2005

Prosody in text-to-speech synthesis using fuzzy logic

Jonathan Brent Williams
West Virginia University

Prosody in Text-to-Speech Synthesis Using Fuzzy Logic

by

Jonathan Brent Williams

**Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of**

**Master of Science
in
Electrical Engineering**

Approved by

**Dr. Norman Lass, Ph.D
Dr. Roy S. Nutter, Ph.D
Dr. Powsiri Klinkhachorn, Ph.D., Chair**

**Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia 2005**

Keywords: Text-to-Speech, Fuzzy Logic, Intonation, Prosody, Neural Networks

Abstract

Prosody in Text-to-Speech Synthesis Using Fuzzy Logic

by

Jonathan Brent Williams

For over a thousand years: inventors, scientists and researchers have tried to reproduce human speech. Today, the quality of synthesized speech is not equivalent to the quality of real speech. Most research on speech synthesis focuses on improving the quality of the speech produced by Text-to-Speech (TTS) systems. The best TTS systems use unit selection-based concatenation to synthesize speech. However, this method is very timely and the speech database is very large. Diphone concatenated synthesized speech requires less memory, but sounds robotic. This thesis explores the use of fuzzy logic to make diphone concatenated speech sound more natural. A TTS is built using both neural networks and fuzzy logic. Text is converted into phonemes using neural networks. Fuzzy logic is used to control the fundamental frequency for three types of sentences. In conclusion, the fuzzy system produces f0 contours that make the diphone concatenated speech sound more natural.

Acknowledgements

I would like to express my gratitude to Dr. Klinkhachorn for his guidance and invaluable advice throughout my project. I would like to thank him for believing in me and for willingly being the chair of my committee. And I would also like to thank him for his patience when my initial thesis topic did not come to fruition. I would also like to thank Dr. Norman Lass for being on my committee and for his guidance throughout my project. His book, "Principles of Experimental Phonetics", guided me in the right direction and was an important asset to my research. I would also like to thank Dr. Roy Nutter for being a member of my committee and for providing me with valuable career advice.

This thesis is dedicated to my late grandparents, Mrs. Madeline Pleasant and Mr. Brent Pleasant. Without them and their sacrifices, I would not have this opportunity.

Special thanks to Dr. Peter Simeonov for providing me with a research assistantship at NIOSH that helped me complete my research.

Table of Contents

List of Figures

Chapter 1: Introduction and Background

1.1 Introduction

Text-to-Speech (TTS) is the process of converting unknown text into sounds that represent the text. Reading out loud is an example of TTS. A TTS system involves converting random text into intelligible synthesized speech. The text can be either directly introduced to the system by a user or scanned from a source [16]. The applications of TTS systems are numerous, from assisting people with disabilities to improving customer service. The ideal TTS system would sound similar to HAL-9000 from the movie, "2001: A Space Odyssey". However, current TTS systems still sound robotic and thus have yet to gain public acceptance. Today, the main problem facing TTS systems is refining the naturalness of synthesized speech. Naturalness of speech is correlated to the prosody of speech. Prosody refers to the intonation, timing, and vocal stress of speech. Currently, TTS research focuses on the improvement of synthesized speech prosody.

1.2 Human Speech Process

Synthesizing human speech is difficult due to the complexity of human speech. The production of human speech involves the lungs, the vocal folds, and the vocal tract (oral cavity, nasal cavity, and pharyngeal cavity) functioning collectively. Figure 1-1 shows the organs used in speech production.

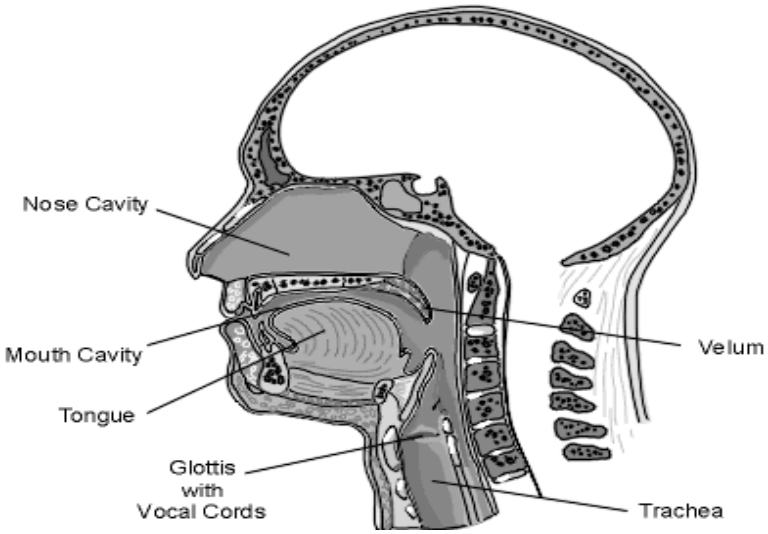


Figure 1-1: Human Speech Organs [40]

Human speech is created by an air source (lungs and the surrounding muscles) causing some type of excitation in the vocal system (vocal folds and vocal tract). The type of sound produced is determined by the vocal system's affect on the air flow. There are two types of speech produced by humans; voiced and unvoiced. With voiced speech, sound is produced from the vibration caused by air flowing through tensed vocal folds. Unvoiced speech is created from air flowing through abducted vocal folds, and the sound is produced by air flowing through a constriction in the vocal tract or air being stopped and then suddenly released [40].

Mimicking the sounds created by human speech is difficult because real continuous speech is a combination of many complex audio signals. With voiced speech, the speech signal is modified by either the oral cavity or the nasal cavity. These cavities act as resonators with pole and zero frequencies. Pole and zero frequencies are called formant and anti-formant frequencies, respectively. These frequencies have their own amplitude and bandwidth. Voiced speech also produces a complex quasi-periodic pressure wave from an interruption in air flow caused by the vibration of the vocal folds. The frequency of impulses from the pressure wave is called the fundamental frequency. With purely unvoiced speech, since there is no vibration of the vocal folds, there is no fundamental frequency.

Researchers believe that the most important signals generated by human speech are the formant frequencies and the fundamental frequencies. When synthesizing speech,

these signals greatly contribute to the naturalness of the speech. The formant frequency represents the shape of the sound that is formed by the vocal tract (oral cavity and the nasal cavity). Different sounds (vowels, nasals, etc.) within a language are distinguishable by their formant frequencies. The fundamental frequency determines the pitch of the voice. For example, women and children have a higher pitch (i.e. higher fundamental frequency) than men. Throughout the years, research involving the naturalness of synthesized speech has focused primarily on these two acoustic features.

Sounds created by humans are merely noise if the sounds do not have meaning. Sounds in speech production are categorized into units. These units can be as large as words or as minute as a phone. However, phonemes are the fundamental units of phonology. The definition of phonemes is the theoretical unit of sound that can distinguish words [29]. The concatenation of phonemes produces the words in the language, i.e. changing a phoneme means changing the word. Phonemes are split into two major categories: vowels and consonants. All vowels are voiced sounds while some consonants are voiced sounds and some are unvoiced sounds. The number of phonemes on a language depends of the actual language, speaker, and the particular dialect. For example, most standard American English consists of 41 phonemes. Diphones are the stable middle region between two phonemes. Diphones represent the transition between two phones. Therefore, the number of diphones in a language is the number of phonemes squared. With speech synthesis, the role of phonemes and diphones is to focus on sounds that the system should yield.

1.3 Speech Synthesis Techniques

Speech synthesis is the artificial production of human speech. The techniques of speech synthesis are categorized into three different approaches. These are concatenative speech synthesis, formant-based synthesis, and Hidden Markov Model synthesis.

1.3.1 Concatenative Speech Synthesis

Concatenative speech synthesis involves combining previously recorded speech to form words or sentences. The concatenative speech synthesis approach currently

produces the most natural sounding speech because of the use of real speech. There are three methods of concatenative speech synthesis: diphone synthesis, domain-specific synthesis, and unit-selection synthesis. The most commonly used methods today are domain-specific and unit-selection synthesis.

Diphone synthesis involves the concatenation of diphones. As previously explained, diphones are the middle region between two phones. The concatenation of diphones minimizes the co-articulatory effects of the phone to phone transition. During synthesis, digital signal processing (DSP) techniques such as linear predictive coding (LPC), POLA, and MBROLA are used to overlay the desired prosody onto these diphones. The amount of diphones for a given language is equal to the number of phonemes of that particular language squared. Therefore, the diphone inventory size is relatively small and diphone synthesis can be used with inexpensive processors and embedded systems. However, diphone concatenation yields robotic sounding speech due to the digital signal processing.

Domain-specific concatenation is a very common form of speech synthesis. Most companies use domain-specific synthesis for their phone-based customer service systems. Consequently, this type of synthesis has gained some notoriety over the years. Domain-specific synthesis concatenates pre-recorded words and phrases to create speech. Domain-specific synthesis is only useful in applications where the output is restricted to a certain domain. Examples of domain-specific synthesis are automatic reports of the weather, talking gadgets, automated banking, and automated customer service. Domain-specific concatenation sounds natural because the variety of sentence type is limited; therefore, the output matches the original recording. However, since this method of synthesis is limited to only specific applications, domain-specific synthesis could not be implemented in a TTS system.

Unit selection-based synthesis is the method currently used in commercial-based TTS systems. Unit selection is created from large databases of hours of recorded speech. These recordings are then converted into units to be concatenated into speech. First, a recording of speech is segmented into individual phonemes, syllables, words, phrases, and sentences. The segmentation is done by hand or automatically by a modified speech recognizer. An index of the units from the speech database is then created based upon

prosodic parameters, like fundamental frequency and segment duration. Synthesis is created using a decision tree that determines the best path of candidate units from the database. Unit selection yeilds the most natural sounding speech today. However, to achieve maximum naturalness requires unit selection-based systems' speech databases to be gigabytes in size [36].

1.3.2 Formant Synthesis

Formant synthesis generates synthetic speech using formants. Formants represent the resonant frequencies of the oral cavity or the nasal cavity. An acoustic model is used to create the speech and parameters, like fundamental frequency, spectral components, and noise levels which are varied over time to create a waveform of speech. The disadvantage of formant synthesis is the robotic sounding quality of speech [36]. However, formant synthesis does have many advantages over concatenative speech.

Formant synthesis does not require stored databases and thus requires little memory. Formant synthesis can also be generated at very high speeds. This makes formant-based systems useful in embedded computing and real-time applications. For example, most reading machines for the blind use formant-based synthesis. In addition, formant-based systems offer the user more control over the output speech. With formant synthesis, intonation and prosody can be altered to represent an assortment of emotions and tones of voice.

1.3.3 Hidden Markov Model Synthesis

Hidden Markov Model (HMM) synthesis is the process of modeling the speech output using the Markov process. The HMM is a Markov process where the parameters are unknown. With HMM, each state has outputs, and future states are dependent only on the present state [36]. HMMs are used in speech synthesis to model the vocal tract and prosody. The quality of HMM speech synthesis is good but not as good as unit selection-based systems [36]. HMM synthesis is a newer method of speech synthesis and has been applied mostly to trainable TTS systems.

1.4 Methods of Text-to-Phoneme Conversion

In order for a TTS system to be accepted by the public, the system must correctly convert the inputted text into the correct pronunciation. Text-to-Phoneme (TTP) conversion consists of translating text into its phonetic transcript. This task can be accomplished using many different methods. The most common methods are the dictionary-based approach, the rule-based approach, and the machine learning approach.

1.4.1 Dictionary-based Approach

The dictionary-based approach is the easiest TTP conversion method to implement. Dictionary-based TTP conversion consists of storing phonological knowledge into a lexicon [16]. Early dictionary-based conversions required locating word pronunciations that were stored in a large lexicon. This process required enormous amounts of memory because the lexicons had to contain every word and its pronunciation. Most dictionary-based TTP conversions today use stored morphemes instead of words in the lexicon. Morphemes are the smallest language unit that carries a semantic interpretation. For example, the word "uncontrollable" contains three morphemes; "un-", "-control-", and "-able". Morphemes require less memory to store and can cover most words in a language. Both types of dictionary-based conversion methods handle unknown words similarly. Rules are used to pronounce unknown words or morphemes [16]. The main drawback of dictionary-based TTP conversion is the amount of memory that the lexicon can consume. Yet, dictionary-based systems are the easiest to create.

1.4.2 Rule-based Approach

The rule-based TTP conversion approach uses expert rules to yield pronunciations. The pronunciations are based on the spelling of the word. These rules are similar to the sounding-out rules used by grade school students when learning how to read [36]. There are some drawbacks to the rule-based approach. For instance, rule-based TTP conversion system rules can become very complex, especially with irregular languages like the English language. Rules are much more complex to code than simple

binary searches used by dictionary-based TTP systems. However, rules-based systems can work on any input presented to the system.

1.4.3 Machine Learning Approach

There are many different types of machine-learning approaches to TTP conversion. Machine learning is the ability of a machine, a computer, or an electronic device to improve its performance based upon previous results. Machine learning requires that a system "learns" how to convert text into its phonetic representation. With TTP conversion, this task can be accomplished with many different methods. Neural networks, Self-Organizing Maps, and Decisions Trees are examples of the machine learning approach. However, for irregular languages like English, the accuracy is not equivalent to the latter approaches.

Chapter 2: Motivation and Objectives

2.1 Motivation

TTS systems are becoming more commercially available as the quality of the systems improves. Most commercial systems use the unit-selection based concatenation approach to produce speech output [36]. However, unit-selection requires large prerecorded speech databases that must be segmented in order to create a useable system [36]. Segmentation of a prerecorded speech database can be very difficult and timely. Formant synthesis uses less amounts of memory, but the speech is very mechanical in sound. For speech synthesis to be widely accepted in robotics and our shrinking electronics, speech synthesis systems must use as little memory as possible, must require little effort to create, and must sound somewhat natural.

Diphone concatenation is adequate enough to produce understandable synthesized speech. Most languages consist of about 2000 diphones. Diphone inventories take relatively little memory to store, and there are many freely available programs that use diphone concatenation. However, diphone concatenation systems produce speech that is robotic and not the quality of current unit-selection based systems. Prosody is what makes speech sound natural, and the prosody in recorded speech segments is more natural sounding than that of synthesized prosody.

This thesis presents a different method to produce more natural sounding speech for diphone concatenation-based TTS systems. Most TTS systems that use diphone concatenation use either neural networks or rule-based approaches to generate prosody. This research exams the use of Fuzzy Logic to generate one aspect of prosody: fundamental frequency. TTP conversion is performed using the neural network Back Propagation algorithm. Using linguistic data as input, the fuzzy logic system produces fundamental frequency and frequency contour as outputs. The rules of the fuzzy logic system are from the O'Shaughnessy fundamental frequency algorithm used in the MITalk system [2]. As a result, the TTS system output speech should sound more natural.

2.2 Research Objectives

The research objective is to apply the flexibility of fuzzy logic to the randomness of TTS conversion in order to produce more natural sounding synthesized speech. Using established information provided by the user of the TTS system, the fuzzy controller will produce output that will reflect the user's input.

The specific research objectives are:

1. Gain an understanding of the process of human speech production. Understand the concepts and theories of generating synthetic speech. Research different algorithms that have been created to generate English prosody automatically.

2. Build an English text-to-phoneme system using neural networks that will produce accurate phonemes given inputted text. The Back Propagation algorithm will be used to train the network. The training set will consist of about 1800 words.

3. Implement a fuzzy control system that will control the intonation of the speech. The fuzzy controller will receive different linguistic parameters as the input and produce fundamental frequency as the output. This fuzzy controller will represent the O'Shaughnessy fundamental frequency algorithm and system will handle three types of sentences: declarative, yes/no question, and interrogative questions. The final system should produce understandable speech that mimics the proper intonation for each type of sentence.

4. Calculate the accuracy of the text-to-phoneme network using the training set and unknown text. Evaluate the fuzzy controller by comparing the f0 contour produced by the controller with the f0 contour of the Microsoft Research Speech Technology Asia (MRSA) on-line TTS system [37].

Chapter 3: Literature Review

TTS research has been extensively examined over the last 40 years. This chapter presents an extensive literature review of past and present research on the different aspects of the TTS system. The review has been performed on the following subjects: generation of synthetic speech, TTS conversion algorithms, TTS systems, automatic prosody generation, and the fuzzy logic inference system.

3.1 Generation of Synthetic Speech

Generating synthetic speech has been a curiosity for the past 1100 years. Around the year 1003, Gerbert of Aurillac created the first known mechanical talking machine. For the next two centuries, inventors like Albertus Magnus and Roger Bacon created machines know as "talking heads" [23]. However, the first known machine that tried to mimic real human speech was developed by Christian Kratzenstein of St. Petersburg in 1779. This machine could produce five long vowel sounds. Twelve years later, Wolfgang Von Kempelen developed a machine that could produce vowels and some consonants [21, 33].

Figure 3-1: Wolfgang Von Kempelen's Talking Machine [33]

With the start of the 20th century and the increasing use of electricity, speech synthesis began to move from mechanical machines to electrical machines.

Electronic speech synthesizers were first developed in the 1920's. The first known electronic synthesizer, VODER, was developed by Homer Dudley in the late 1930's [21]. Dudley was a research physicist at the Bell Laboratories in New Jersey. Dudley reconstructed the Bell Laboratories speech analysis, VOCODER, into the speech synthesizer VODER. The VODER was controlled by an operator using a keyboard to

Figure 3-2: Woman Operating the VODER [36]

adjust the filter output, foot pedals to control the fundamental frequency, and special keys to create closure and the release required for stops [23]. The VODER was operated like a musical instrument. Eventually, human operated machines became obsolete. After World War II, the spectrograph provided a new tool for researching acoustic phonetics. As a result, researchers began to study speech based on acoustical data.

In the 1950's, speech synthesizers like the Pattern Playback were developed to produce speech from copied speech waveforms. The speech synthesis by rule approach (formant synthesis) began to become prevalent in the following decade. Concatenative speech synthesis became a focus of research in the 1970's with the initial focus on phoneme concatenation. However, it was quickly discovered that diphone concatenation would be more feasible than phoneme concatenation [21]. In 1976, Olive and Spickenagle used linear prediction speech analysis to automatically create a full diphone inventory for concatenation [27]. In 1988, Nakajima and Hamada wrote about a method of speech concatenation that used a unit-selection based approach, instead of the more

common diphone concatenation approach [25]. Today, diphone concatenation and unit-selection concatenation are the most common methods of speech synthesis with the latter becoming more common commercially.

3.2 Text-to-Phonemes Algorithms

Today, there are many different ways to produce speech from text. One of the earliest methods of converting text to phonetics was the use of sophisticated heuristics [21]. The first full English TTS system used this method in combination with a syntactic analysis module. As computer memory increased, the preferred method for TTP conversion was the use of a look-up dictionary. Look-up dictionary algorithms consist of matching inputted words to words in a lexicon, utilizing phonological rules as a back-up. By the 1980's, TTS systems like the KlatTalk and DECTalk used a combination of a look-up dictionary and phonological rules [2, 21, 22]. As computer technology became more sophisticated and accessible, researchers began to develop new ways of tackling the TTP conversion problem.

