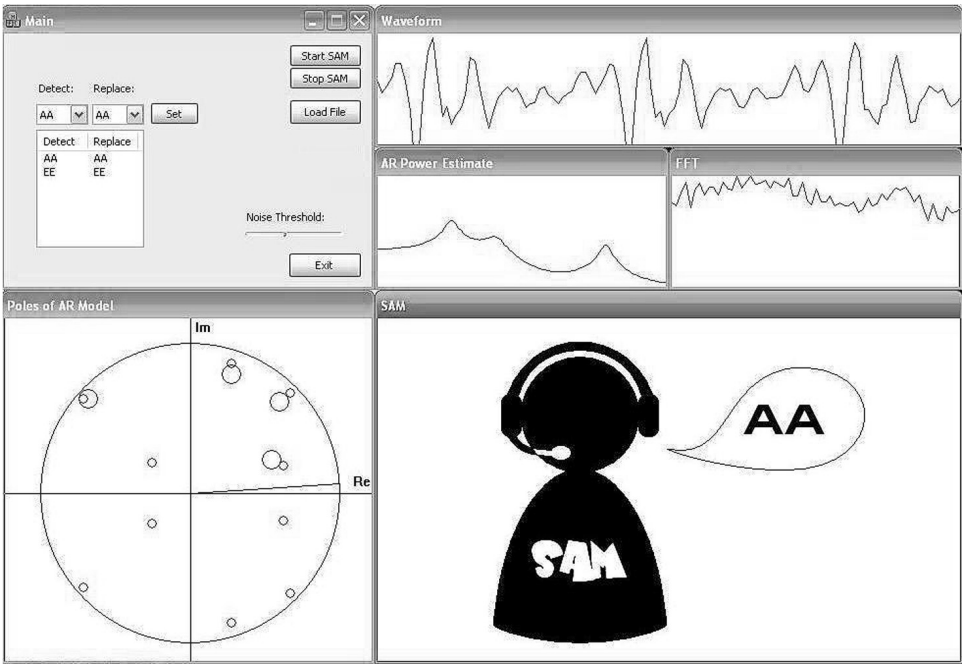


Figure 3.6: The 'Running Mode' user interface. It allows the researcher to change the parameters of the detection and replacement system, and provides several analysis tools

4 Results and Discussion

4.1 Detection Performance

The detection performance of SAM was evaluated for two sentences spoken by the subject under test. It was analyzed based on three performance criteria for the vowel sounds 'AH' and 'EE':

- The percentage of correct detections
- The percentage of incorrect detections
- The percentage of missed detections

For example, a correct detection is said to have occurred when an 'AH' sound is spoken in a particular block of sound samples, and SAM properly detects the input as an 'AH' sound. Similarly, an incorrect detection occurs when SAM recognizes a sound sample as a particular vowel at a time when the vowel sound is not spoken. Finally, a missed detection occurs when a certain vowel sound was spoken and was not recognized by SAM.

The following two sentences in which the sounds 'AH' and 'EE' are prevalent were used to test SAM:

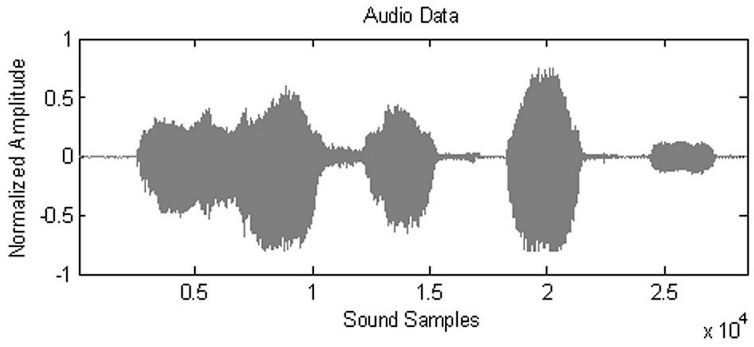
- Test Sentence 1: "Dalia has fat feet"
- Test Sentence 2: "Antonio has bad teeth"

Figure 4.1 shows the detection results for the occurrences of the 'AH' and 'EE' sounds in the first test sentence. Figure 4.1(b) illustrates occurrences of the 'AH' sound in the test sentence, and the corresponding detections made by SAM. It can be noted from this figure that a number of negative detections of short duration were made, as well as small segments that correspond to missed detections. Overall, for the first test sentence, SAM correctly detected 'AH' 83.7% of the time, falsely detected the sound 14.4% of the time, and missed detecting 1.8% of the time.