The use of machine learning techniques for TTP conversion was researched in the 1980s. In 1987, Sejonowski and Rosenberg used neural networks to convert inputted text into phonemes. This system used a 120 hidden neuron multi-layered perceptron trained with 2000 words [35]. Neural networks use neurons with weights that represent the neurons and the synapse of the neurons, respectively. Each weight represents the firing strength of the neuron synapse. Initially, all of the weights for each neuron in the network are randomized. The weights are updated using the calculated error from the training data and the network's actual output. In the multi-layered perceptron algorithm, the error is propagation backwards throughout the layers of network. Figure 3-3 shows an overview of the NetTalk network.

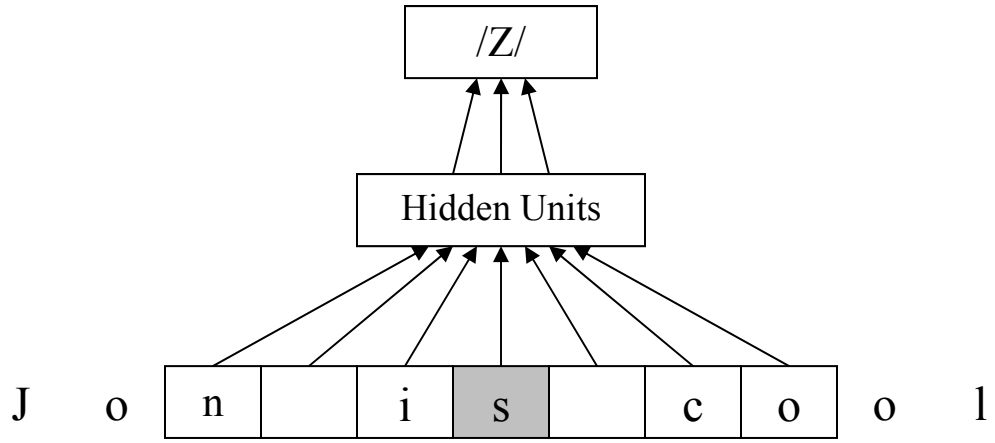


Figure 3-3: NetTalk Multi-layered Perceptron

McCulloch devised a re-implementation of the NETTalk multi-layer perceptron algorithm called NETSpeak, which yielded about 85% accuracy [24]. Recently, Arciniegas and Embrechts used a staged neural network algorithm to handle some of the previous problems of the single-staged multi-layer perceptron algorithm. In that paper, two stages of neural networks were used to convert text to phonemes. One stage separated regular words from special words. Special words were categorized as words that contained single letters represented by two output phonemes. The other stage found the phonetic output for the two types of words [4]. Gubbins used a hybrid-neural network approach which used both neural networks and a simple rule-based system to convert text to phonemes [18]. The multi-layer perceptron neural network is not the only neural network algorithm researchers have used for TTP conversion.

Adamson and Damper researched different ways to improve the performance of TTP conversion neural networks [24]. They used a recurrent neural network which addressed some of the problems of the multi-layer perceptron algorithm. A recurrent neural network uses a single letter as input and trains itself using the Back Propagation through Time algorithm. This algorithm removes the need for alignment of the training set. The network has a recurrent structure with no constraints regarding which direction units can interact [19]. Adamson and Damper's algorithm initially performed worse than the NETTalk and NETSpeak, but their paper was the foundation for other recurrent neural network based algorithms [1]. In 2004, Bilcu, Astyola and Saarinen improved the performance of the recurrent neural network algorithm by using three letters as the input

to a recurrent neural network [6]. In 2003, Bilcu compared the performance of multi-layered perceptron to two types of recurrent networks. The results showed that the multi-layered perceptron is the most accurate of the three [7]. Besides neural networks, TTS systems have also used other forms of machine learning to convert text to phonemes.

The pronunciation by analogy (PBA) algorithm is another machine learning algorithm that has been used for TTP conversion. PBA is similar to the dictionary-based algorithms because both algorithms use dictionaries to convert text to phonemes. However, the PBA algorithm uses a different method to handle words not in the dictionary. The PBA algorithm concatenates partial pronunciations of substrings using learned phonological knowledge [12]. Using an aligned lexicon, pronunciation can be achieved through explicit and implicit learning. An example of a system that uses this algorithm is the PRONOUNCE system [12, 13]. In the PRONOUNCE system, an inputted word is matched to words in a dictionary. Then substrings with common letters are found between the found dictionary entry and the input. Phonetic substrings are also built from the matched substrings. Information from these substrings is then used to build a pronunciation lattice. Lattice nodes are first labeled with L_i and P_i representing the matched letter and the corresponding phoneme in the substring, respectively. P_i is labeled P_{im} to represent the m_{th} matched substring. If there is a match between two L_i and L_j, then an arch is placed between the two nodes with the P_{im} and P_{jm} being the arc labels. The pronunciation for an unknown string is created by the best path through the lattices [12]. PBA do suffer from one major drawback however. Due to incomplete paths, PBA tend to have silences during text conversion. This problem was solved in Sullivan and Damper, but the pronunciation lattices' size greatly increased [39].

3.3 Text-to-Speech Systems

The first full TTS system was developed in the late 1960's. Since then, there have been many advances in the accuracy and the quality of TTS systems. Companies like IBM, Microsoft, and Bell Labs have developed both free and commercially available systems.

In 1968, Umeda of Japan developed the first demonstrated TTS system for the English language. This system transformed text to phonemes using linguistic rules. Sentence pauses were placed in sentences with ten or more syllables [21]. In 1973, the Haskins TTS system was developed but was later discontinued due to inadequate research funds. This system used the Kenyon and Knott 140,000 word phonetic dictionary with a rule system to handle unknown words. The Haskins TTS system was developed to aid the blind with reading. Although the system was never produced commercially, the Haskins TTS system is considered a significant step in TTS research [21].

In 1976, Allen, Hunnicutt, and Klatt developed the MITalk at MIT [2]. This TTS used different levels to convert text to synthesized speech. In the first level, abbreviations, numbers, and symbols were transformed into words. Then, using a 12,000 morph (prefixes, roots, and suffixes) lexicon, words were converted to their phonetic equivalent. Words not in the lexicon were converted to phonemes by using rules. Stress and "part of speech" for each word was determined on another level. Then the final level produced the synthesized speech. Phoneme, syllabic, and pause duration were determined using the Klatt duration rules [20, 2]. Fundamental frequency contour was determined using an adaptation of the O'Shaughnessy algorithm [28, 2]. The f0 contour was smoothed, and the waveform was generated using a terminal synthesizer.

The Klatttalk TTS system was developed in 1983 by Dennis Klatt. Dennis Klatt had previously worked on the MITalk system a few years earlier. This system used the Hunnicutt letter-to-phoneme rule system plus an exception dictionary to convert text to speech. The Klattalk was more rule-based than the dictionary-based MITalk system [22]. The Klattalk system was the basis for the 1982 Digital Equipment Corporation DECTalk. The DECTalk system later became commercially available in 1983. DECTalk was very versatile because of its ranges of voices and different speech speeds. Due to the flexibility of the DECTalk hardware, the DECTalk was easily updated with improved versions of the Klattalk system [21].

Figure 3-4: Stevie Wonder Introducing the DECTalk in 1983 [41]

In the early 1980's, Richard Gagnon developed a very inexpensive segmental synthesis program. This system became the commercial Votrax Type-n-Talk. The system was built with very inexpensive hardware and a small phoneme inventory to synthesize speech. The Echo TTS system was another inexpensive segment-based system which used a diphone inventory and a linear predictive synthesizer to produce speech [21].

Today, there are many types of TTS systems for many different purposes. The Festival Text-to-Speech system developed by Black and Clark is an example of current TTS systems developed for research. Festival uses letter-to-sound rules and a large lexicon for TTP conversion. Speech synthesis is accomplished using unit-selection concatenation of diphones [8]. The MBROLA project is a TTS system that is freely available for researchers. The MBROLA project was developed by TCTS Lab of the Faculté Polytechnique de Mons in Belgium and is a back-end system [15]. This system is to be used as a speech producer for a TTS system developed by the user. MBROLA uses diphone concatenation to produce speech for many different languages. Microsoft has many English TTS systems for different applications. Microsoft based products use either unit-selection or diphone concatenation to synthesize speech. Microsoft Reader is free software from Microsoft that converts text from e-mails and other documents to

speech. The Apple PlainText TTS system is standard on PowerPC computers. The PlainText uses a dictionary-based system to convert text to sounds and diphone concatenation to produce speech. A very high quality commercial system is the AT&T Natural Voices TTS system. The AT&T Natural Voices enables users to define the pronunciation of certain words. This system uses unit-selection synthesis to produce speech [5].

3.4 Automatic Prosody Generation

Currently, prosody is the major issue in speech synthesis. As a result, most research deals with prosody of speech. Prosody, in the context of speech, consists of the properties of speech, such as pitch, loudness, and duration [16]. Prosodic events can be phonemes, syllables, or words. Prosody in a speech system mainly deals with fundamental frequency and segmental duration.

Segmental duration refers to the timing of the units that create speech. These units can be either as small as a phone or as large as a phrase. The size of the units must be determined in order to adequately model real speech. Researchers know that sub-phonetic segments do exhibit different durations. However, this information would be too complex for TTS systems, so most research is focused on either the phonetic or the supra-phonetic sized segments [16]. Early research on segmental duration addresses the principles of isochrones. A speaker would unconsciously use an internal clock while speaking. As a result, Campbell contended that segments in a syllable frame are found using the following formula:

$$Dur_i = \exp(\mu_i + k\sigma_i) \tag{3.1}$$

Dur_i is the duration of the segment at syllable$_i$ and μ_i and σ_i statistical measures of a large corpus. However, this formula is too strict of an application to accurately model duration [15].

More accurate segmental duration models have focused primarily on the phoneme as the segmental unit. Although other segments are taken into account, these segments

do not directly affect the duration of the units. Currently, there are two types of segmental duration models: rule-based models and corpus-based models. Rule-based models use rules to modify intrinsic durations, while corpus-based models use sophisticated methods to automatically derive models using data within the corpus [16].

One of the best known rule-based duration models was developed by Dennis Klatt in 1976. This model has been used in MITalk, Klattalk, and DECTalk TTS systems. Each phoneme consists of an inherent duration and a minimum duration [2]. There are eleven rules that alter the duration of the phoneme due to factors like their location, their manner of articulation, their stress, etc. The duration of the phonemes is changed by using the formula:

$$PRCNT1 = (PRCNT1)*(PRCNT2)/100 \quad (3.2)$$

Where *PRCNT1* represents the current duration of the phoneme, and *PRCNT2* is a number that the phonemes need to be altered [2]. Although this duration model works well, recent research has focused more on the corpus-based approach to duration modeling.

Corpus-based duration models take advantage of the advancements in computational resources. Using a large recorded speech corpus, parameters are extracted and models are created using some type of abstract learning method. In 1994, Riley used a corpus of 400 utterances from a single speaker and 4000 utterances from 400 speakers to model segmental duration. From this corpus, Riley built a classification and regression tree (CART) as the segmental duration model [32]. In 1992, Campbell used a neural network to model segmental duration. In that paper, he computed syllable duration independent of inherit segmental durations. Campbell believed that previous rule-based models were incorrectly based on inherent durations. His focus was to model higher levels of prosodic structure like syllables and prosodic phrases. Consequently, Campbell's network was trained to model Japanese and English syllables [11].

Although segmental duration is a very important aspect of speech synthesis prosody, it is not as essential as pitch and intonation. For example, a modification in pitch can change a statement into a question. Initially, research on this aspect of prosody

mainly focused on the relationship between intonation and stress. Intonation deals with the pattern of tones in an utterance. Stress deals with the emphasized syllable of a word. It was believed that there was a direct association between intonation and pitch and that stress was created by changes in vocal intensity and syllable duration [21]. However, it is now known that the change in the fundamental frequency indicates stress and intonation [21]. Over the years, there have been different theories to predict the rise and fall of fundamental frequency (f0).

In the mid 1960's, Mattingly developed a fundamental frequency theory that used three tunes placed on the last prominent syllable of a clause. The tunes were rise, fall, and rise-fall which correspond to a statement end, a question end, and a continuation rise, respectively. Later in the decade, researchers tried to develop models that mimic the exact fundamental frequency contour of natural speech. In 1969, Öhman stated that f0 contours can be modeled in terms of a discrete signal fed to a linear smoothing filter [21, 26]. A Japanese intonation model by Fujisaki was able to closely match natural intonation contours using the ideas proposed by Öhman [17]. Fujisaki listed two types of events: phrase and accent command [16]. Phrase commands were modeled as a pulse function, and accent commands were modeled as a step function [16]. Hart and Cohen described intonation as a hat pattern. The fundamental frequency will rise on the first stressed syllable of a phrase and then remain high until the final stressed syllable. At the end of the phrase, there will be either a large fall or a fall-rise of the fundamental frequency [21].

The O'Shaughnessy fundamental frequency algorithm was developed in 1979. This rule-based method first assigned peaks to stresses in the sentence. The size of the peak depended on the length of the sentence, the location of the stress, and the importance of the word. Then rises and falls were placed around the peaks. The final rise or fall of the sentence depended on whether the sentence was a statement, a yes-no question, or an interrogative question [2]. The Pitch Contour theory was developed in the late 1960's for British English. This theory splits speech into four components: prehead, head, nucleus, and tail [16].

By the early 1980's, models for fundamental frequency generation began to become more comprehensive and flexible. A well known model was developed in 1984

by Anderson and Pierrehumbert [30]. Anderson and Pierrehumbert believed that stress patterns in a sentence affected the fundamental frequency contour. This model advances on some of the earlier research that used two tones. The model separates intonation into two main tones, a high and a low [30]. These tones are placed on stressed syllables as a single tone or a combination of tones. The sequence of tones is restricted by a three level finite state grammar [16]. With the tones reflecting target points in the fundamental frequency contour, this model does generate good intonation contours. As the 1980's came to an end, researchers began to focus on more statistical and data-driven models.

Some researchers began to focus on the use of classification and regression trees (CART) to generate intonation. These trees were used with different intonation models to automatically generate fundamental frequency for TTS systems. Classification and regression trees consist of a question on each leaf (node) about the feature. The answers from the tree nodes form another sub-tree path. The leaves of these trees contain statistical measures that define the path taken [14]. The Festival TTS system by Dusterhoff and Black uses classification and regression trees in combination with the Tilt Intonation model to generate prosody. The Tilt Intonation model is used to automatically analyze intonation. There are two types of Tilt events: pitch accents and boundary tones [31]. Unlike previous work which used categorical parameters, the Tilt model uses continuous parameters [31]. Using real speech, information is extracted from the database. Then regression trees are built for each parameter in the Tilt model. These trees are used as the fundamental frequency model for the system.

Neural networks have also been used to automatically generate fundamental frequency. In 1989, Scordilis and Gowdy of Clemson University first used neural networks to generate fundamental frequency. Using a small training corpus of real speech, a network was trained to learn f0 values and f0 fluctuations in phonemes [34]. Similar to the NETTalk and NETSpeak systems, this network used the Back Propagation algorithm to train itself. The network consisted of three layers with a hidden layer of 30 neurons. The results revealed that the network could learn to generate fundamental frequency [34]. In 1992, Traber developed a recurrent neural network to predict the number of pitch values in a syllable. The network was trained using automatically labeled data. The network had two hidden layers with the output representing the

different f0 values [10]. Recent systems that use neural networks for f0 generations closely follow the system developed in the early 1990's. Today neural networks have been used to automatically generate fundamental frequency for a large range of languages.

3.5 Back Propagation Algorithm

The Back Propagation algorithm was first developed around the mid 1970's. Learning is based on the gradient descent in the error [38]. This algorithm consists of two main phases: the forward pass and the backwards pass [38]. This algorithm is very similar to Rosenblatt's Perceptron algorithm and is called the Multi-layered Perceptron.

3.5.1 Forward Pass

With the Back Propagation algorithm, the network should first be considered a black box with inputs and a single output.

Figure 3-5: The Black Box

Hidden within the black box are neurons in many different layers. The first layers consist of the input neurons, the middle layers are the hidden neurons, and the last layer is the output neurons. Each layer is interconnected to the surrounding layers.

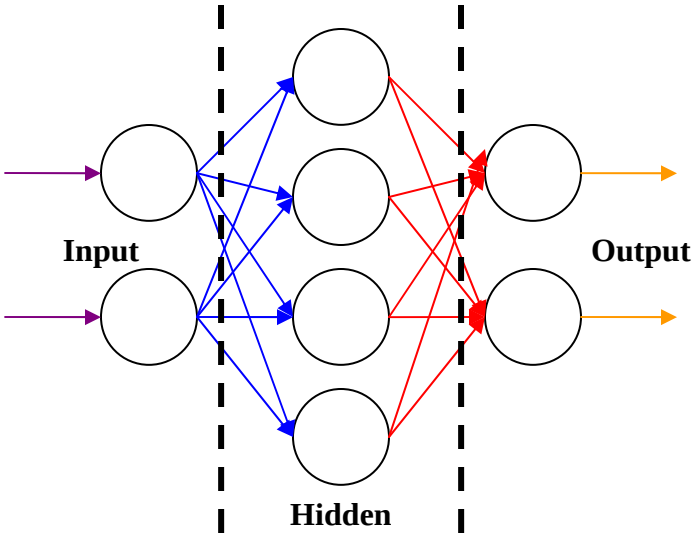


Figure 3-6: Interconnection of Neurons

The outer and input layers are static, while the neurons in the hidden layers can be adjusted. The first step in the Back Propagation algorithm is to create random weights for each neuron in the hidden layers and output layer. The input vector needs to be a vector of ones and zeros. The network is presented with an input vector and a desired output. The hidden weights' outputs are computed using an activation function. The activation function can be sigmoid, tangential, etc.