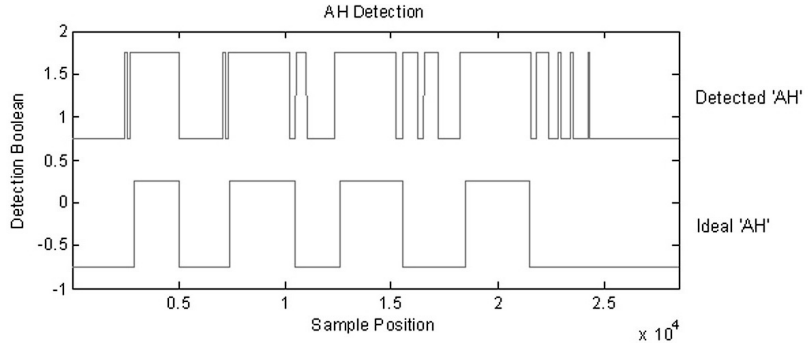
The occurrences of the 'EE' vowel sound in the first test sentence are shown in Figure 4.1(c), as well as the corresponding positive detections, false detections, and missed detections. For this vowel sound, SAM correctly detected 91.8% of the time, falsely detected 6.4% of the time, and missed the 'EE' sound 1.7% of the time.

The results of SAM's performance on Test Sentence 2 are shown in Figure 4.2. It can be seen that there were significantly more of both negative detections and missed detections of 'AH' than there were when testing with the first sentence. These negative and missed detections were also of longer duration than in the previous example. Overall, for Test Sentence 2, SAM correctly detected 'AH' 71.0% of the time, falsely detected the sound 24.7% of the time, and missed detecting 4.20% of the time.

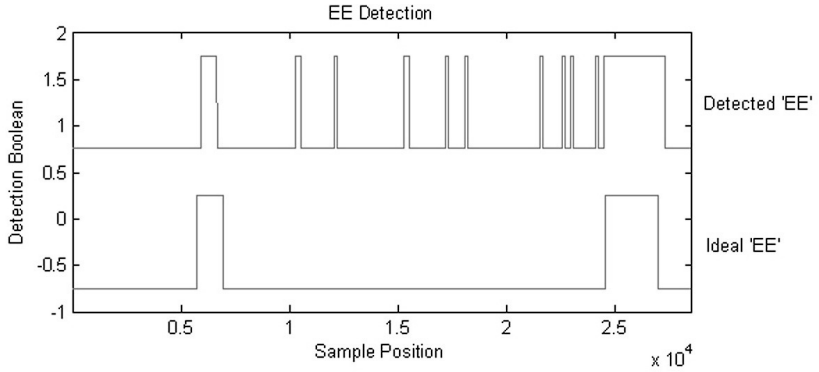
Figure 4.2(c) shows the occurrences of the vowel sound 'EE' in Test Sentence 2, and the corresponding positive detections, false detections, and missed detections. Although there were significantly more negative detections for 'EE' than there were in the first test sentence, there was a substantial improvement in the number of missed detection. For the sound 'EE', SAM correctly detected 86.0% of the time, falsely detected 13.2% of the time, and missed a spoken 'EE' 0.69% of the time.



(a) Time domain waveform

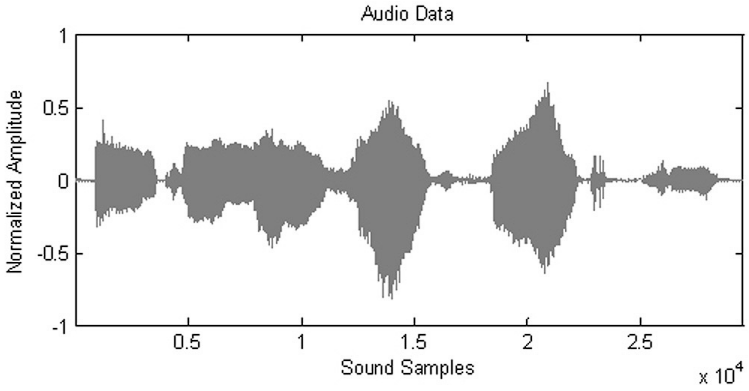


(b) Detections of "AH"

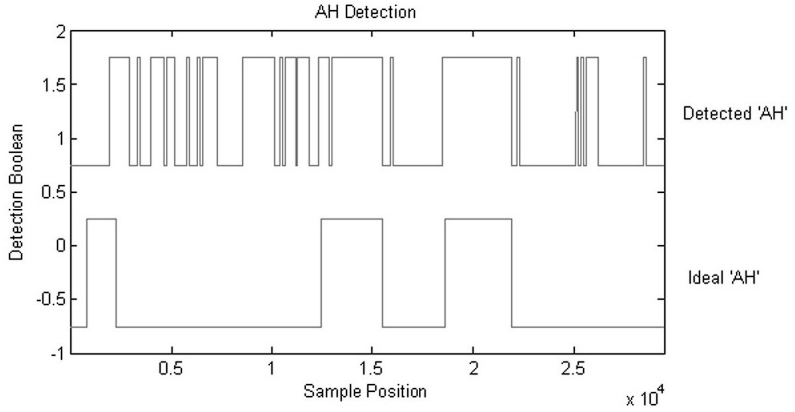


(c) Detections of "EE"

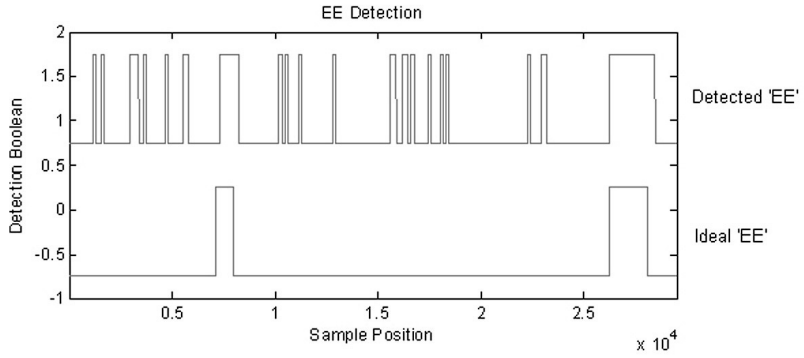
Figure 4.1: Evaluation of the first test sentence, "Dalia has fat feet"



(a) Time domain waveform



(b) Detections of "AH"



(c) Detections of "EE"

Figure 4.2: Evaluation of the second test sentence, "Antonio has bad teeth"

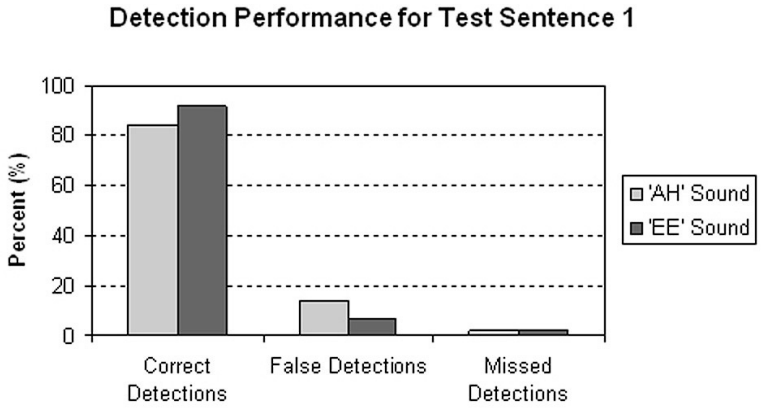


Figure 4.3: Detection statistics for "Dalia has fat feet"

A summary of the detection performance for Test Sentences 1 and 2 are shown in Figures 4.3 and 4.4 respectively. It can be seen that SAM performs better in the detection of the 'EE' sound in both cases. It can also be noted that the performance was generally better for Test Sentence 1 than for Test Sentence 2.

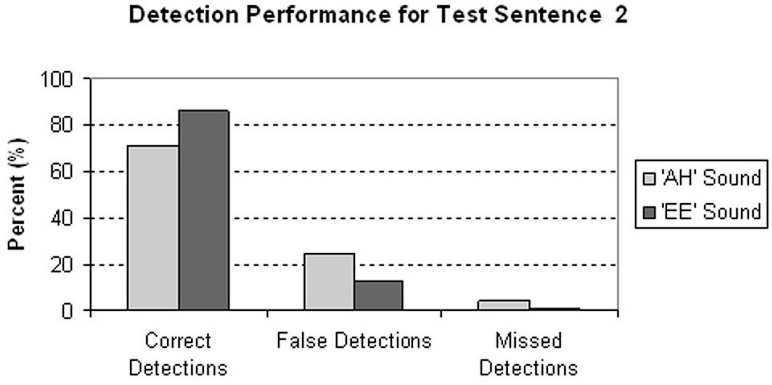


Figure 4.4: Detection statistics for "Antonio has bad teeth"

For both vowel sounds in the test sentences, SAM displayed a number of negative detections. Although the negative detections were of short duration, this has a significant affect on the quality of the output speech signal. In a block of speech that has a negative detection, the resulting output speech will sound unnatural, since there will be a short-duration vowel

sound where one doesn't belong.

It can also be seen that in the occurrences of both vowel sounds in the test sentences, there were small segments of missed detections. Although the percent of missed detections is very small for both vowel sounds, this will still have a noticeable effect on the output speech signal quality by the vowel replacement switching on and off in short durations.