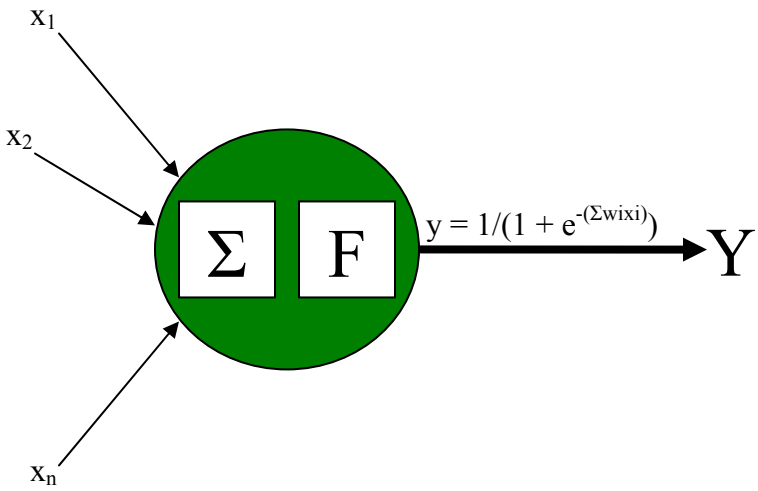


Figure 3-7: The Sigmod Activation Function Neuron

Using the input vector, the hidden layer neuron weights, and the activation function, the hidden layer nodes' output is computed using the equation:

$$y_{km} = 1/(1 + e^{-\Sigma(\text{input vector} * \text{hidden neurons weights})}) \quad (3.3)$$

In the equation above, y_{km} corresponds to the output of the hidden layer. The subscript k represents the number of layers in the system, and the subscript m represents the number of nodes in layer k. The hidden layer outputs now become the inputs to the output layer. The output layer nodes' outputs are calculated using the activation function, weights of each output node, and output from each of the hidden layers node.

$$y_{(k+1)m} = 1/(1 + e^{-\Sigma(\text{hidden layer outputs} * \text{output neurons weights})}) \quad (3.4)$$

where $y_{(k+1)m}$ is the output of the output nodes, which is the actual output of the network. Figure 3-8 shows an example of a network and the different inputs and outputs in the layers.

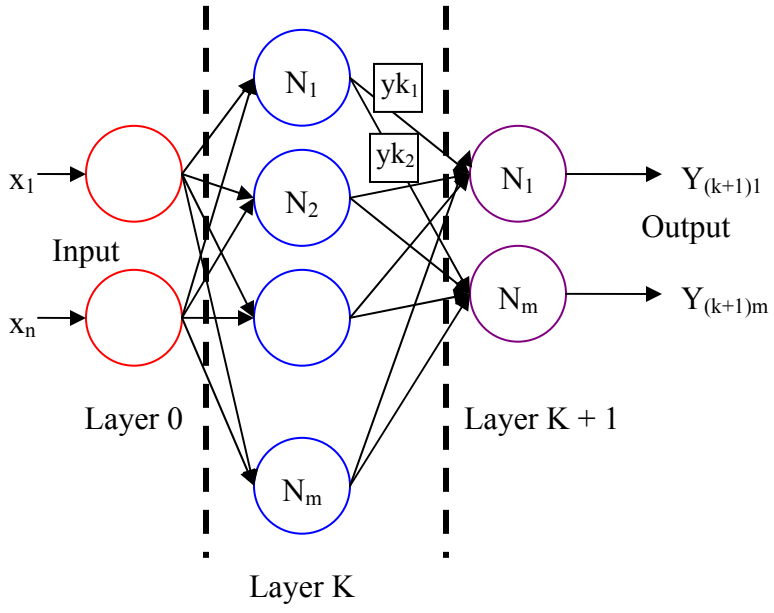


Figure 3-8: K layered Back Propagation Network

Using the actual output and the desired output the Mean Square Error can be determined with the following equation:

$$E = \Sigma (d_m - y_{(k+1)m})^2 \quad (3.5)$$

In this equation, d_m is the desired output for the output nodes and $y_{(k+1)m}$ is the actual output of the output nodes. The next phase of the Back Propagation algorithm can now be computed using the Mean Square Error [38].

3.5.2 Backward Pass

After the error has been calculated, the weights are updated in each layer to fit more closely to the desired output. The output layer weights are updated using the delta rule equation. This involves changing the weights using the gradient of the calculated error. The error term δ for the output layer is:

$$\delta_{(k+1)m} = y_{(k+1)m} (1 - y_{(k+1)m}) (d_m - y_{(k+1)m}) \quad (3.6)$$

where $y_{(k+1)m}$ is the actual output at node m and d_m is the desired output at node m of the network. Using delta, the weights in the output layer can now be updated thus reducing the error of the network.

$$W_{(k+1)m}(t + 1) = W_{(k+1)m}(t) + \eta\delta_{(k+1)m} y_{km} \quad (3.7)$$

$W_{(k+1)m}(t + 1)$ are the new updated weights in the output layer, and $W_{(k+1)m}(t)$ are the old weights in the output layer. η is the learning rate of the network. The learning rate can be a number greater than zero and less than one. If the learning rate is high, the network will converge faster but might be less accurate. However, a low learning rate will cause the network to converge much slower. Also convergence can be changed by using a momentum term α in equation 3.8.

$$W_{(k+1)m}(t + 1) = W_{(k+1)m}(t) + \alpha\eta\delta_{(k+1)m} y_{km} \quad (3.8)$$

Using $\delta_{(k+1)}$ from the output layer, the hidden layer error term δ_k can be computed using the equation:

$$\delta_{km} = y_{km}(1 - y_{km})\Sigma(W_{(k+1)m}\delta_{(k+1)m}) \quad (3.9)$$

In equation 4.7, y_{km} are the hidden layer outputs and W(k+1) are the updated weights in the output layer. After the error term δ_{km} is computed for the hidden layer, the hidden layer weights are updated using δ_{km}, the presented input vector, and the learning rate.

$$W_{km}(t + 1) = W_{km}(t) + \eta\delta_{km}\text{input vector}_m \quad (3.10)$$

$W_{km}(t + 1)$ represents the new hidden layer nodes' weights and $W_{km}(t)$ represents the old hidden layer nodes weights. This process continues until some stopping criterion has been met. Once again, convergence can be changed by using the momentum term α in equation 3.11 [38].

$$W_{km}(t + 1) = W_{km}(t) + \alpha\,\eta\delta_{km}\text{input vector}_m \quad (3.11)$$

3.6 Fuzzy Logic Inference System

Fuzzy logic has its roots in philosophy. In ancient Greece, a group of philosophers wrote the "Law of Thought". One of the laws stated that logic must either be true or false. The seeds of fuzzy logic were later planted by Plato who contended that there could be a middle ground between true and false [9]. About 2500 years later, a Polish philosopher named Jan Lukasiewicz described in detail an alternative to the true/false logic of early Greek philosophy. Lukasiewicz mathematically created a tri-valued logic and then later expanded his theory to four-value and five-value logic. Lukasiewicz's work affirmed that someday logic could be expanded to infinite-value logic [9].

The father of modern fuzzy logic created this infinite-value logic. In 1965, Professor Lotfi Zadeh of UC Berkeley wrote the paper "Fuzzy Sets" which described, mathematically, the theory of fuzzy logic. In this paper, Zadeh described the two opposites, true and false, as membership functions. These membership functions truths are determined through a range of numbers, normally between 0 and 1 [42]. In 1973, Zadeh expanded on this theory, solving complex systems and decision processes [9].

Fuzzy systems are designed to take advantage of imprecision. Solving real world problems with precise logic can be difficult. Logic deals with making an absolute choice, like true or not true. But sometimes the right choice is somewhere in between true and not true. Fuzzy systems add fuzziness to the choice in order to accommodate the grey area of logic.

Fuzzy systems assign truth values to statements to determine their truthfulness. These numbers are usually between 0 and 1. The truth value determines the membership of a certain group. To describe these groups, the fuzzy system uses the membership function. With a given statement, the membership function determines the truthfulness of that statement. The membership function can be many shapes and sizes. Some examples of membership functions are shown in Figure 3-9.

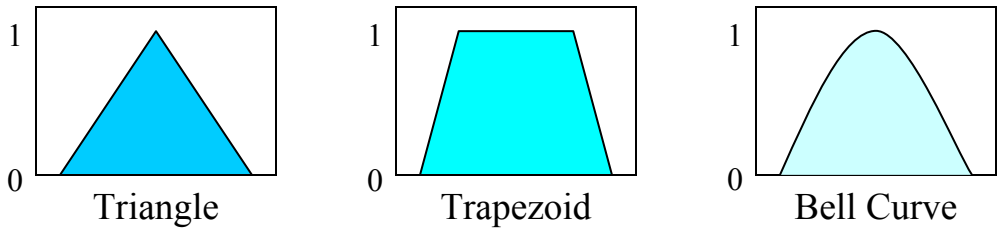


Figure 3-9: Different Types of Membership Functions

The fuzzy system is built with linguist rules and membership functions. The linguist rules are used to conduct the actions of the fuzzy system. The membership functions are used to determine how these rules affect the fuzzy system. The rules are built using the logical terms like, *OR*, *AND*, *NOT*, and *THEN*. Rules can also be built with other terms to hedge behavior like, *MORE*, *LESS*, *VERY*, and *SOMEWHAT*. Some examples of how these terms can be used to build rules are:

"If Jon is tall *OR* Jon is fat *THEN* Jon's size is big."
"If the car is old *AND* the car is cheap *THEN* the car is worthless"
"If the car is *NOT* old *AND* the car is *NOT* cheap *THEN* the car is valuable"

Each term used in the fuzzy system has different effects on the behavior of the system. The *OR* term is the same as the *UNION* of two variables. The *OR* term is equivalent to the maximum. The *AND* term is the equivalent to the *INTERSECTION* of

two variables. The *AND* term is equivalent to the minimum. The *NOT* term corresponds to the *COMPLEMENT* of a variable which is its opposite. Finally, the *THEN* term indicates the consequence produced by the rule.

The descriptive linguist variables in the rules represent the membership of the statement. Therefore, descriptive linguist variables are the membership functions of the fuzzy system. The fuzzy system is constructed of different fuzzy inputs and consequences of the inputs. Each input and consequence of the fuzzy system has its own set of membership functions. When an input is presented to the fuzzy system, the output is determined by the firing strength of the rules onto the input. From the rules and the presented input, a crisp output is calculated to produce the output of the fuzzy system.

Chapter 4: Text-to-Phoneme Conversion

4.1 Text-to-Phoneme Conversion Overview

This chapter presents the algorithm used to convert user's text into phonemes. Phonemes are the smallest phonetic unit in a language within a word. When American students first learn to read, they are taught all of the English phonemes. Table 4-1 gives some examples of phonemes and the sounds that they represent.

Table 4-1: Examples of English Phonemes

Phoneme	**Sound**
/p/	pit
/k/	cat
/t/	tap
/ə/	about
/ĕ/	bend
/ä/	father
/sh/	show
/zh/	measure

In order to convert text into speech, the algorithm must be able to handle words in the English language that do not sound like the way they are written. The method used to accomplish this task is neural networks.

Neural networks have been used to solve many types of problems over the years. These problems include pattern recognition, data classification, and other very complex problems. The advantage of neural networks is in their parallel architecture. The neural network simulates how the brain uses interconnected neurons to process information. With neural networks, the nodes of the networks are interconnected and work in unison to solve specific problems. However, the only way the neuron network learns is by

example. Therefore, the network must guess an output, compute an error using the actual output, and then correct itself by adjusting its neurons [38].

Converting English text to English speak is an example of a complex problem that neuron networks can solve. English text can be very tricky to pronounce. For example, words like “Philadelphia”, “bike”, and “brought” are not pronounced how they are spelled. Using data to correct itself, a network can learn to pronounce English text. For this system, a neural network is trained with the Back Propagation algorithm using a data set of about 1800 words [38]. The network was trained on a Dell Optiplex 2.80 GHz Intel Pentium 4 computer.

4.2 Text-to-Phoneme Conversion Algorithm

The TTP algorithm used by this system follows very closely to the algorithm used by Sejonowski and Rosenberg. Sejonowski and Rosenberg created the NETTalk system in the late 1980’s. As previously discussed in Section 3.2, this system used the Back Propagation algorithm to train a network to pronounce inputted text. The original NETTalk used a three layered network with 120 hidden neurons and 26 output neurons that represented each phoneme. Figure 4-1 shows the outline of the NETTalk program. The training data used in the NETTalk system consisted of 2000 of the most common words aligned with their pronunciations [35]. Although this algorithm is similar to the original NETTalk algorithm, there are some slight modifications to the network, training data, and training algorithm.

The overall network should be thought of as a loop. An input is taken from the training data, an output is calculated, and the nodes of the network are adjusted according to the error. Then another input vector follows the same steps, and the process continues until some stopping criterion is met.

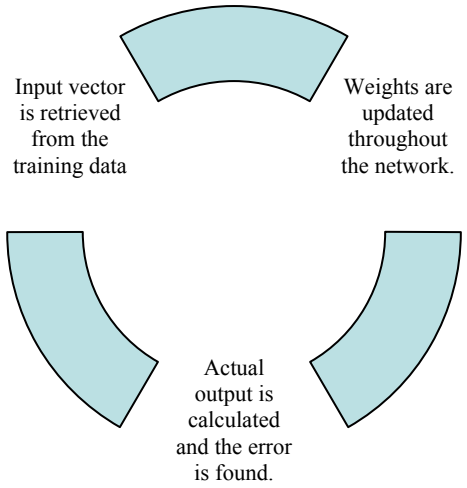


Figure 4-1: Network Training Cycle

4.2.1 Network Input and Output Design

The network consists of seven windows for the input and one output for the system. The input to the network is not the total word but single characters. Each word from the training data is entered into the window one character at a time. The inputted training word enters the system and moves through the seven windows one character at a time. The network's input is the seven windows. Figure 4-2 shows an example of how this process takes places.

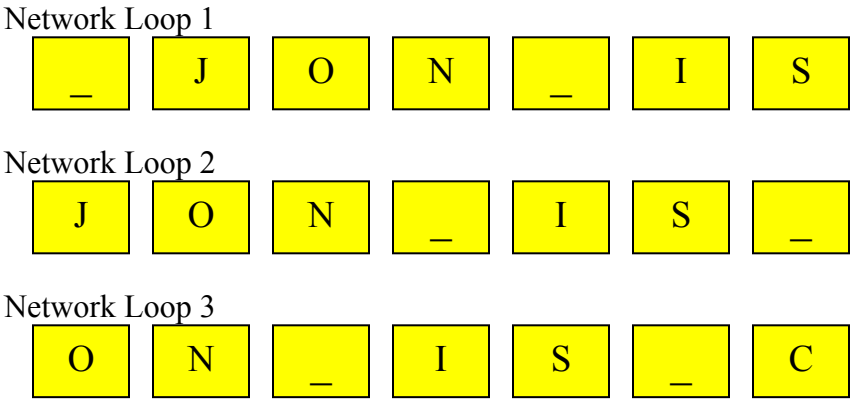


Figure 4-2: Moving Window Example

The output of the network is the phonetic representation of the 4th input window at any given time. The input and output must be aligned to ensure that the output represents the correct input window. The output can also be thought of as seven windows aligned with the seven windows of the input. However, the only output window of concern is the 4th window. As the input letters move through the seven input windows, the output phonemes move through the seven output windows. During training, the network inputs are all seven letters in the input windows, and the network desired output is the phoneme in the fourth position of the output windows. Figure 4-3 shows an example of how the output windows work.

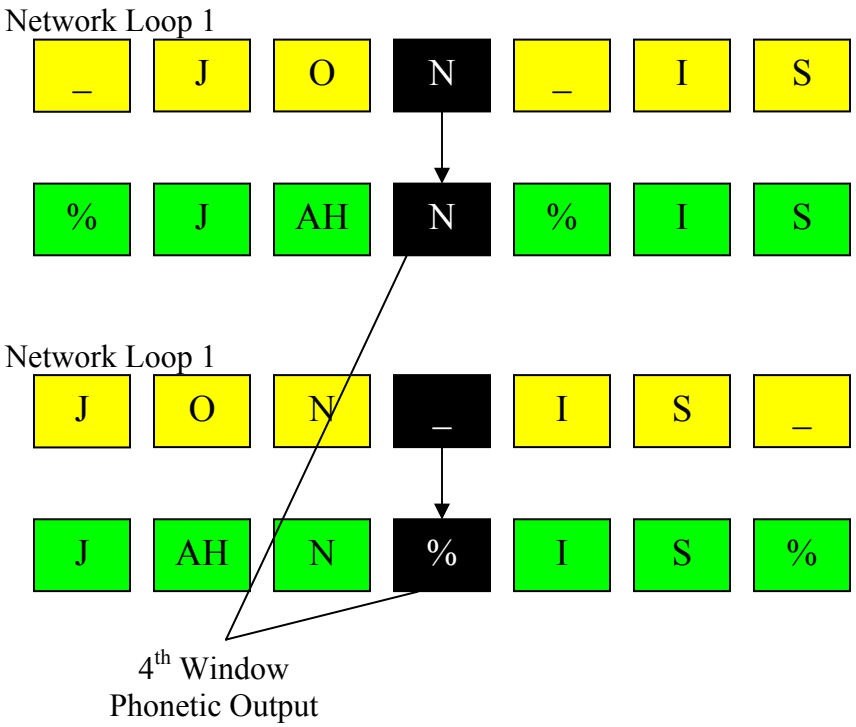


Figure 4-3: Moving Window Example

4.2.2 Network Detailed Training Design

The training of the network involves obtaining a word and its phonetic representation from the training set, placing the word's first character in the first input window, calculating the actual output of the seven windows, and updating the weights. The network then moves the characters throughout the windows until the word ends. If

training is not done, the network retrieves the next word from the training set. Figure 4-4 shows an outline of the training algorithm.

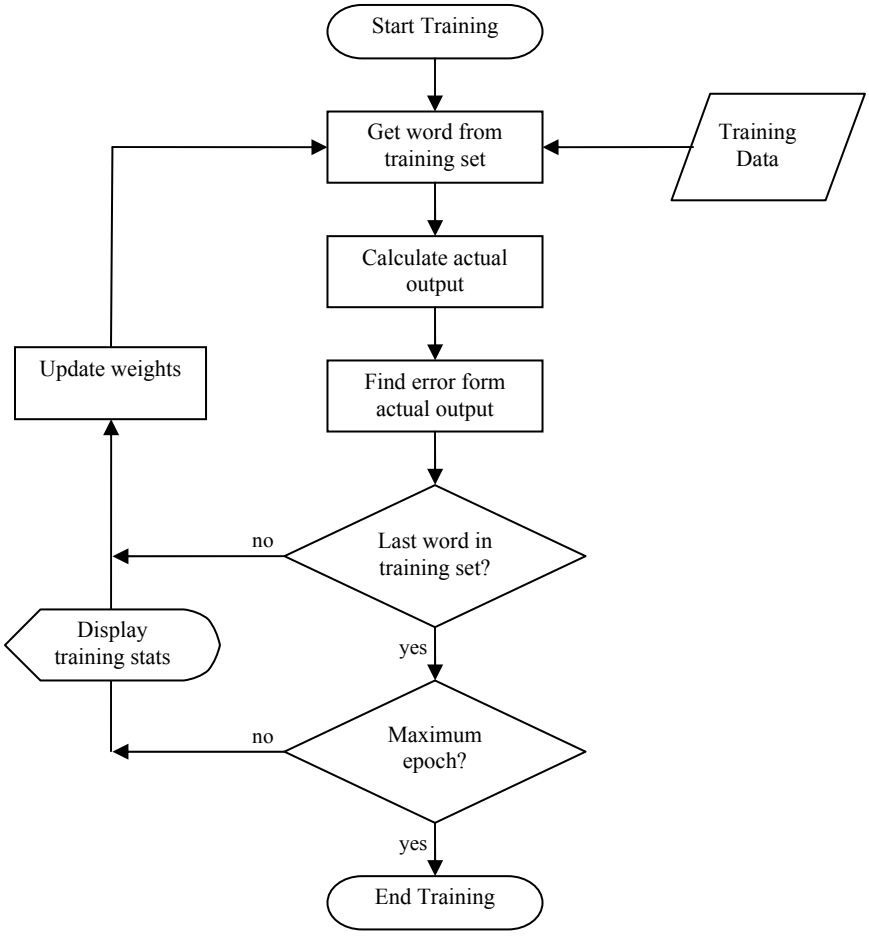


Figure 4-4: Network Training Flow Chart

The first step of the network is retrieving the input vector from the training data. However, the network does not start with the first word of the data when training first begins. Initially, the training network input is set to all silences with the desired output being a silence. This was performed to simplify the beginning of the training process and to insure that the network and the output are aligned properly. Figure 4-5 shows the initial look of the network at the beginning of training.