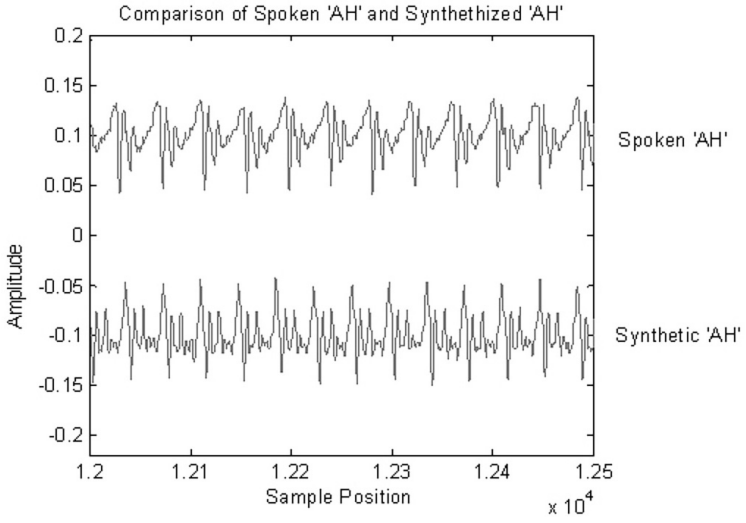
4.2 Vowel Synthesis Quality

Another important feature of SAM is the quality of speech reproduction. In order for SAM to be used effectively in a research setting, the output speech signal must be as close to the speaker's voice as possible. It is for this reason that the quality of voice reproduction upon vowel replacement is an important measure of performance.

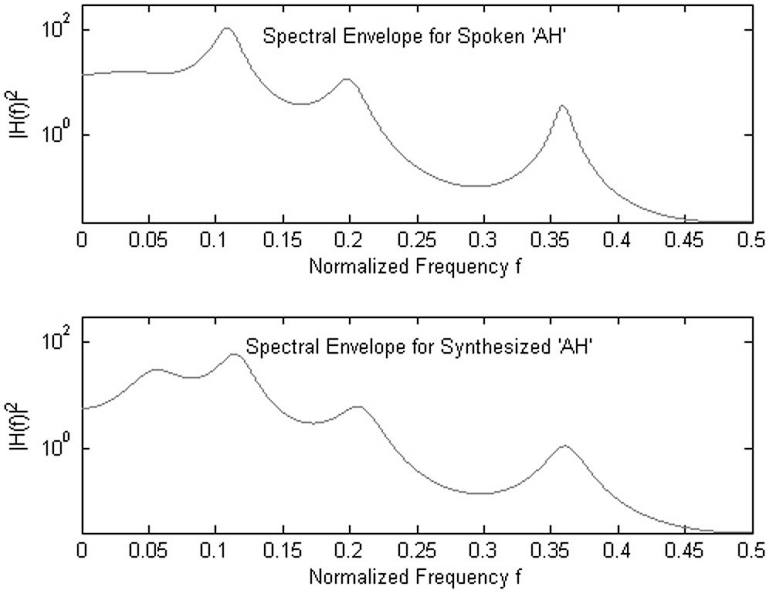
The quality of the re-synthesized vowel sound was examined for the vowels sounds 'AH' and 'EE'. While it is difficult to quantify a characteristic such as sound quality, it is informative to compare the properties of spoken vowel and the re-synthesized sound in both time and frequency domains.

Figures 4.5(a) and 4.6(a) show the time-domain plots comparing the original and synthesized 'AH' and 'EE' sounds respectively. It can be seen that the waveforms are of similar shape, amplitude, and period, indicating an output that is very close to the original vowel sound in quality.

A more important measure of reproduction quality is the spectral characteristics of the output signal. It is important for the synthetic output to have a power spectral density similar to the sound it attempts to emulate. Figure 4.5(b) shows the spectral characteristics of a spoken 'AH' sound compared to those of a synthetic 'AH'. Although the amplitudes of the most significant frequencies differ slightly, it can be seen that, overall, the spectral plots of both signals are very similar. This is a strong indication of a high quality reproduction. Similarly, Figure 4.6(b) shows the spectral characteristics of an original spoken 'EE' sound, and the re-synthesized 'EE'. One can see that the two most significant frequencies are identical in the original and synthetic 'EE' sounds; the formants of the original and synthesized vowels are identical, indicating they sound the same. Although the magnitude of the main lobe of the synthetic signal is significantly greater than the original, the spectral plots agree quite closely overall, which again indicates quality performance.

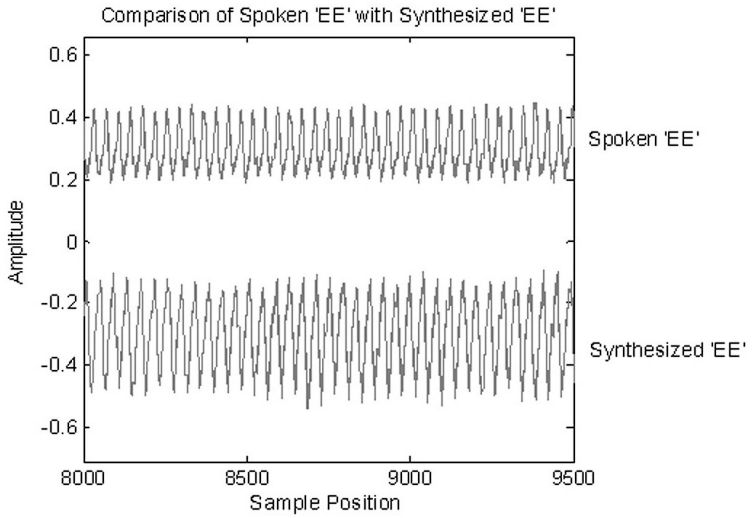


(a) Time domain comparison

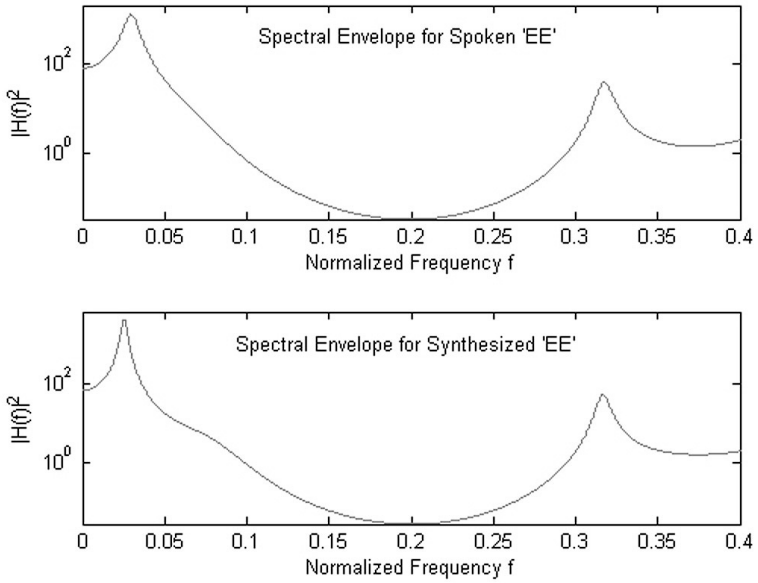


(b) Frequency domain comparison

Figure 4.5: Comparison of the "AH" sound from an original source, and one that is synthesized from an "EE" sound using the methods described in Section 3.6



(a) Time domain comparison



(b) Frequency domain comparison

Figure 4.6: Comparison of the "EE" sound from an original source, and one that is synthesized from an "AH" sound using the methods in Section 3.6

5 Conclusions

A real-time vowel detection and replacement tool, nicknamed SAM, was designed and programmed in Microsoft Visual C++. There are two major components of the tool: a training component, and a main running component. The training component builds a library of vowel sounds, specific to a user, and the running mode loads a library, and performs the detections and replacements. A graphical user-interface was designed to allow the user to set the parameters of the system, such as which vowels to detect, which to replace, and if they are to be replace, which vowels to replace them with.

The detection algorithm uses a subset of the poles of autoregressive models to compare incoming speech data to a library of vowels, attempting to detect specified vowel sounds. The tool accomplishes this detection with the use of a pair of weighted difference functions, given in Equations 3.19 and 3.20. Upon a detection, the tool filters the incoming speech with a transforming filter, performing the function achieved by the difference equation given in 3.23. This filtering alters the original voice data to synthesize a new vowel sound.

Currently, SAM performs satisfactorily in detecting a spoken 'AH', replacing it with a synthesized 'EE', and vice-versa. The system results in positive detections roughly 80% of the time, with few missed detections, and several false detections. Upon positive detections, the replacement filtering functions extremely well. While results are better conveyed by listening to the audio output, plots of the time domain waveforms and their spectral characteristics do suggest the synthesized vowels sound similar to the original ones.

6 Recommendations

There is definitely room for improvement in SAM. For example, expanding the system to allow for detection and replacement of additional vowel sounds, such as 'OO' and 'UH' for instance, is desirable. In addition, the problem of Speech Recognition may be further explored by investigating the possibility of detecting non-periodic speech, such as consonants. Though the metrics SAM uses were not tested on such sounds, they will undoubtedly need to be updated to accommodate such studies.

More sophisticated spectrum techniques could be used to improve performance, such as those suggested by Thomson [6], but the calculation time must not exceed the allowable 16 ms.

There is a lot of low frequency interference and noise coming from the microphone input.

One strong signal is generally around 60 Hz, coming from electrical noise. Filtering prior to performing the speech analysis can be performed in order to remove this noise. Another option is to use adaptive filters while the total power from the microphone input is below some threshold in order to remove the background noise signals.

The most negative detections (SAM detecting a vowel when no vowel was spoken) occurred for unvoiced speech inputs, such as from many consonants. The poles for these sections tended to be scattered and random, and would sometimes trigger a detection. To avoid this, an algorithm to detect voiced versus unvoiced speech can be added so that a detection is only made when the input is voiced.

Finally, should it be decided that SAM is suitable for more commercial purposes beyond research, it would be advisable to update the Graphical User Interface to include more parameters relevant to the psychology researchers rather than the mathematicians. For example, displaying subject reaction time would be more practical than pole placement graphs. Also be aware of the licenses and copyrights involved in this software; there are components in this tool that come from other sources. These are listed in Appendix A.