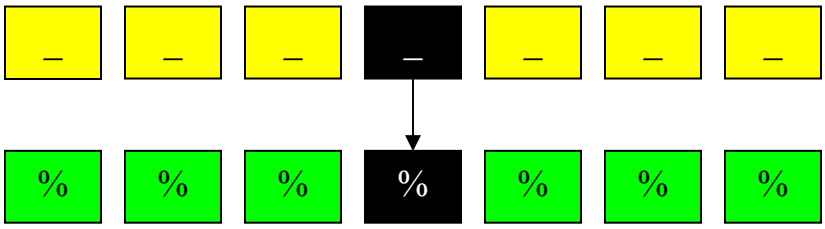


Figure 4-5: Initial State of the Network

After the first iteration, the first word is retrieved from the training set. The first letter of the word is placed in the seven letter window. If the first word is "aardvark", then the seven windows would have six silences and one letter after the first iteration. The desired output would still be a silence. Figure 4-6 shows the seven input windows with the desired output.

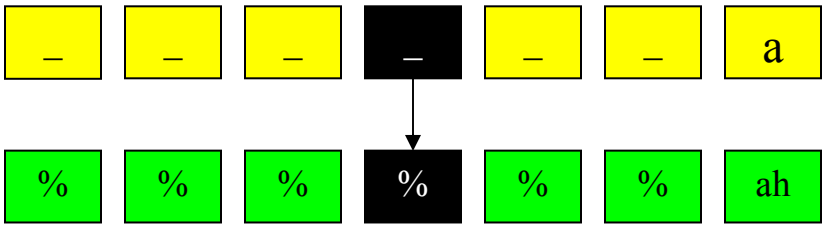


Figure 4-6: State of the Network after First Iteration

In order to calculate the actual output of the system, the letters and phonemes need to be converted to numbers. In this program, each input letter is converted into a vector. The vector size is the total number of characters used in the network. For the network, there are 27 characters used for the input, 26 characters that represent the alphabet and one character that represents a silence. The input vector consists of zeros and a one. Each input vector is all zeros, with a one located to indicate the letter. For example, the letter *d* would be a vector of zeros with a one located at the 4^{th} position of the vector.

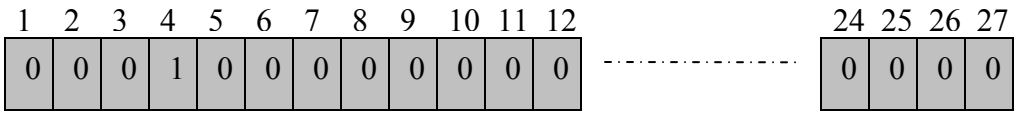


Figure 4-7: Input Vector for the Letter d

The output vector is very similar to the input vector. The vector size is the total number of different phonemes used in the network. There are 41 phonemes used, 40 for sound and one for silence. The ideal output consists of all zeros and a one; therefore, the network is trained using a desired output vector with all zeros and a one. Each phoneme has its own location in the output vector. Like the input vector, the one is located where the phoneme is located in the vector. For example, a desired output phoneme */ah/* would produce a desired output vector of all zeros and a one placed in the 2nd position of the vector.

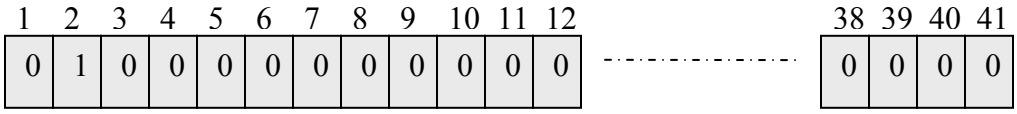


Figure 4-8: Desired Output Vector /ah/

Once the letters are converted to vectors, the actual output can be calculated. During training, the output vector is computed using all seven windows. Therefore, the input used for calculation is a 7*27 long vector with seven ones located throughout the vector. All the weights of the network, i.e. hidden weights and output weights, are initially set to random values between -.5 to .5. Using the equations from Section 3.5.1, the actual output is calculated.

$$\textit{hidden layer vector} = 1/(1 + e^{-\Sigma(\text{seven window input vector} * \text{hidden neurons weights})}) \quad (4.10)$$

$$\textit{actual output vector} = 1/(1 + e^{-\Sigma(\text{hidden layer vector} * \text{output neurons weights})}) \quad (4.11)$$

These equations will produce an output vector output of the network. The actual output and the error of the actual output are calculated. Weight adjustment occurs after each

loop, i.e. after every single letter of every word in the training set. The weights of the system are changed using the backward propagation equations described in Section 3.5.2.

$$\textit{output layer weights} = W_{\text{old output}} + \alpha\,\eta\delta_{(k+1)m}\ \text{hidden layer ouput} \quad (4.12)$$

$$\textit{hidden layer weights} = W_{\text{old hidden}} + \eta\delta_{km}\text{input vector}_m \quad (4.13)$$

Once the weights are adjusted, the network will retrieve the next letter and desired output of the word. If there are no letters left in the word, the network will retrieve the next word from the training set. Once the last letter of the last word is retrieved for the training set, an epoch has been completed and the average error is calculated and displayed. The next word entered into the network would then be the first word of the training set. Training will continue in this manner until the maximum number of epochs is reached. Once the training is complete, the weights of the network are saved, and the weight can be used for TTP mapping.

4.2.3 Training Set Detail and Alignment

Training the network involves using real words aligned with their exact pronunciation phonemes. For example, ‘cat’ or ‘example’ would be aligned with ‘kat’ and ‘eksampul’, respectively. The list of the 41 phonemes used for this system is in Appendix A. Originally, about 900 words were used from the “Cue Practice With the 1000 Most Common Words” website as training data. However, this amount of data was not sufficient enough to yield an acceptable performance. The rest of the data was chosen based on the network performance using www.allwords.com. For example, the network initially struggled with words beginning with “th”, so about 20 words beginning and ending with “th” were added to the training data. Appendix C lists the entire training set.

Certain factors need to be accounted for when choosing the training data. The network input size consists of only seven windows. However, since the words are entered into the training window one letter at a time, the word size can be any number of characters. In addition, all words, even the words initially found on the web, need to be aligned by hand in order to maximize the performance of the system. Figure 4-9 shows an example of how this alignment is done.

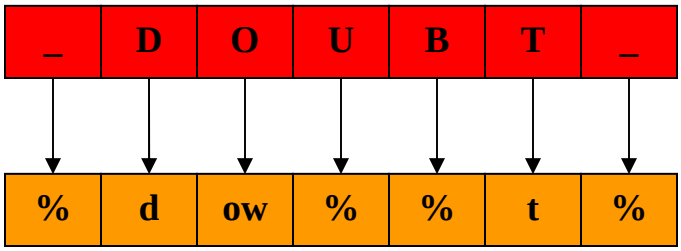


Figure 4-9: Word Alignment

Chapter 5: Automatic Prosody Generation

5.1 Generation of Prosody Overview

Prosody deals with the tone and the timing of speech. In speech synthesis systems, the most complex problem is the generation of natural sounding prosody. Real speech often reflects the speaker's personal knowledge of the audience, the mood and emotional state, the general knowledge of the world around them, and the reaction from the audience. With TTS systems, this problem is more complex because the text input is random. These issues have yet to be resolved by speech synthesis systems. Automatic prosody generation has to factor in these setbacks and focus on neural language that is understandable to the audience.

The main focus of this thesis is the automatic generation of prosody. This system generates two main aspects of prosody: fundamental frequency and segmental duration. Fundamental frequency is generated using fuzzy logic. The primary focus of this chapter will be on this aspect of prosody. Segmental duration is determined using rules described by Klatt [2]. After the network converts the text to phonemes, the system then computes the prosody of the system. Both the phonemes found by the network and the actual texts are used to generate prosody. The prosody generator can be thought of as a finite loop, with information describing each phoneme used to produce both duration and fundamental frequency. The first step of prosody generation is segmental duration.

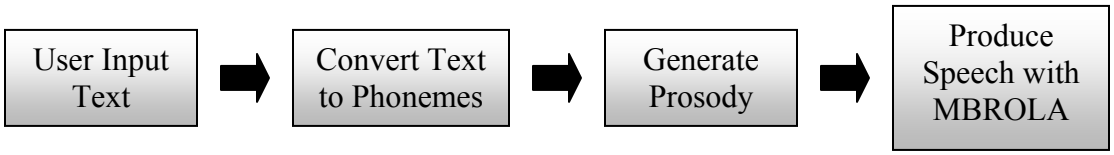


Figure 5-1: System Overview

5.2 Segmental Duration

Segmental duration handles the length of segments within words. These segments can be either syllables or phonemes. For this system, the focus will be on phoneme duration. Researchers believe that English phoneme durations are only affected by a few parameters. In 1979, Dennis Klatt developed rules that governed the durations of phonemes within words. These rules were used in the aforementioned MITTalk TTS system [2]. The *Klatt Duration Rules* are used to generate the segmental duration of this system.

The original MITalk system was a formant synthesized speech. With formant synthesizers, certain factors need to be taken into account. Phonemes with different articulations are handled differently when it comes to duration. For example, the duration of a fricative is measured by the visible noise. For stops, the duration also includes the closure [2]. With this TTS system, the speech synthesis is generated using diphone concatenation. Therefore, these factors are not factored into consideration when computing the segmental duration.

The Klatt Duration rules were designed to replicate observed duration from a speaker. There are two main rules that govern the model: segments that are altered by a percentage using the rules and segments that cannot be shorter than a minimum duration. Each segment has its own minimum and average duration. The minimum and average durations for each phoneme used in this system are shown in Appendix B. The formula used to alter the phoneme duration is:

$$DURATION = ((AVEDUR-MINDUR)*(PRCNT))/100 + MINDUR \quad (5.1)$$

The variable *AVEDUR* is the average duration, and variable *MINDUR* is the minimum duration. The Klatt rules work by altering the *PRCNT* variable. There are a total of 11 Klatt Duration rules. Each rule has a justification on how the rule was created. The rule formulas were developed through extensive trial and error. The Klatt duration rules are listed below. These rules are obtained directly from the book, “From Text to Speech: The MITalk System” [2].

1. Pause insertion Rule
   Insert a 200 millisecond pause before each sentence-internal main clause and at boundaries delimited by a syntactic comma, but not before relative clauses.
2. Clause-final lengthening
   The vowel or syllabic consonant in the syllable before a pause is length by PRCNT = 140.
3. Non-phrase-final shortening
   Vowels and syllabic consonants are shortened by PRCNT = 60, if not in the phrase's last syllable. A phrase final postvocalic liquid or nasal is lengthened by PRCNT = 140.
4. Non-word-final shortening
   Vowels and syllabic consonants are shortened by PRCNT =85, if not in the word's last syllable.
5. Polysyllabic shortening
   Vowels and syllabic consonants in word with multiple syllables are shortened by PRCNT = 80.
6. Non-initial-consonant shortening
   Consonants are shortened by PRCNT = 85, if not in the word's initial position.
7. Non-phrase-final shortening
   Vowels and syllabic consonants are shortened by PRCNT = 60, if not in the phrase's last syllable. A phrase final postvocalic liquid or nasal is lengthened by PRCNT = 140.
8. Lengthening for emphasis
   Emphasized vowels are lengthened by PRCNT = 140.
9. Postvocalic context of vowels
   The consonant after a vowel in the same word influences the length of the vowel. The list below shows these effects.
   - open syllable, final word PRCNT1 = 120
   - before a voiced fricative PRCNT1 = 160
   - before a voiced plosive PRCNT1 = 120
   - before a nasal PRCNT1 = 85
   - before a voiced plosive PRCNT1 = 70
10. Shortening Clusters
    Segments are shortened in consonant-consonant sequences and in vowel-vowel sequences.
    - vowel followed by a vowel PRCNT1 = 120
    - vowel proceeding a vowel PRCNT1 = 70
    - consonant surrounded by consonants PRCNT1 = 50
    - consonant proceeding a consonant PRCNT1 = 70
    - consonant followed by a consonant PRCNT1 = 70
11. Lengthening due to plosives aspirations
    A primary or secondary stressed vowel or sonorant preceded by a voiceless plosive is lengthened by 25 milliseconds.

5.3 Stress Assignment

One of the most crucial components of generating automatic prosody is stress assignment. Stress assignment is placing lexical stresses on the proper syllable. All stresses are placed on vowels. With TTS synthesis this can be very complicated. Since the input is random, the words are random. Therefore, a stress assignment algorithm needs to be implemented that is both automatic and flexible. The stress assignment algorithm used for this system is the rule-based system used in the MITalk. Those rules were based on the Halle and Keyer lexical stress rules developed in 1971 [2].

The rules are based upon phonetic input without regard to part-of-speech. Stresses are placed on individual words independently. There are three levels of stress; 0-stress, 1-stress, and 2-stress. The 1-stress represents the primary stress in a word and the 2-stress represents lesser stresses. 0-stress represents no stress. The stress rules have two different phases. The first phase is called the cyclic and is committed to placing primary stresses on the word. There are three rules in the cyclic phase. The first rule in the cyclic phase is the main stress rule, and the other two rules are exceptions to the first rule. The second phase is called the non-cyclic phase and includes the application of the entire word of rules. The non-cyclic phase reduces the final word to just one 1-stress mark and turns the rest of the primary marks into 2-stress marks [2].

5.4 Fuzzy Fundamental Frequency

Fundamental frequency is a vital part of the naturalness of any TTS system. This frequency, sometimes called f0 frequency, produces the tone of speech. An example of fundamental frequency can be heard in the difference between a male and a female voice. Male voices exude a lower overall fundamental frequency while female voices typically have a higher overall fundamental frequency. Fundamental frequency can be generated in many ways for TTS systems. Most systems today generate fundamental frequency using unit selection synthesis together with pre-recorded units (sounds). These pre-recorded sounds contain natural frequency fundamentals. Other methods used to generate frequency fundamentals are rule-based and neural network approaches.

The fundamental frequency of this TTS system is generated using the flexibility of fuzzy logic. Fuzzy logic can be used to control complicated problems. The generation of fundamental frequency for TTS systems is a very complex problem for many reasons. For example, the user's input needs to be thought of as random; therefore, the system must be flexible. Also, the fundamental frequency in speech is different for every sentence, and many factors control this overall curve. Fuzzy logic can be a solution to the fundamental frequency problem by using a small set of rules. The rules are constructed based upon the fundamental frequency algorithm used in the MITalk system, the O'Shaughnessy algorithm.

5.5 O'Shaughnessy Algorithm

The O'Shaughnessy fundamental frequency algorithm was used in the MITalk system to generate fundamental frequency. The result of the algorithm is a fundamental frequency curve throughout the sentence. This curve is called the f0 contour. This algorithm is detailed in the book, "From Text to Speech: The MITalk System" [2]. The algorithm has two main levels, high and low, which work as two different systems. The first level is the high level. This level generates an outline of how the overall f0 contour should be shaped. The high level of this algorithm takes grammatical information about the sentence and builds a basic contour of the system. The f0 contour in the high level is augmented using four factors: sentence type, phrase contour, word contour, and prosodic indicators [2].

The first factor, sentence type, is determined by the first word and punctuation of the sentence. For example, the endings of questions are indicated by question marks and the endings of statements are indicated by commas or periods. There are three tunes that represent three types of sentences: declarative sentences, yes/no question and "wh" questions. These three tunes determine the f0 contour trend of the sentence. Every tune has a downward linear trend throughout the sentence [2]. Figure 5-2 shows this trend.

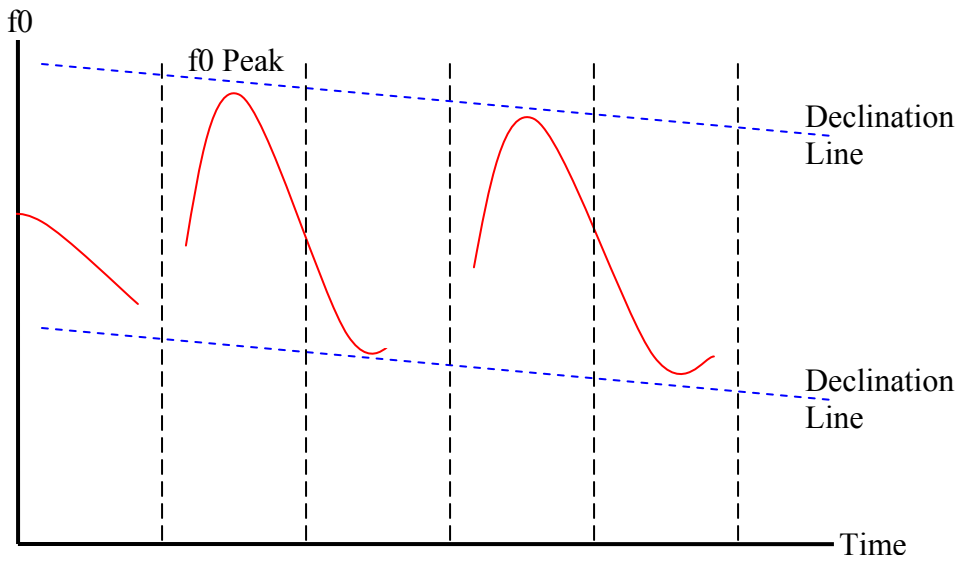


Figure 5-2: Downward Linear Trend

Tune A, which deals with declarative sentences, causes the f0 contour to fall linearly throughout the sentence with the sharp decline at the end of the sentence or phrase. Tune B manages yes/no questions. Tune B sentences have an initial rise to the f0 contour, followed by a flat contour with a final sharp rise. The final tune, which handles "wh" questions, starts with a high peak on the "wh" word, followed by a steep fall and a high rise at the end of the sentence [2].

The second factor of the algorithm contends with the phrase contour of the sentence. The different phrases of a sentence can be determined with many different algorithms. Each phrase must have two or more content words to be affected by this factor. At the beginning of each phrase, the f0 curve rises sharply on the first content word. At the end of each phrase, the f0 curve falls on the final content word.

The third factor handles the individual words in a sentence. Within each word, the f0 contour fluctuates. This third factor determines the amount of change within each word. The amount of change is correlated with the importance of the word and the amount of syllables in the word [2]. Words that are more important and have a great deal of syllables contain the most f0 changes. Table 5-1 shows the importance of these words.

Table 5-1: Word Importance

Level	Part of Speech
0	article
1	conjunction, relative pronoun
2	preposition, auxiliary verb
3	personal pronoun
6	verb, demonstrative pronoun
7	noun, adjective, adverb, contraction
8	reflexive pronoun
9	stressable modal
10	quantifier
11	interrogative adjective
12	negative element
14	sentential adverb

The prosodic indicator assignment is the last factor of the high level part of the algorithm. First, accent numbers are given to each accent in the sentence. The number depends on the size and importance of the word. Then an integer that represents the word position is placed on each word. Words at phrase boundary positions are given larger integer values than words that are in the middle of the phrase. The high level part of the algorithm combines all of these factors to form the outline of the sentence f0 contour. The high level then becomes the input to the low level component of the algorithm [2].

The low level of the O'Shaughnessy algorithm sculpts the details of the final f0 contour. In the low level, the f0 contour is affected by the number of syllables in combination with the lexical stress. The importance of the word affects the height of the stressed peak. More important words have higher peaks on their stresses. The amount of syllables directly influences the fundamental frequency of the sentence. The first and highest peak should have an f0 contour of about 190 Hz. However, larger sentences with more syllables have a higher starting peak frequency, and smaller sentences with fewer syllables have a lower starting peak frequency. The number of syllables between stressed syllables also affects the height and outer shape of a peak. Stressed peaks separated by two or three syllables have their peak height decreased by 15% and 20%. Stressed peaks separated by two, three, or four syllables have their peak rise increased by 15%, 20%, and 30%, respectively. Stressed peaks that are adjacent have their peak rises decreased by 40%. All other peaks maintain their heights and are shaped by a default rise [2].

5.6 Fuzzy System Inputs and Consequence

The fuzzy system is developed to model the O'Shaughnessy algorithm using a fuzzy interference system. The inputs to the fuzzy system are the phonemes' linguistic data. The output of the fuzzy system is the fundamental frequency. The system consists of four inputs and one output. The inputs are: word importance, sentence size, position in sentence, and distance from stress. Each input has three linguistic variables and the output has seven linguistic variables. Using a system similar to the system described in 4.3.2, the fundamental frequency can be computed for any given sentence.

5.6.1 Word Importance

The inputs to the fuzzy system are formulated from the main parts of the O'Shaughnessy algorithm. The first input is the importance of the word. This input comes from the word importance of the O'Shaughnessy algorithm. The O'Shaughnessy algorithm places importance on words based on part-of-speech. There are two types of words this system takes into consideration, function words and content words. Content words have more importance than function words, with articles having the least importance. The system calculates the importance of the word based on the size of the word and the type of word. The calculation yields a number between 0 and 10; therefore, the fuzzy input consists of a number range between 0 and 10. The *Word Importance* input has three linguistic variables for this input: useless, semi-important, and important. Figure 5-3 shows the membership functions of the *Word Importance* input.

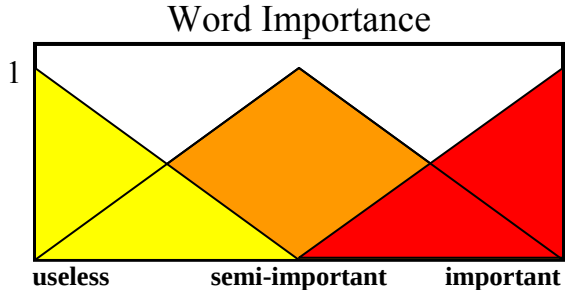


Figure 5-3: Word Importance Input Membership Functions

5.6.2 Sentence Size

The second input to the fuzzy system is the sentence size. In the O'Shaughnessy algorithm, peak size is correlated with the length of the sentence. According to the algorithm, smaller sentences, i.e., sentences with fewer syllables have smaller fundamental frequency peaks. Larger sentences have much larger fundamental frequency peaks. The input is divided into three linguistic variables: small, medium, and large. The sentence size is determined by the number of syllables multiplied by an offset.

*Size = (number of phonemes) * offset* (5.2)
offset = .5

The input range values are from 0 to 10. Any size that is greater than 10 is considered to be equal to 10 by the system.

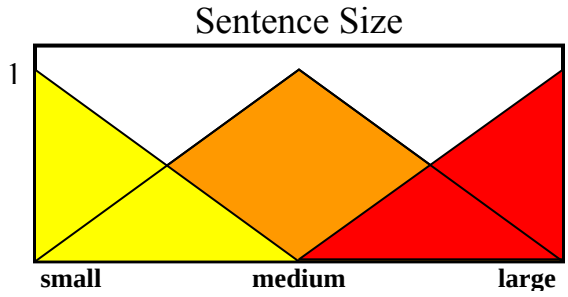


Figure 5-4: Sentence Size Input Membership Functions

5.6.3 Sentence Position

The third input is the position in the sentence. The position in the sentence deals with the word, not the phoneme. Sentence position is calculated using the following equation:

*Position = (word location in sentence / total number of words)*10* (5.3)

The location of the word in the sentence is a very important aspect of the O'Shaughnessy algorithm. As described in Section 5.5, each tune's fundamental frequency contour differs throughout the sentence. Therefore, the location of the phoneme or word determines the frequency contour depending on the type of tune. For example with tune B sentences (yes/no questions), at the end of the sentence, the fundamental frequency increases sharply. Like the previous inputs to the fuzzy system, the *Sentence Position* input has three linguistic variables and a number range between 0 and 10. The input variables are start, middle, and end. Figure 5-5 shows the membership functions for the *Sentence Position* input.

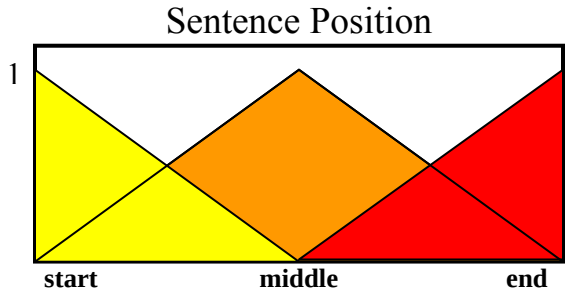


Figure 5-5: Sentence Position Input Membership Functions

5.6.4 Stress Distance

The final input to the fuzzy system is the phoneme distance from the stress. The O'Shaughnessy algorithm places peaks on primary stresses in sentences. The fundamental frequency of phonemes around these stresses is also altered. Peak size depends on the location, sentence type, and importance of the word. Stress distance is calculated using the closest, right-hand, most stress within the word. If a phoneme is stressed, its distance value is "dead-on". Stressed phonemes equal the maximum. Phonemes inside of a word without a stress are assigned a zero. The fuzzy input ranges from 0 to 10. If the phoneme is not stressed and the word contains a stress, the distance is calculated using the formulas:

$$stress\ distance = ((stress\ location - phoneme\ location)+1)*offset \quad (5.4)$$

$$offset = 10/(length\ of\ the\ word) \quad (5.5)$$

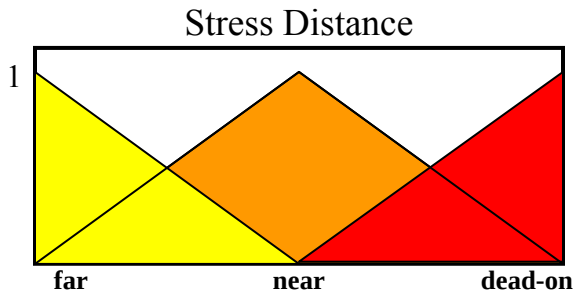


Figure 5-6: Stress Distance Input Membership Functions

5.6.5 Consequence

The consequence (output) of the system is the fundamental frequency. The consequence of the fuzzy system has seven linguistic variables. These variables represent the peak of the fundamental frequency. These peaks range from negligible to large. These rule consequences are constructed following the O'Shaughnessy algorithm's peak alterations.

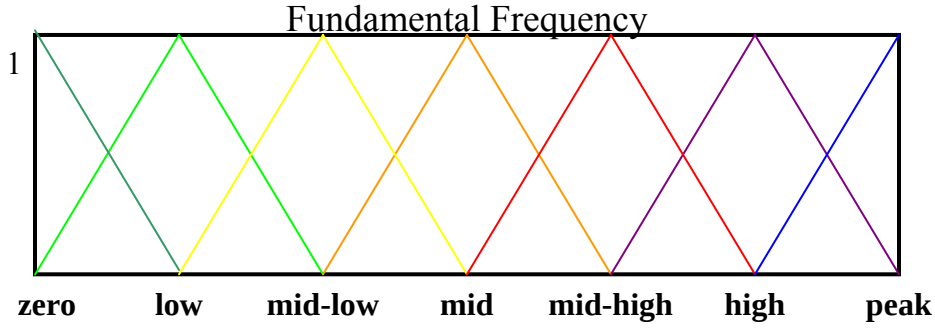


Figure 5-7: Consequence Membership Functions

5.7 Fuzzy System Rules

The rules of this system are designed to follow the overall idea of the O'Shaughnessy algorithm. The O'Shaughnessy algorithm is very specific on the shapes of each individual peak. However, these rules create an overall contour of the sentence,

which follows the O'Shaughnessy algorithm. The rules are divided into three categories: declarative sentences, yes/no questions, and interrogative questions. To simplify each rule set, the rules are divided by location within the sentence. This made it easy to edit, delete, or add rules. Since there are four inputs to the system, there are over 50 rules. This adds the proper detail and improves the flexibility of the fuzzy system.

5.7.1 Declarative Sentence

As described in the O'Shaughnessy algorithm, declarative sentences are characterized by the f0 contour falling linearly with a sharp decline at the end of the sentence or phrase. Peaks within the sentence are altered, considering their location in the sentence. Declarative sentences do not have dramatic f0 contour changes inside the middle of the sentence. Therefore, syllables and word importance should produce subtle changes. With declarative sentences, the final fall should be located after the last accented syllable [2].

Beginning of Sentence Rules

- IF the word is useless AND position in sentence at the beginning THEN output is low.
- IF the word is semi-important AND position in sentence at the beginning THEN output is mid-low.
- IF the word is semi-important AND sentence size is large AND position in sentence at the beginning THEN output is mid.
- IF sentence size is small AND position in sentence at the beginning AND the stress is dead-on THEN output is high.
- IF position in sentence at the beginning AND the stress is dead-on THEN output is peak.
- IF sentence size is small AND position in sentence at the beginning THEN output is mid.

Middle of Sentence Rules

- IF the word is useless AND position in sentence at the middle THEN output is low.
- IF the word is useless AND sentence size is large AND position in sentence at the middle THEN output is mid-low.
- IF the word is semi-important AND position in sentence at the middle THEN output is mid-low.
- IF the word is semi-important AND sentence size is large AND position in sentence at the middle AND the stress is dead-on THEN output is mid.

- IF position in sentence at the middle AND the stress is dead-on THEN output is mid-high.
- IF sentence size is medium AND position in sentence at the middle THEN output is mid.

End of Sentence Rules

- IF sentence is at the end AND the stress is far THEN output is mid-low.
- IF sentence is at the end AND the stress is near THEN output is zero.
- IF sentence size is middle AND sentence at the end THEN output is mid-low.
- IF word is useless AND sentence is at the end THEN output is zero.

5.7.2 Yes/no Question

Yes/no questions are questions that can be answered with either a "yes" or a "no". These questions do not begin with "who", "what", "where", "when", "why", or "how". The overall f0 contour of these questions begins with a rise, followed by a flat contour, and then ending with a final sharp rise [2].

Beginning of Sentence Rules

- IF the word is useless AND position in sentence at the beginning THEN output is mid-low.
- IF the word is semi-important AND position in sentence at the beginning THEN output is mid.
- IF the word is semi-important AND sentence size is large AND position in sentence at the beginning THEN output is mid-low.
- IF position in sentence at the beginning AND the stress is near THEN output is mid-high.
- IF position in sentence at the beginning AND the stress is dead-on THEN output is mid-high.

Middle of Sentence Rules

- IF the word is useless AND position in sentence at the middle THEN output is mid-low.
- IF the word is useless AND sentence size is large AND position in sentence at the middle THEN output is low.
- IF the word is semi-important AND position in sentence at the middle THEN output is mid-low.
- IF the word is important AND position in sentence at the middle AND the stress is dead-on THEN output is mid-high.
- IF position in sentence at the middle AND the stress is dead-on THEN output is mid.

- IF sentence size is medium AND position in sentence at the middle THEN output is mid.
- IF sentence size is large AND position in sentence at the middle THEN output is mid-low.

End of Sentence Rules

- IF sentence is at the end AND the stress is far THEN output is high.
- IF sentence is at the end AND the stress is near THEN output is mid-high.
- IF sentence size is large AND sentence at the end AND the stress is dead-on THEN output is high.
- IF word is useless AND sentence is at the end AND the stress is near THEN output is mid-high.
- IF word is important AND sentence is at the end AND the stress is dead-on THEN output is high.

5.7.3 Interrogative Question

Interrogative questions are questions that are answered with more than just a "yes" or "no". These sentences begin with either "who", "what", "where", "when", "why", or "how". Overall, the f0 contour should initially be high with the final steep fall. The fall of the interrogative question should be much steeper than the fall of the declarative sentence. According to the O'Shaughnessy algorithm, the peaks throughout the sentence should be much higher and the overall contour should be higher than the declarative sentence contour. Syllables and word importance should produce change near accents [2].

Beginning of Sentence Rules

- IF position in sentence at the beginning THEN output is mid-low.
- IF the word is important AND position in sentence at the beginning THEN output is mid.
- IF the word is semi-important AND sentence size is large AND position in sentence at the beginning THEN output is mid.
- IF sentence size is small AND position in sentence at the beginning AND the stress is dead-on THEN output is mid.
- IF position in sentence at the beginning AND the stress is dead-on THEN output is mid-high.
- IF sentence size is small AND position in sentence at the beginning THEN output is mid-low.

Middle of Sentence Rules

- IF the word is useless AND position in sentence at the middle THEN output is mid.
- IF the word is useless AND sentence size is large AND position in sentence at the middle THEN output is mid-low.
- IF the word is semi-important AND sentence size is large AND position in sentence at the middle AND the stress is dead-on THEN output is mid-high.
- IF position in sentence at the middle AND the stress is dead-on THEN output is mid-high.
- IF sentence size is medium AND position in sentence at the middle THEN output is mid.
- IF the word is important AND position in sentence at the middle AND the stress is near THEN output is mid-high.
- IF sentence size is medium AND position in sentence at the middle AND the stress is near THEN output is high.
- IF sentence size is medium AND position in sentence at the middle THEN output is mid-low.

End of Sentence Rules

- IF sentence is at the end AND the stress is dead-on THEN output is mid-low.
- IF sentence is at the end AND the stress is near THEN output is zero.
- IF sentence size is medium AND sentence at the end THEN output is low.
- IF word is useless AND sentence is at the end THEN output is zero.

5.8 Fuzzy Output Calculation

In a fuzzy system, the output is calculated using the rules, the membership functions, and the inputs. Each input parameter value is calculated using the formulas in Section 5.6. The firing strength of each input is determined by the membership functions. Figure 5-8 shows an example of the *Word Importance* input with a calculated input of 6.

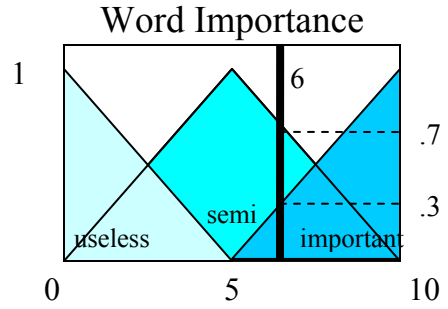


Figure 5-8: Example of Word Importance with Calculated Input of 6

The firing strength of each membership function is shown in Figure 5-8. The membership function "semi" and "important" have a firing strength of .7 and .3, respectively. The "useless" membership function has a firing strength of 0 because the input value does not touch the membership function. The rules of the fuzzy system are used to compute the consequence of the inputs presented to the system. For example, if a rule stated," IF word important is semi-important AND … ", then for an input value of 6 on input *Word Importance,* the firing strength onto that rule would be .7. The operators then determine the effects of the input onto the consequence membership function. The *AND* operates equals the minimum value and the OR operator equals the maximum value. Figure 5-9 shows an example of how an input to the fuzzy system determines the consequence. For simplicity, there are only two rules and the consequence only has three membership functions. The rules in Figure 5-9 are:

- IF word is importance AND sentence at the end THEN fundamental frequency is high.
- IF word is semi-importance AND sentence at the middle THEN fundamental frequency is middle.

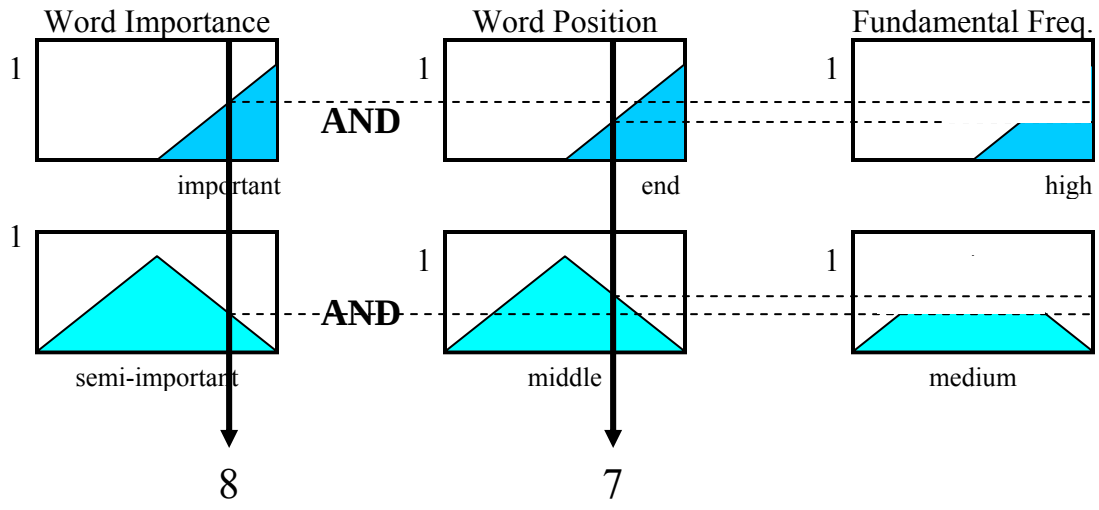


Figure 5-9: Firing Strength of Input onto Rules

The consequences are then combined to produce a final shape. The crisp output of the fuzzy system is calculated by finding the centroid of the final shape.