References

[1] Gersho, A., and Gray, R. M. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Boston, 2001.

[2] Nave, C. R. "Vowel Sounds." Online. *Hyperphysics*. 2005. Accessed: 14 Nov. 2006 <http://hyperphysics.phy-astr.gsu.edu/hbase/music/vowel.html>.

[3] Pickard, D. "The Human Ear." Online. *Wikipedia*. 22 Jan. 2006. Accessed: 12 Feb. 2007 <http://en.wikipedia.org/wiki/Ear>.

[4] Proakis, J. G., and Manolakis, D. G. *Digital Signal Processing*, 4th ed. Pearson Prentice Hall, New Jersey, 2007.

[5] Silverthorn, D. U. *Human Physiology*, 3rd ed. Pearson - Benjamin Cummings, San Francisco, 2003.

[6] Thomson, D. J. *Scientific Spectrum Estimation: Advanced Multitaper Methods of Time-Series Data Analysis*. Queen's University, 2005. Draft.

A Program Code

The code for the project is written in Microsoft Visual C++, using Microsoft Visual Studio 2005. This language was chosen because it was available through MSDN Academic Alliance Program from the Queen's ECE department, and provides a somewhat user-friendly programming interface for visual programming. C++ is a relatively efficient language, which was required for real-time programming.

A.1 Disclaimer

This program is intended for educational purposes only. Be aware that some elements of the code are taken from other sources, outlined in the next section. Although we may know how to program, we are not programmers. The code is provided 'as-is'. It may not follow standard programming conventions, and may contain errors.

Copied Code

In order to interface with the sound-card, Microsoft DirectSound was used, which is part of the Microsoft DirectX SDK package. This is available for download from the Microsoft DirectX website. With this package came a set of example projects. The sound card interface code in SAM was based on the project entitled "FullDuplexFilter".

Several of the classes involved in plotting are modified from a freeware example project entitled "Frequency Analyzer", which can be accessed at the following URL:

<http://www.relisoft.com/Freeware/recorder.html>

Instead of writing a routine to factor a polynomial over the complex plane, the GSL open-source math library was used. This is under the GNU General Public License, and can be obtained from:

<http://www.gnu.org/software/gsl/>

A.2 Compiling the Code

In order for the code to compile, make sure to include the 'include' directories for GSL and Microsoft DirectX SDK, as well as the library directories. If the project is opened through the solution file 'Thesis.sln', most settings should be set. The path names will need to be changed, however, to fit your directory structure.

A.3 The Code

Due to the size of the code, full source code will not be included in this report. Instead, only the header files for customized classes and sets of functions are included. Full source code is attached on a CD.

```
\*==========================================================================================
DSPAudio.h

Description: Provides interface to sound card.  This contains code taken from Microsoft DirectX's
SDK, 'FullDuplexFilter', and was modified to provide a simple interface

Initialize:
DSPAudio *MyAudio = new DSPAudio();
MyAudio->SetFormat(...);
DSP->AddHandler_Filter(...);  //adds a Filter Function that will run after every block of samples
DSP->Start();

==========================================================================================*/


#include "dxstdafx.h"

class DSPAudio{

public:
DSPAudio();
~DSPAudio();

private:

//sets the wave format
static void SetWaveFormat(int nSamplesPerSecond, int wBitsPerSample, int nChannels,
WAVEFORMATEX *wfx);
HRESULT InitDirectSound(); //initializes the directsound library
HRESULT FreeDirectSound(); //frees the resources

HRESULT CreateOutputBuffer();  //sets up the buffer that plays through the soundcard
//if the buffer was stopped, will reset it
HRESULT RestoreBuffer( LPDIRECTSOUNDBUFFER pDSBuffer, BOOL* pbRestored );
HRESULT HandleNotification();

//callback function to use as filter
HRESULT (*TransformData)( VOID* pbOut, VOID* pbIn, DWORD dwLength );
HRESULT SetBufferFormats( WAVEFORMATEX* pwfxInput, WAVEFORMATEX* pwfxOutput,
DWORD nNotifySamples);
HRESULT StartBuffers();
void TimerCallback(); //checks if callback event triggered

public:

//Adds a filtering process
void AddHandler_Filter(HRESULT (*)( VOID* pbOut, VOID* pbIn, DWORD dwLength ));
```

```
void Start(); //starts process
void Stop(); //stops process

//Sets the format; Fs = 8000,16000,44100,48000 wBitsPerSample = 8,16, nChannels = 1,2
void SetFormat(int Fs, int wBitsPerSample, int nChannels, int nBlockLength);

private:

#define NUM_PLAY_NOTIFICATIONS  16
#define NUM_BUFFERS     (16)
#define MAX(a,b)        ( (a) > (b) ? (a) : (b) )
#define SAFE_DELETE(p)  { if(p) { delete (p);     (p)=NULL; } }
#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }


//These were initialized to NULL in original program
LPDIRECTSOUND               g_pDS;
LPDIRECTSOUNDCAPTURE        g_pDSCapture;
LPDIRECTSOUNDBUFFER         g_pDSBPrimary;
LPDIRECTSOUNDBUFFER         g_pDSBOutput;
LPDIRECTSOUNDCAPTUREBUFFER g_pDSBCapture;
LPDIRECTSOUNDNOTIFY         g_pDSNotify;

//notification in 'play' buffer
DSBPOSITIONNOTIFY    g_aPosNotify[ NUM_PLAY_NOTIFICATIONS ];
HANDLE               g_hNotificationEvent;

//buffer sizes
DWORD                g_dwOutputBufferSize;
DWORD                g_dwCaptureBufferSize;

//used when accessing the buffers
DWORD                g_dwNextOutputOffset;
DWORD                g_dwNextCaptureOffset;

DWORD                g_dwNotifySize; //byte size until next notification
WAVEFORMATEX         g_wfxCaptureWaveFormat; //Format of capture buffer


//My created variables
bool bFilterSet; //set to true if user supplies a filter;
bool bPlaying; //set to true if DSPAudio has started but not stopped
MMRESULT m_timerID; //timer that checks if any event occured

};


\*=============================================================================================
Library.h

Description: Contains classes for vowel library management, and has public functions used in
trianing
```