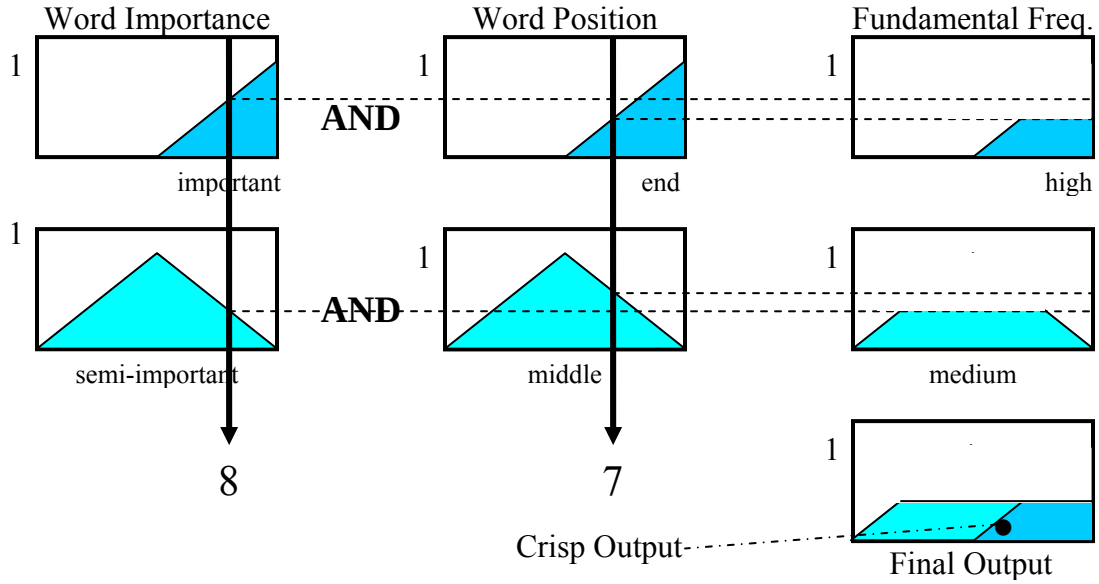


Figure 5-10: Final Shape and Centroid

5.9 Final Speech Production

Once the network is trained and the prosody is calculated, the MBROLA program is used to produce the final speech output. As discussed briefly in Section 3.3, the MBROLA program creates speech using diphone concatenation. This program was developed by the TCTS Lab of the Faculté Polytechnique de Mons in Belgium and was created for use by researchers [15]. The advantage of using the MBROLA program is its ease of use. The only input needed to produce speech is phonemes and durations. Users of MBROLA do not need any knowledge of diphone concatenation algorithms; the program automatically converts the user phonemes to diphones.

The trained hidden and output weights are stored in a MATLAB *.mat file. The TTP conversion is accomplished using the trained weights and formulas. The output is computed using the formulas used to calculate the actual output during training as described in Section 4.2.1.

$$\textit{hidden layer outputs} = 1/(1 + e^{-\Sigma(\text{input vector} * \text{hidden neurons weights})}) \quad (5.6)$$

$$\textit{actual output} = 1/(1 + e^{-\Sigma(\text{hidden layer outputs} * \text{output neurons weights})}) \quad (5.7)$$

The actual output of the system will be a 41 element long vector. The ideal output would be all zeros and a one; however, since the network is not ideal, the maximum of the output needs to be determined. This maximum value of the vector is the output phoneme produced by the network from the text.

When text is entered into the system, the fuzzy controller and the Klatt duration rules compute the prosody of the system. Next, the symbols are converted to the MBROLA symbols. The phonemes, duration, and fundamental frequency values are written to *output.pho* file. *Pho* files are the files used by MBROLA to convert phonemes to sounds. The *output.pho* file is opened and the speech is converted into sound file. See Appendix for more detail on the MBROLA system.

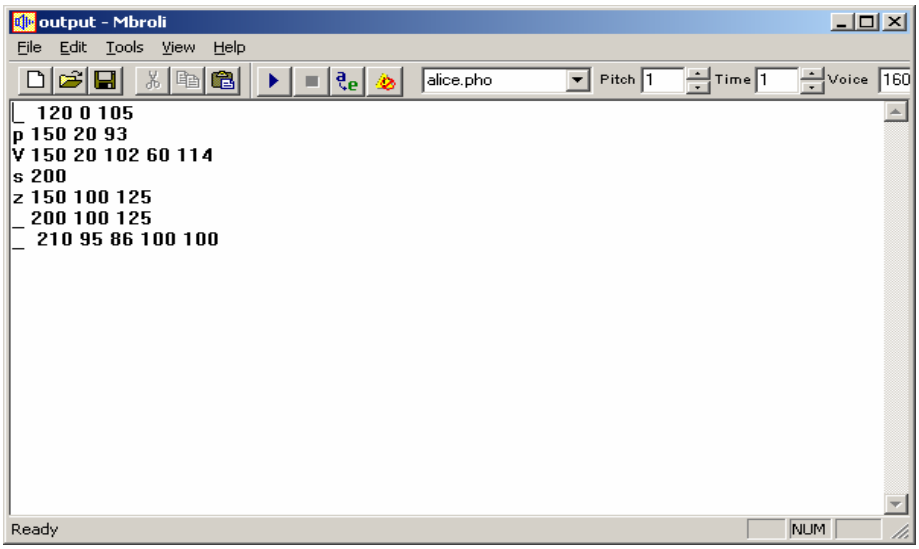


Figure 5-11: MBROLA *.pho file

Chapter 6: Results

6.1 Text-to-Phoneme Results

Neural networks are used in TTS applications in many different ways. Current research focuses on the use of neural networks to produce prosody. This thesis uses neural networks for TTP mapping. Current neural network TTP mapping techniques yield about an 80% accuracy [7]. The performance of the neural network algorithm in this system is determined by the amount of accurate phonemes the network identifies. Two types of tests are performed to determine the performance of the algorithm used. The first test calculates the performance using the full training set. The second test uses words that were not in the training set. Performance was measured by the percentage of correct phonetic conversions. Figure 6-1, shows the convergence of the networks. The number of hidden neurons is correlated to the network's convergence. However, after 50 hidden neurons, the network convergance doesn't change much.

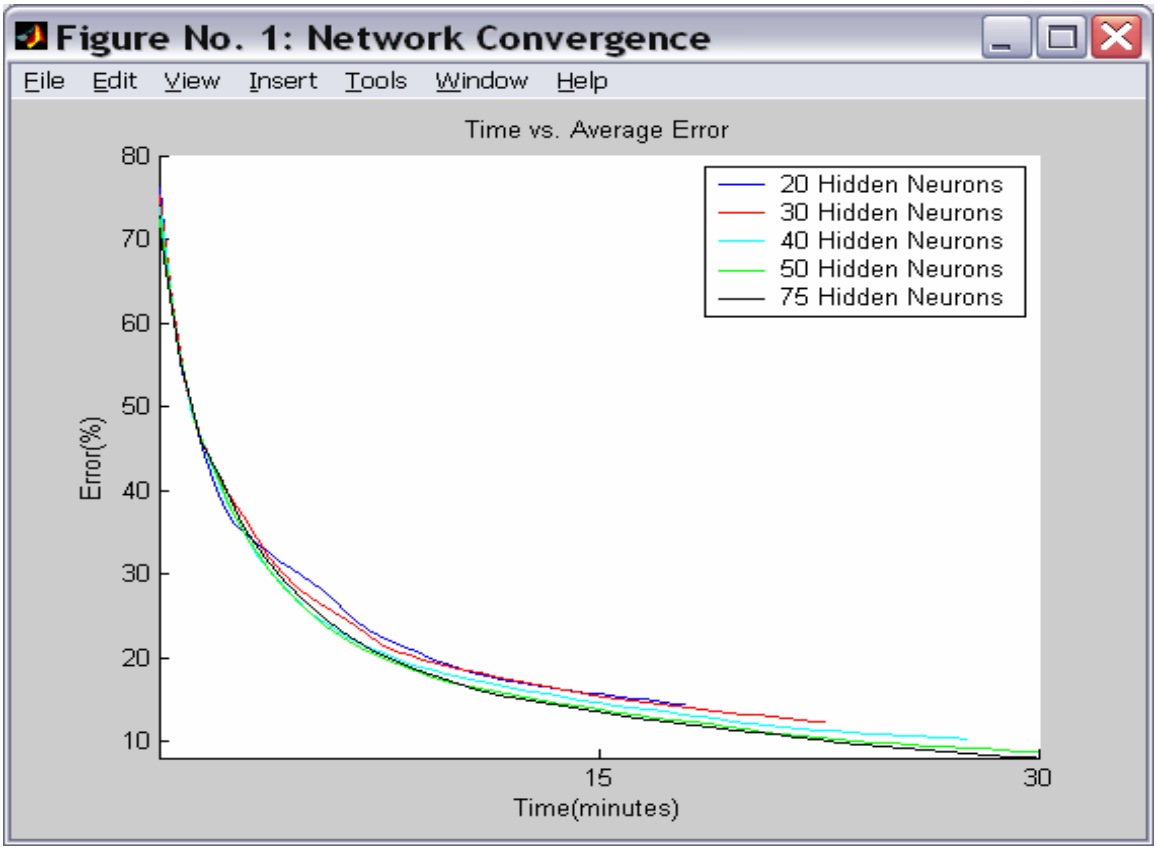


Figure 6-1: Network Convergence

In the first test, network performance is computed by calculating the percentage of correct phonemes determined by the weights on the entire training set. Five different types of trained weights were tested for overall accuracy. The hidden neurons are different for the trained weights sets. The overall performance of the network did change with an increase in the hidden neurons. The maximum accuracy is about 80%. The networks were trained on an Intel Pentium 4 2.80 GHz computer. Table 6-1 shows the overall accuracy and training results for each network tested.

Table 6-1: First Test Results

Network	**TTP Accuracy (%)**	**Training Time (P4 2.80GHz)**	**Epochs**	**Hidden Nodes**
1	66.93	15 min 10s	75	20
2	68.81	19 min 30s	75	30
3	72.45	22 min 59s	75	40
4	75.77	25 min 13s	75	50
5	80.16	35 min 46s	75	75

In the second test, network performance is computed by calculating the percentage of correct phonemes calculated on 100 random unknown words (i.e., words that were not in the training set). The same trained weights were tested for overall accuracy. The overall performance, shown in Table 6-2, is about 67%, which is lower than the performance in the first test. The performance is lower in the second test because of the rule expectations in the English language. For example, the word "vice" is pronounced with a /ie/ phoneme for the "i". But the "i" in the word "service" is pronounced with a /uh/ phoneme. Consequently, the network does not catch expectations and hence the performance is lower with unknown text.

Table 6-2: Second Test Results

Network	**TTP Accuracy (%)**
1	62.78
2	63.09
3	63.59
4	67.12
5	67.52

Overall, these tests show that the network is about 80% accurate on the training data. When data not in the training set is introduced, the network performs around 67% accuracy. When testing the TTS system manually, the text-to-phoneme conversion accuracy is good with some minor faults. The TTP errors of the system might go unnoticed to the user. For example, if the correct (trained) phonetic representation of the word "the" is "\th\ \u\" and the system produces "\th\ \ee\", the user will not notice the incorrect TTP conversion. On the whole, this network is accurate enough to produce the correct text-to-phoneme conversion without many major errors.

6.2 Fuzzy Fundamental Frequency Results

Testing the fuzzy system can be complicated. Comparing the sound of the speech produced by the system is somewhat subjective. However, the research objective was to make to the synthesized speech sound more natural using fuzzy logic. Therefore, the fuzzy controller was tested by evaluating the f0 contour produced by the fuzzy system. Two types of tests were used to evaluate the fuzzy system. In the first test, the f0 contour

produced by the system was analyzed to see if it was producing a natural looking f0 contour trend. With speech, only voiced sounds create fundamental frequency. Therefore, some phonemes will not have fundamental frequency. The fuzzy system was tested by evaluating the three different sentences types. Figure 6-2, 6-3, and 6-4 show the output produced by the system for a declarative sentence, an interrogative question, and a yes/no question, respectively. The sentences are, “My name is Jonathan Williams”, “What time is the thesis defense?”, and “Is it going to rain at noon?" All three f0 contours produced by the fuzzy system are the accurate f0 trends.

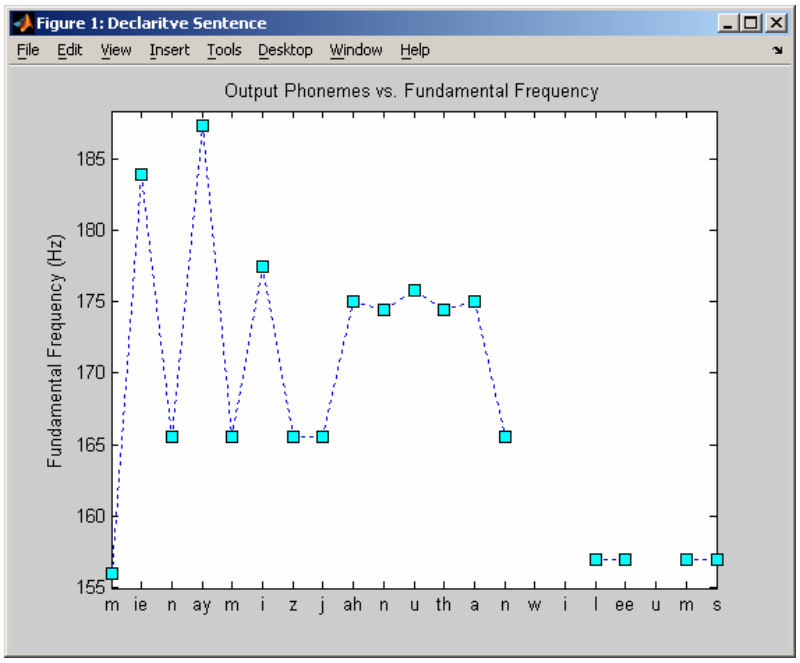


Figure 6-2: Declarative Sentence - “My name is Jonathan Williams”

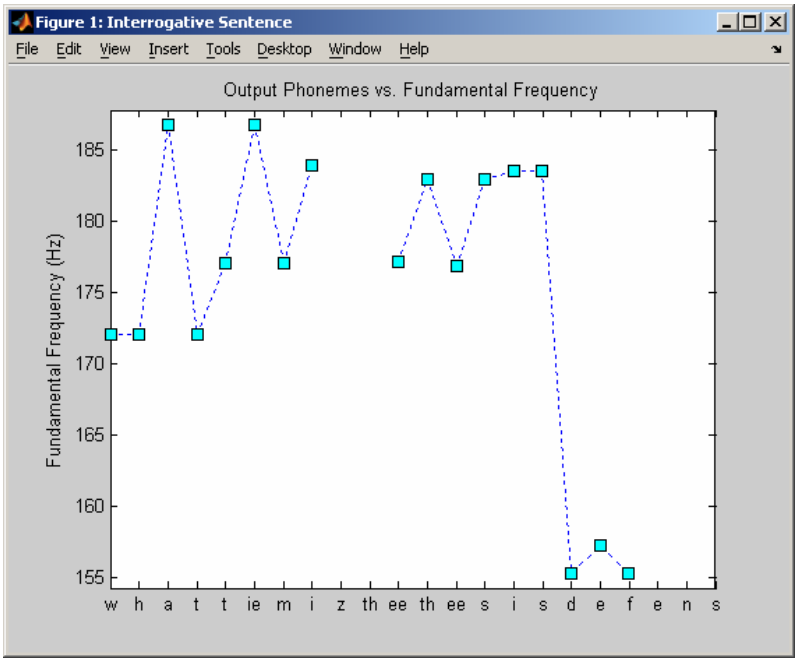


Figure 6-3: Interrogative Question - “What time is the thesis defense?”

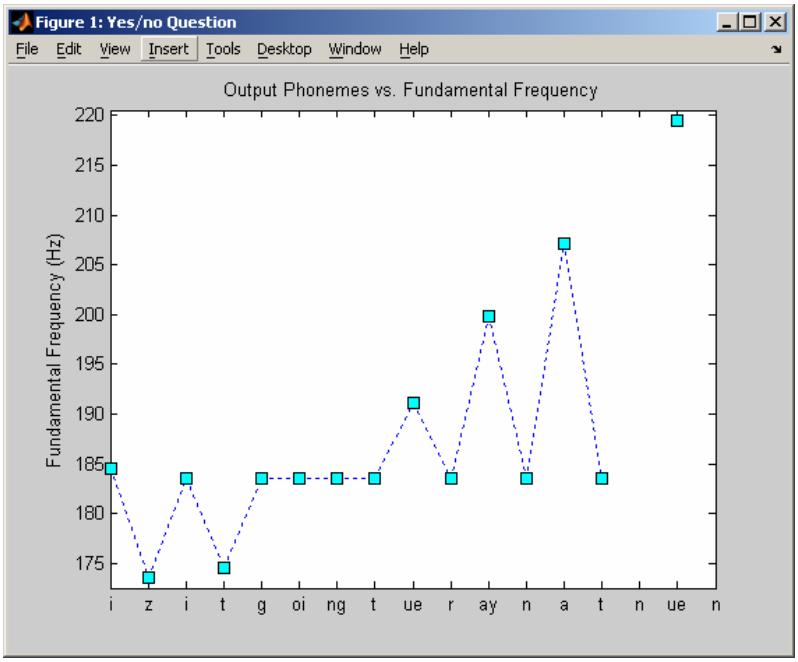


Figure 6-4: Yes/no Question - “Is it going to rain at noon?"

The second test involves viewing the f0 output produced by the fuzzy system with the f0 output of a *high quality* system. The purpose is to further prove that the fuzzy system’s f0 contours are natural looking. It would be impossible to make a comparison

between both systems, since they are completely different systems. The other f0 contour is the Microsoft Research Speech Technology Asia (MRSA) on-line TTS system, which is one of the best unit-selection based TTS systems [37]. The MRSA system's output f0 contour is determined using the MBROLIGN program. The MBROLIGN program is a tool that aligns phonetic transcripts with a speech signal. MBROLIGN then calculates the f0 contour of the speech signal. Using the same sentences, three f0 contours produced by the fuzzy system are graphed with the f0 contours of the MSRA system. The sentence types are declarative sentence, interrogative question, and yes/no question. The following figures show the f0 contour produced by both systems. The sentences are, "My name is Jonathan", "What time is it?" and, "Is it raining today?"

The f0 contour of declarative sentences should be high at the beginning of the sentence and then drop at the end of the sentence. Peaks are on the primary stresses of the stressed syllables. The peak sizes decrease linearly throughout the sentence. For the f0 contours below, both systems produce a downward linear f0 contour trend with a drop at the end of the sentence. Both systems also produce peaks on the stressed words within the sentence.

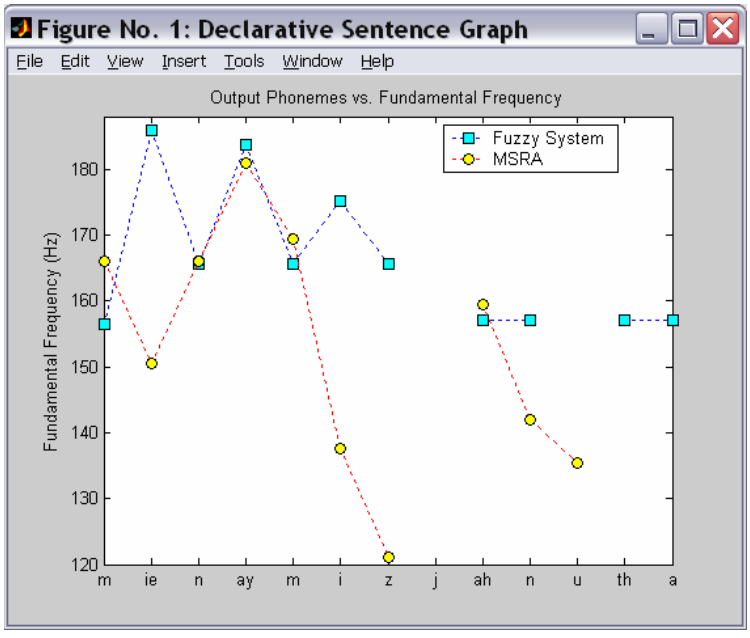


Figure 6-5: F0 Contours for Declarative Sentence, "My name is Jonathan."

The interrogative question f0 contour is similar to the declarative sentence f0 contour because both contours fall at the end of the sentence. The interrogative question

should initially be high with a very sharp drop after the last stress. There should also be peaks around the stresses and the highest peak should be on the first word of the question. These peaks do not fall linearly like the declarative sentence. In the figure below, both contours follow this trend. The fuzzy f0 contour peaks on the initial words in the sentence. Then the fuzzy system's f0 contour sharply falls at the end of the sentence.

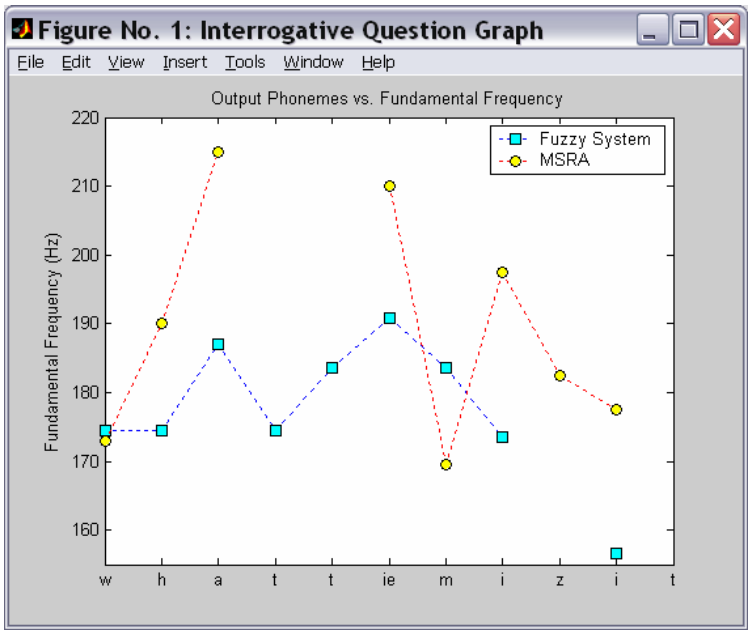


Figure 6-6: F0 Contours for Interrogative Question, "What time is it?"

The yes/no question f0 contour should be relatively low at the beginning with a sharp rise at the end. The peaks throughout the question are smaller compared to the final rise. The graph below shows that the fuzzy f0 contour raises at the end the sentence. The fuzzy controller also has peaks in the middle of the sentence.

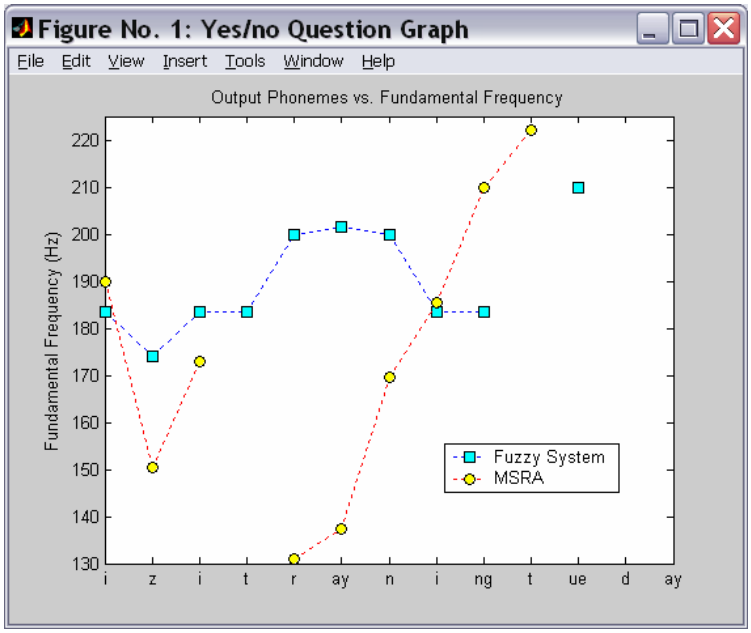


Figure 6-7: F0 Contour Comparison for Yes/no Question, “Is it raining today?”

Chapter 7: Conclusion

7.1 Summary and Conclusion

The goal was to build a TTS system with fuzzy logic controlling the fundamental frequency of the speech. First, a neural network based on the NetTalk network was trained to convert text into phonemes. The training set contained 1800 words with their phonetic transcription. The Back Propagation algorithm was used to train a three layer network. Then stress was assigned using the Halle and Keyer lexical stress rules. Next, segmental duration was calculated using the Klatt duration rules which were designed to replicate observed segmental durations from a speaker. A fuzzy inference system was built to control the fundamental frequency of the speech output. The inputs to the fuzzy controller were word importance, sentence size, stress location, and sentence position. The output was the fundamental frequency. The system had three sets of rules for three types of sentences: declarative sentence, interrogative question, and yes/no question. Finally, the phonemes and calculated prosodic information was set to the MBROLA program to produce the final speech.

The contribution to this thesis is the idea of using a fuzzy controller to control fundamental frequency. No publications on using fuzzy logic to control fundamental frequency were found. The fuzzy controller generates the expected f0 contour for each of the three sentence types. Thus, the system produces more natural sounding speech. Since evaluating the speech produced by the system is subjective, the f0 contour is compared to a high quality TTS system's f0 contour. In comparison, the fuzzy system produces similar f0 trends compared to the high quality TTS system. However, there are many improvements that can be made to the fuzzy fundamental frequency controller.

Overall, the final speech produced by the TTS system sounds more natural. The speech is very understandable and the user can hear the intonation. The purpose of this thesis was to explore the possibilities of using fuzzy logic for automatic prosodic control. This thesis proves that fuzzy logic can a make low memory method of speech synthesis sound more natural. Currently, unit selection is the best sounding speech synthesis method. However, the speech databases must be segmented and the speech databases

sizes are measured in gigabytes. Compared to unit selection-based synthesis, the diphone database does not need to be segmented and the speech database requires a small amount of memory. Using the MBROLA diphone concatenation program, the total program size is under 8 MB. The TTS system itself is less than a megabyte in size and the diphone database is only 6.75MB. With further research, fuzzy logic controlled speech could be the preferred speech synthesis method for our shrinking electronics.

7.2 Future Work

The neural network text to phoneme conversion accuracy needs to improve. Currently, the Back Propagation algorithm yields the best correctness. Different methods like Bi-Direction Recurrent Networks (BRNN) and Self-Organizing Maps (SOM) produce about 70% TTP conversion accuracy [7]. Performance can be improved in many ways. One way to improve network performance is to increase the size of the training set. The results showed that the network produces 80% accuracy on known text and about 70% accuracy of unknown text. Therefore, minimizing the amount of unknown text during training would yield better results. However, the algorithm would still produce errors. Improvements to the Back Propagation method or other machine learning methods need to be further researched in order for machine learning methods to be comparable to the current technologies. Also, research into error correction after training would be very beneficial to the acceptance of the machine learning method.

Improvements to certain components of the TTS system would produce more natural-sounding speech. The stress assignment algorithm can be greatly improved. The current stress assignment method does not assign stress with complete accuracy. Improvements to the system stress algorithm would improve the prosody and naturalness of the speech output. A part-of-speech parser also needs to be added to the system. The current TTS system only handles two types of words: function and content. Assigning word importance based on the part-of-speech would also improve the speech output.

Although the fuzzy controller works adequately, the fuzzy system needs improvements. One improvement can be on the actual rules of the fuzzy system. Decreasing the amount of rules could improve the speed, since the system computation requirements would decrease. The system currently has over 50 rules. Another

improvement can be on the composition of the rules. There needs to be research into designing expert rules for a fuzzy system controlling prosody. If more research went into creating expert fuzzy prosodic rules, a fuzzy system could control fundamental frequency and other aspects of prosody, like segmental duration and stress assignment. The membership functions may not be optimized in the fuzzy system. Research into different types of membership function could also produce more natural sounding speech. The use of different types of membership functions or the use of ANFIS could improve the fuzzy system.

References

1. Adamson, M. J., and Robert I. Damper. "A Recurrent Network that Learns to Pronounce English Text", *ICSLP-1996*, (1996), 1704-1707.

2. Allen, John, Hunnicutt, Sharon, and Dennis Klatt. *From Text To Speech, The MITTALK System*. Cambridge: Cambridge University Press, 1987.

3. Anderson, Mark, and Janet Pierrehumbert. "Synthesis by Rule of English Intonation", *Proceedings of the International Conference on Acoustic, Speech and Signal Processing* 84, (1984), 281-284.

4. Arcieniegas, Fabio and Mark J. Embrechts. "Text-To-Speech Conversion with Staged Neural Networks", *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Vol. 10*, C. H. Dagli et al., eds., ASME Press, (2000), 733-738.

5. "AT&T Natural Voices – Home Page." 2002. http://www.naturalvoices.att.com/ (4/13/2005).

6. Beatrice, Enikö, Astola, Jaakko, and Jukka Saarinen. "Recurrent Neural Network with Both Side Context Dependence for Text-To-Phoneme Mapping*", Proceedings of the IEEE First International Symposium on Control, Communications, and Signal Processing, ISCCSP 2004*, Hammamet, Tunisia, March, (2004).

7. Bilcu, Beatrice, Salmela, Janne, and Jaarinen Saarinen. "Application of the Neural Networks for Text-to-phoneme Mapping", *Proceedings of XI European Signal and Image Processing Conference, EUSIPCO 2002*, Toulouse, France, September, (2002), 892-896.

8. Black, A., and P. Taylor. *The Festival Speech Synthesis System: system documentation*, Human Communications Research Centre, University of Edinburgh, Scotland, January 1997.

9. Brule, James F. "Fuzzy Systems – A Tutorial." 1999. www.austinlinks.com/Fuzzy/tutorial.html/ (6/1/2005).

10. Buhmann, Jeska, Vereecken, Halewijn, Fackrell, Justin, Martens, Jean-Pierre and Bert van Coile. "Data driven intonation modelling of 6 languages", *ICSLP-2000*, vol.3, 179-182 (2000).

11. Campbell, W.N. "Syllable-based segmental duration", *Talking Machines: Theories, Models and Designs*, Amsterdam: Elsevier Science Publishers, 211-224

12. Damper, Robert I., and Y. Marchand. "Improving Pronunciation by Analogy for Text-to-Speech Applications", *SSW3-1998*, (1998), 65-70.

13. Dedina, M. J., and H. C. Nusbaum. "PRONOUNCE: A program for pronunciation by analogy", *Computer Speech and Language*, 5(1):55–64, (1991).

14. Dusterhoff, Kurt E., Black, Alan W. and Paul A. Taylor. "Using Decision Trees within the Tilt Intonation Model to Predict F0 Contours", *Eurospeech 99*, (1999).

15. Dutoit, T., and H. Leich. "MBR-PSOLA: Text-To-Speech Synthesis Based on an MBE Re-Synthesis of the Segments Database", *Speech Communication*, Elsevier Publisher, vol. 13, no. 3-4 (1993).

16. Dutoit, T., *An Introduction to Text-To-Speech Synthesis*¸ Kluwer Academic Publishers, 1996.

17. Fujsaki, H. "The Role of Quantitative Modeling in the Study of Intonation" , *Proceedings of the International Symposium on Japanese Prosody*, (1992), 163-174

18. Gubbins, P.R., Curtis, K.M., and J.D. Burniston. "A Hybrid Neural Network/Rule Based Architecture Used as a Text to Phoneme Transcriber", *Proc. International Symposium on Speech, Image Processing and Neural Networks (ISSIPNN'94),* Hong Kong, April, (1994), 113-116.

19. Hemming, Cecilia. Department of Languages, University College of Skövde Swedish National Graduate School of Language Technology. "Using Neural Networks in Linguistic Resources" 2002. http://www.hemming.se/gslt/LingRes/NeuralNetworks.htm/ (4/19/2005).

20. Klatt, Dennis. "Linguist Use of Segmental Duration in English: Acoustic and Perceptual Evidence", *Journal of the Acoustical Society of America*, 59, vol. 3, 1208-1221, (1976).

21. Klatt, Dennis. "Review of Text-to-Speech Conversion for English", *Journal of the Acoustical Society of America*, 82, vol. 3, 737-793, (1987).

22. Klatt, Dennis. "The Klattalk Text-to-Speech Conversion System", *Proceedings of ICASSP 82,* (1982), 1589-1592.

23. Mattingly, I. G. "Speech Synthesis for Phonetic and Phonological Models", *T.A. Sebeok (Ed.) Current Trends in Linguistics, vol 12, Linguistics and Adjacent Arts and Sciences*, vol. 4, Mouton: The Hague. 2451-2487, (1974).

24. McCulloch, N., Bedworth, M., and J. Bridle. “NETspeak – A Re-implementation of NETtalk*”, Computer Speech and Language*, vol. 2, 284-301, (1987).

25. Nakajima, S. and H. Hamada. "Automatic Generation of Synthesis Units Based on Context oriented Clustering," *Proc. ICASSP-88*, (1987).

26. Öhman, S. “Word and Sentence Intonation: a Quantitative Model”, *K.T.H. Quarterly Progress Report*, vol. 2, 25-54, (1969).

27. Olive, J. P. and N. Spickenagel. “Speech Resynthesis from Phoneme-related Parameters," *JASA*, vol. 59, no. 4, Apr 1976, 993-996.

28. O'Saughnessy, D. *Speech Communication - Human and Machine*, Reading, PA: Addison-Wesley, 1987.

29. “Phoneme.” Encyclopedia on Labor Law Talk [online encyclopedia] (Jelsoft Enterprises Ltd. 2000); available from http://encyclopedia.laborlawtalk.com/ (7/3/2005).

30. Pierrehumbert, Janet. ‘Synthesizing Intonation*”, Journal of the Acoustical Society of America*, vol. 70, no. 4, 1981, 985-995.

31. Taylor, P. “The Tilt Intonation Model”, In R. Mannell and J. Robert-Ribes, eds, *Proc. ICSLP 98*, vol. 4, (1998), 1383–1386.

32. Riley, M.D. “Tree-based Modeling for Speech Synthesis”, *Talking Machines: Theories, Models, and Designs*, G. Baily and C. Benoit, eds., North Holland, 265-273, (1992).

33. Schroeder, M. “A Brief History of Synthetic Speech”. *Speech Communication*, vol. 13, 231-237, (1993).

34. Scordilis, M. and J. Gowdy. “Neural Network Based Generation of Fundamental Frequency Contours”, *IEEE International Conference on Acoustics, Speech, and Signal Processing*, (1989), 219-222.

35. Sejnowski, T. J., and C. R. Rosenberg. “Parallel Network that Learn to Pronounce English Text,” *Complex Systems*, vol. 1, no. 1, 145-168, (1987).

36. “Speech Synthesis.” Wikipedia, the Free Encyclopedia [encyclopedia online] (Wikimedia Foundation, Inc.); available from http://en.wikipedia .org/wiki/.

37. “Speech Technology.” Microsoft Research Speech Technology (Asia). 2005. http://www.research.microsoft.com/speech/ (7/19/2005).

38. Stacey, Deborah and Steven Kramer. “Artificial Neural Networks: From McCulloch Pitts to Back Propagation.” Jan. 1999. http://hebb.cis.uoguelph.ca/~skremer/Teaching/27642/BP/BP.html/ (6/2/2005).

39. Sullivan, K.P., and R.I., Damper. “Novel Word Pronunciation: A Cross-lallgtlage Study”, *Speech Communications*, vol. 13, 44l-452, (1993).

40. Tangsangiumvisai, Nisachon. Department of Electrical Engineering, Chulalongkorn University. “Study of LPC Speech” 2004. http://www.student.chula.ac.th/~47704705/index.html/ (5/16/05).

41. “Timeline: Digital Computing, DECTalk” Microsoft Research. 21 April 1998. http://research.microsoft.com/~gbell/Digital/timeline/1983-5.htm (7/15/05).