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


#include "MathStuff.h"
#include <fstream>
#include "windows.h"

const double VAR_OFFSET = 0.05; //prevents from dividing by zero in metric
const int ALIGN_ITERATIONS=2; //number of iterations to perform in pole alignment algorithm
const double SD_FACTOR=2; //to keep 90% of points, roughly
const double SD_FACTOR_AVG=1.2; //to keep 80% of points, roughly


//Stores all information for a particular vowel sound
class LibEntry{

public:
LibEntry(){ Full = false;}
~LibEntry();

bool Fill(PComplex *Poles, PComplex *Weights, int numPoles, double *FilterCo,int FilterOrder,
double D, wchar_t *EntryName);
bool Clone(LibEntry *CloneThis); //clones vowel
double Compare(complex<double> *Poles,int numPoles); //returns angle metric
double Check(complex<double> *Poles, int numPoles); //returns magnitude metric
double Compare(PComplex *Poles,int numPoles);
double Check(PComplex *Poles, int numPoles);

//AR model
int nOrder; //AR order
double *a; //filter coefficients
double D; //distortion from LPC, introduced to adjust gains

//Top library poles
int nPoles; //number of poles in library entry
PComplex *W; //weights associated with poles (Note: PComplex contains angle and magnitude)
PComplex *Pole; //poles

wchar_t *Name; //name of model

private:
bool Full; //for memory-freeing purposes

};


//Collection of vowels
class Library{

public:
Library();
~Library();

bool Fill(LPCWSTR filename); //fills from a SAM file
```

```
bool AddEntry(LibEntry *NewEntry); //adds a vowel to the lib
bool Write(LPCWSTR filename); //write the whole library to a text file

int Find(complex<double> *Poles, int numPoles); //tries to match a vowel to input

PComplex eps;

int NumEntries; //number of vowels
LibEntry *Vowel; //the vowels
bool *VowelSearch; //boolean for whether we're searching for particular vowel
int *VowelReplaceWith; //collection of indices for replacing vowels

private:
bool Full; //for memory clearing purposes

};

//trains system
void Train(double **LPCCoeffs, PComplex **Roots, int numpoles, int numblocks,int LibLength,
wchar_t *Name, LibEntry *NewEntry);

//returns a subset of supplied complex roots, arranged by angle
void FirstNRoots(PComplex **Roots, int numblocks,int LibLength, PComplex **FirstRoots);

//averages poles
void avg(PComplex **FirstRoots,int numblocks,int LibLength,PComplex *Poles, double *vradius,
double *vtheta);

double getPowerSum(PComplex *Poles, int numpoles); //returns 'total power' in roots

//calculates weights (both angle and magnitude)
void getWts(PComplex *Poles, PComplex *Variances, int LibLength, PComplex *Wts);

//averages LPC coefficients
void avgLPC(double **LPCCoeffs,int numblocks, int numpoles, double *LPCAvg);

//re-arranges complex roots so those in upper-half plane are at front
int rearrange(PComplex *Roots,int numroots);


\*============================================================================================
MathStuff.h

Description: Contains public functions for performing various mathematical tasks

============================================================================================*/

#include "math.h"
#include <complex>
#include <vector>
#include <gsl/gsl_poly.h>
using namespace std;

#ifndef CONSTS_H_INCLUDED
```

```
#define CONSTS_H_INCLUDED

const double PI = 3.1415926535;

//custom for complex number in polar form
struct PComplex{
double r;
double theta;
};


//attaches variances to make easier for sorting while training
struct PComplex2{
double r;
double theta;
double varr;
double vartheta;
};

#endif // include guard

double round(double x, int numdecimals);

//forms a polynomial from a set of roots, whether real or complex
int formpoly(const double *r, int n, double *a, int numdecimals, bool REAL);

//These two are used in forming the polynomial coefficients given a set of roots
void complexgroupmultiply(const double *r, double *result, double factorR, double
factorI, int n, int depth, int pos);
void realgroupmultiply(const double *r, double *result, double factor, int n,
int depth, int pos);

//evaluates a polynomial
double evalpoly(double *a, short nOrder, double x);

//evaluates power given an AR model
double evalARpow(double *a, short nOrder, double w);

//returns the roots of a polynomial with real coefficients, uses GSL
void getRoots(double *a, int order, double *z);
void getRoots(double *a, int order, complex<double> *z);
void getRoots(double *a, int order, PComplex *z);

//changes complex number in the form of a double array to complex<double> type
void toComplex(double *zin,int array_size, complex<double> *zout);
//changes complex number in the form of a double array to customized PComplex type
void toPComplex(double *zin, int arraysize, PComplex *zout);

//Sorting

//extracts poles in upper-half plane, arranges by magnitude, takes nTopPoles of these,
// then sorts by angle
int sortPickyPoles(complex<double> *z,int array_size, double MinAngle, double MaxAngle,
int nTopPoles);
```

```
//sorts complex type using supplied sorting function
void sortComplex(complex<double> *z,int array_size, int (*compar)(const void *, const void*));


int ThetaRU(const void *z1, const void *z2); //sort increasing by angle, equal angles sorted by r
int ThetaRD(const void *z1, const void *z2); //sort decreasing by angle, then by r
int RThetaD(const void *z1, const void *z2); // ...decreasing ... r, ... angle
int RThetaU(const void *z1, const void *z2); // ...increasing ... angle, ... r
int RThetaDPComplex(const void *z1, const void *z2); // ...decreasing ... r, ... angle
int RThetaDPComplex2(const void *z1, const void *z2); // ...decreasing ... r, ... angle
int VThetaRUPComplex2(const void *z1, const void *z2); // Variance...decreasing ... r, ... angle
int VRThetaUPComplex2(const void *z1, const void *z2);
int ThetaRUPComplex(const void *z1, const void *z2); //sort by increasing angle, then by r
int ThetaRUPComplex2(const void *z1, const void *z2); //sort by increasing angle, then by r

//conversion of types
void ToPComplex2(PComplex *Array, PComplex *Var, int nSize, PComplex2 *Result);
void ToPComplex(PComplex2 *DecomposeThis, int nSize, PComplex *Array, PComplex *Var);
void CopyPComplex(PComplex *From, PComplex *To, int nSize);

//euclidean distance between two complex numbers
double Dist(PComplex *A, PComplex *B);
double Dist(complex<double> *A, PComplex *B);




\*=============================================================================================
SignalStuff.h

Description: Contains public functions for performing various signal-related tasks

=============================================================================================*/

#include "MathStuff.h"

//calculates AR-model
double LPC(short *s,int n, int p, double *r, double *a); //LPC with supplied autocorrelation
double LPC(double *s, int n, int p, double *a);
double LPC(short *s, int n, int p, double *a);

void covariance(double *s, int n, int p, double *r); //actually performs autocorrelation
void covariance(short *s, int n, int p, double *r);

//forms the filter difference equation, generating an output
//underrun is stored samples from the previous iteration to account for convolution near endpoints
void RFilter(short *x, short *y, int buffsize, short **underrun, int underbuffsize, double *num,
double *den, int order, double Gain);
void RFilter(short *x, short *y, int buffsize, short **underrun, int underbuffsize, double *num,
int p, double *den, int q, double Gain);

//performs the fourier transform on data, pre-calculates all complex exponentials for optimization
```

```
class FourierTransformer{
public:
FourierTransformer(double SampleFreq, double FreqResolution);
~FourierTransformer(){}

public:
void FFT(double *data, int nSamples, complex<double> *result);
void FFT(double *data, int nSamples, double *power);
void FFT(short *data, int nSamples, complex<double> *result);
void FFT(complex<double> *data, int nSamples, complex<double> *result);
void FFT(short *data, int nSamples, double *power);

void IFFT(complex<double> *data, complex<double> *result);
void IFFT(double *data, int NumPoints, double *result);
void IFFT(complex<double> *data, double *result);

double GetFs(){return Fs;}
double GetFreqRes(){return fRes;}

private:
int nPoints;
    double Fs;
double fRes;
int Log2Points; //binary logarithm of # of points

vector<vector<complex<double>>> W; //complex exponentials
vector<int> BitMap; //bit-reversal algorithm

};
```