42. Zadeh, Lofti. “Fuzzy Sets.” *Journal of Information and Control 8*, 338-353, (1965).

Appendix A – Phoneme List

Network	MBROLA	Sound
%	_	silence
a	{	apple
ah	A	Arthur
aw	O	all
ay	EI	able
b	b	cob
ch	tS	notch
d	d	nod
e	E	else
ee	i	even
f	f	for
g	g	jog
h	h	harm
i	I	illness
ie	AI	island
j	Z	garage
k	k	rock
l	l	doll
m	m	palm
n	n	john
ng	N	bong
oh	@U	over
oi	OI	oyster
oo	U	good
ow	aU	out
p	p	drop
r	r	star
s	s	boss
sh	S	wash
t	t	plot
th	T	cloth
u	@	about
ue	u	oodles
uh	V	nut
ur	r=	her
v	v	salve
w	w	show
xh	D	clothe
y	j	yacht
z	z	was
zh	s	seizure

Appendix B – Phoneme Inherit Duration

Phoneme	Minimum(ms)	Maximum (ms)
a	80	230
ah	100	240
aw	100	240
ay	70	150
e	100	190
ee	55	155
i	40	135
ie	150	250
oh	80	220
oi	150	280
oo	60	160
ow	100	260
u	60	120
ue	70	210
uh	60	140
ur	80	180
h	20	80
l	40	80
r	30	80
w	60	80
y	40	80
m	60	70
n	50	60
ng	60	95
f	80	100
s	60	105
sh	80	105
th	60	90
v	40	60
xh	30	50
z	40	75
zh	40	70
b	60	85
d	50	75
g	60	80
k	60	80
p	50	90
t	50	75
ch	50	70
j	50	70
%	25	25

Appendix C – Training Set

a	/u/	guts	/guts/	Renee	/rennay/
abbey	/abbee/	habits	/habits/	reply	/riplie/
able	/aybul/	hacker	/hackur/	report	/ripawrt/
about	/ubowt/	had	/had/	require	/rikwier/
above	/ubuhv/	hags	/hagz/	rest	/rest/
abs	/abz/	hail	/hayl/	result	/rizuhlt/
action	/akshun/	hair	/her/	retch	/rech/
add	/ad/	half	/haf/	return	/riturn/
adjoin	/adjoin/	hall	/hawl/	rich	/rich/
advance	/udvans/	hallo	/hawloh/	riches	/richiz/
affix	/ufiks/	halo	/hayloh/	ridden	/ridun/
afraid	/ufrayd/	halt	/halt/	ride	/ried/
after	/aftur/	halve	/hav/	right	/riet/
again	/ugen/	hamper	/hampur/	ring	/ring/
age	/ayj/	hand	/hand/	rise	/riez/
agree	/ugree/	handle	/handul/	river	/rivur/
ahoy	/uhoi/	handy	/handee/	road	/rohd/
air	/er/	hang	/hang/	rock	/rawk/
airplane	/erplayn/	happen	/hapun/	roll	/rohl/
airway	/erway/	happy	/hapee/	room	/ruem/
all	/awl/	hard	/hahrd/	round	/rownd/
alley	/alee/	hardly	/hawrdlee/	row	/roh/
alloy	/owlloi/	hardy	/hawrdee/	royal	/roiyul/
along	/ulawng/	hare	/her/	ruby	/ruebee/
already	/awlredi/	has	/haz/	ruches	/roochiz/
also	/awlsoh/	hasty	/haystee/	rule	/ruel/
although	/awlthoh/	hat	/hat/	rummy	/ruhmee/
always	/awlwayz/	hatch	/hach/	run	/ruhn/
am	/am/	have	/hav/	rush	/ruhsh/
amount	/umownt/	he	/hee/	rush	/rush/
and	/and/	head	/hed/	rushes	/ruhshiz/
anger	/anggur/	heal	/heel/	rusty	/ruhstee/
Anglo	/angloh/	health	/helxh/	sad	/sad/
angry	/anggree/	health	/helth/	safety	/sayftee/
animal	/anumul/	heap	/heep/	said	/sed/
annex	/aneks/	hear	/hir/	sail	/sayl/
annoy	/unnoi/	heard	/hurd/	salt	/sawlt/
anoint	/unoint/	heart	/hahrt/	same	/saym/
anomie	/unahmee/	heat	/heet/	sandy	/sandee/
anomy	/unahmee/	heater	/heetur/	sashes	/sashiz/
another	/unuhthur/	heaven	/hevun/	sat	/sat/
anoxia	/anoksu/	heavy	/hevee/	Saturday	/Saturday/
answer	/ansur/	hedge	/hej/	save	/sayv/
antics	/antiks/	height	/hiet/	saw	/saw/
any	/eni/	held	/held/	say	/say/
apex	/aypeks/	hello	/heloh/	scene	/seen/

appear	/upir/	help	/help/	school	/skuel/
apple	/apul/	hem	/hem/	sea	/see/
April	/ayprul/	her	/hur/	seat	/seet/
arch	/arch/	herbs	/hurbz/	second	/sekund/
are	/ahr/	here	/hir/	see	/see/
arm	/ahrm/	heresy	/herusee/	seed	/seed/
army	/ahrmee/	hero	/heeroh/	seem	/seem/
around	/urownd/	hers	/hurz/	seen	/seen/
array	/array/	hey	/hay/	self	/self/
arrive	/uriev/	hey	/hay/	sell	/sel/
art	/art/	hiccup	/hikup/	sense	/sens/
article	/ahrtukul/	hiding	/hieding/	sent	/sent/
as	/az/	high	/hie/	separate	/seprit/
ash	/ash/	hijack	/hiejak/	September	/septembur/
ashes	/ashiz/	hike	/hiek/	serve	/surv/
ask	/ask/	hikes	/hiekz/	service	/survis/
astray	/ustray/	hill	/hil/	set	/set/
at	/at/	him	/him/	settle	/setul/
attach	/uttach/	hinder	/hindur/	seven	/sevun/
attept	/utempt/	hint	/hint/	several	/sevrul/
August	/Awgust/	hip	/hip/	shabby	/shabbee/
aunt	/ant/	hire	/hieur/	shade	/shayd/
avoid	/uvoid/	his	/hiz/	shake	/shayk/
away	/uway/	history	/histree/	shall	/shal/
ax	/aks/	hit	/hit/	sham	/sham/
axis	/aksis/	hoax	/hoaks/	shame	/shaym/
baby	/baybee/	hobble	/hawbul/	shape	/shayp/
back	/bak/	hobby	/hahbee/	share	/sher/
bad	/bad/	hold	/hohld/	shark	/shahrk/
bag	/bag/	hole	/hohl/	sharp	/shahrp/
balk	/bawlk/	holy	/hohlee/	shave	/shayv/
ball	/bawl/	home	/hohm/	she	/shee/
balmy	/bahlmee/	homely	/hohmee/	shed	/shed/
bangs	/bangz/	homy	/hohmee/	sheep	/sheep/
bank	/bangk/	honey	/huhnee/	sheer	/shir/
banker	/bangkur/	hoof	/hoof/	sheet	/sheet/
barmy	/bahrmee/	hook	/hook/	shelf	/shelf/
bash	/bash/	hooks	/hooks/	shell	/shel/
basket	/baskut/	hoops	/huepz/	shield	/sheeld/
batch	/bach/	hope	/hohp/	shift	/shift/
bath	/bath/	horrid	/hawrid/	shine	/shien/
baths	/bathz/	horse	/hawrs/	ship	/ship/
battle	/batul/	hot	/haht/	ships	/shipz/
batty	/batee/	hotty	/hawtee/	shirk	/shirk/
baulk	/bowlk/	hour	/owr/	shirt	/shurt/
bawdy	/bawdee/	hours	/owrz/	shit	/shit/
bay	/bay/	house	/hows/	shock	/shahk/
bay	/bay/	how	/how/	shoe	/shue/
be	/bee/	however	/howevur/	shoot	/shuet/
bean	/been/	hugs	/huhgz/	shop	/shahp/

beauty	/beeuetee/	hulk	/hohlk/	shore	/shawr/
became	/bikaym/	humid	/huemid/	short	/shawrt/
because	/bikawz/	humor	/huemawr/	shot	/shaht/
become	/bikuhm/	hump	/huhmp/	should	/shood/
bed	/bed/	hums	/huhmz/	shoulder	/shohldur/
beech	/beech/	hunch	/huhnch/	shout	/showt/
been	/bin/	hungry	/hunggree/	shove	/shuhv/
before	/bifawr/	hunt	/huhnt/	show	/shoh/
began	/bigan/	hurry	/huree/	shower	/showur/
begin	/bigin/	hurt	/hurt/	shown	/shohn/
behind	/bihiend/	husband	/huhzbund/	shred	/shred/
being	/beeing/	I	/ie/	shrill	/shril/
believe	/bileev/	ice	/ies/	shrine	/shrien/
bell	/bel/	idea	/iediu/	shrink	/shrink/
belong	/bilawng/	if	/if/	shrug	/shruhg/
below	/biloh/	iffy	/iffee/	sick	/sik/
bends	/bendz/	ilk	/ilk/	side	/sied/
berth	/berth/	ill	/il/	sight	/siet/
beside	/busied/	important	/impawrtunt/	sights	/siets/
best	/best/	in	/in/	sign	/sien/
betray	/beetray/	inch	/inch/	silk	/silk/
better	/betur/	inches	/inchiz/	silver	/silvur/
between	/bitween/	include	/inklued/	simple	/simpul/
beyond	/biyahnd/	increase	/inkrees/	since	/sins/
bicycle	/biesikul/	indeed	/indeed/	sing	/sing/
big	/big/	industry	/industree/	single	/singgul/
bilk	/bilk/	inside	/insied/	sister	/sistur/
birch	/burch/	instead	/insted/	six	/siks/
bird	/burd/	into	/intue/	sixty	/siktee/
birth	/berth/	iron	/ieurn/	size	/siez/
black	/blak/	is	/iz/	skulk	/skuhlk/
bleach	/bleech/	it	/it/	sleep	/sleep/
blood	/bluhd/	itch	/ich/	slept	/slept/
bloody	/bluhdee/	its	/its/	slow	/sloh/
blow	/bloh/	jab	/jab/	small	/smawl/
blue	/blue/	jabber	/jabur/	smell	/smel/
blush	/bluhsh/	jack	/jak/	smoke	/smohk/
board	/bawrd/	jacket	/jakit/	snow	/snoh/
boat	/boht/	jade	/jayd/	so	/soh/
body	/bahdee/	jail	/jayl/	soft	/sawft/
boil	/boil/	jam	/jam/	soil	/soil/
bone	/bohn/	jammy	/jamee/	sold	/sohld/
book	/book/	jangle	/jangul/	soldier	/sohljur/
books	/books/	jar	/jahr/	solo	/sohloh/
born	/bawrn/	jargon	/jahrgawn/	some	/suhm/
borrow	/bawroh/	jaunt	/jawnt/	son	/suhn/
botch	/bawch/	jaw	/jaw/	song	/sawng/
both	/bohxh/	jazz	/jaz/	soon	/suen/
bottle	/bawtul/	jazzy	/jazee/	sorry	/sahree/
bounds	/bowndz/	jeans	/jeenz/	sort	/sawrt/

box	/bahks/	jeer	/jeer/	sound	/sownd/
boxer	/bahksur/	jelly	/jelee/	south	/sowth/
boxy	/bahksee/	jemmy	/jemee/	soy	/soi/
boy	/boi/	jerk	/jerk/	space	/spays/
boy	/boi/	Jeron	/jerahn/	speak	/speek/
boyish	/boiish/	jersey	/jursee/	special	/speshul/
brain	/brayn/	jest	/jest/	spend	/spend/
brains	/braynz/	jester	/jestur/	spent	/spent/
branch	/branch/	Jesus	/jeesuhs/	spoke	/spohk/
branch	/branch/	jet	/jet/	spot	/spaht/
brash	/brash/	jetty	/jetee/	spread	/spred/
breach	/breech/	Jew	/jue/	spring	/spring/
bread	/bred/	jewel	/juel/	square	/skwer/
break	/brayk/	jib	/jib/	stalk	/stahlk/
bridge	/brij/	jiffy	/jifee/	stand	/stand/
briefs	/breefs/	jig	/jig/	star	/stahr/
bright	/briet	jiggle	/jigul/	starch	/stahrch/
bring	/bring/	jigsaw	/jigsaw/	start	/stahrt/
broad	/brawd/	jilt	/jilt/	state	/stayt/
broke	/brohk/	jingle	/jingul/	station	/stayshun/
broken	/brohkun/	jitter	/jittur/	stay	/stay/
broth	/brahth/	jive	/jiev/	step	/step/
brought	/brawt/	job	/jahb/	stick	/stik/
brown	/brown/	jockey	/jawkee/	still	/stil/
brunch	/bruhnch/	jog	/jahg/	stitch	/stich/
bucks	/buhks/	join	/join/	stock	/stawk/
buddy	/buhdee/	joint	/joint/	stone	/stohn/
bug	/buhg/	joke	/johk/	stood	/stood/
build	/bild/	joker	/johkur/	stop	/stahp/
building	/bilding/	jolly	/jawlee/	store	/stawr/
built	/bilt/	Jon	/jahn/	storm	/stawrm/
bulk	/buhlk/	joshes	/jawshiz/	story	/stawree/
burn	/burn/	joy	/joi/	straight	/strayt/
bus	/buhs/	judge	/juhj/	strange	/straynj/
bush	/boosh/	jug	/juhg/	stranger	/straynjur/
bushes	/booshiz/	juice	/jues/	stream	/streem/
business	/biznus/	juke	/juek/	street	/street/
busty	/buhstee/	July	/Joolie/	strength	/strengxh/
busy	/bizee/	jumble	/juhmbul/	strike	/striek/
but	/buht/	jump	/juhmp/	strong	/strawng/
butter	/buhtur/	June	/Juen/	student	/stuedunt/
buy	/bie/	jungle	/juhngul/	study	/stuhdee/
by	/bie/	junior	/juenyur/	sturdy	/sturdee/
cake	/kayk/	junk	/juhnk/	subject	/suhbjikt/
call	/kawl/	just	/juhst/	succeed	/sukseed/
came	/kaym/	kecks	/keks/	success	/sukses/
can	/kan/	keep	/keep/	such	/suhch/
captain	/kaptun/	kept	/kept/	sudden	/suhdun/
car	/kahr/	keshes	/keeshiz/	suffer	/suhfur/
care	/ker/	key	/kee/	sugar	/shoogur/

carry	/karee/	kicks	/kikz/	suit	/suet/
case	/kays/	kidney	/kidnee/	sulk	/suhlk/
cash	/kash/	kilo	/keeloh/	summer	/suhmur/
cashes	/kashiz/	kind	/kiend/	sun	/suhn/
catch	/kach/	king	/king/	Sunday	/Suhnday/
caught	/kawt/	kiss	/kis/	supply	/suhplie/
caulk	/kawlk/	kitchen	/kichun/	suppose	/supohz/
cause	/kawz/	kitty	/kitee/	sure	/shoor/
cello	/cheloh/	labor	/laybur/	surprise	/supriez/
cent	/sent/	laches	/lachiz/	sweet	/sweet/
century	/senchuri/	ladder	/ladur/	switch	/swich/
chafe	/chayf/	lady	/laydee/	syntax	/sintaks/
chair	/cher/	lake	/layk/	system	/sistum/
chair	/cher/	land	/land/	table	/taybul/
chalk	/chawlk/	language	/langgwij/	tail	/tayl/
chance	/chans/	large	/lahrj/	take	/tayk/
chance	/chans/	lashes	/lashiz/	talem	/taykun/
change	/chang/	last	/last/	talk	/tawk/
chant	/chant/	late	/layt/	talks	/tawlkz/
chaos	/kayahs/	laugh	/laf/	tall	/tawl/
character	/karuktur/	laughter	/laftur/	tammy	/tamee/
charge	/chahrj/	law	/law/	taste	/tayst/
charm	/chahrm/	lax	/laks/	tasty	/taystee/
chase	/chays/	lay	/lay/	tax	/taks/
chat	/chat/	lead	/leed/	taxi	/taksee/
cheat	/cheet/	leader	/leedur/	teach	/teech/
cheer	/cheer/	learn	/lurn/	tear	/tir/
chesty	/chestee/	least	/leest/	tell	/tel/
chew	/chue/	leave	/leev/	ten	/ten/
chief	/cheef/	leches	/lechiz/	testy	/testee/
child	/chield/	led	/led/	thai	/tie/
childhood	/chieldhood/	left	/left/	than	/than/
children	/childrun/	leg	/leg/	thank	/xhangk/
chill	/chil/	legs	/legz/	thanks	/thanks/
chin	/chin/	length	/lengxh/	that	/that/
choke	/chohk/	less	/les/	the	/thee/
choose	/chuez/	let	/let/	theft	/theft/
chore	/chawr/	letter	/letur/	their	/ther/
Christ	/chriest/	liar	/lier/	theirs	/therz/
chunk	/chuhnk/	lie	/lie/	them	/them/
church	/church/	life	/lief/	theme	/theem/
churn	/churn/	light	/liet/	then	/then/
cigarette	/siguret/	limy	/liemee/	thence	/thens/
circle	/surkul/	line	/lien/	theory	/thirree/
city	/sitee/	liquid	/likwid/	there	/ther/
civics	/siviks/	liquor	/likur/	therefore	/therfawr/
class	/klas/	list	/list/	these	/theez/
clay	/klay/	listen	/lisun/	thesis	/theesus/
clean	/kleen/	little	/litul/	they	/thay/
clear	/klir/	live	/liv/	thick	/xhik/

clock	/klahk/	loath	/lohth/	thief	/theef/
close	/klohs/	lobby	/lahbee/	thigh	/thie/
cloth	/klawxh/	loin	/loin/	thin	/xhin/
clothes	/klohz/	loiter	/loitur/	thing	/xhing/
cloud	/klowd/	lone	/lohn/	things	/thingz/
clutch	/kluhch/	long	/lawng/	think	/xhingk/
coat	/koht/	look	/look/	third	/thurd/
coil	/koil/	Lord	/Lawrd/	thirst	/thurst/
coin	/koin/	lose	/luez/	thirteen	/xhurteen/
cold	/kohld/	loss	/laws/	thirty	/thurstee/
college	/kahlij/	lost	/lawst/	this	/this/
color	/kuhlur/	lot	/laht/	those	/thohz/
come	/kuhm/	loud	/lowd/	though	/thoh/
company	/kuhmpunee/	love	/luhv/	thought	/xhawt/
complete	/kumpleet/	low	/loh/	thousand	/xhowzund/
condition	/kundishun/	lower	/lohur/	thrall	/thral/
consider	/kunsidur/	lox	/lawks/	thrash	/thrash/
considerable	/kunsidurubul/	loyal	/loiyul/	thrawn	/thrawn/
contain	/kuntayn/	lurch	/lurch/	thread	/thred/
continue	/kuntinyue/	lushes	/luhshiz/	threat	/thret/
control	/kuntrohl/	lusty	/lahbee/	three	/xhree/
convey	/kohnvay/	lynch	/linch/	threw	/xhrue/
convoy	/kahnvoi/	Ma	/Mah/	thrice	/thries/
cook	/kook/	machine	/musheen/	thrill	/thril/
cool	/kuel/	mad	/mad/	thrive	/thriev/
corn	/kawrn/	made	/mayd/	throb	/thrahb/
corner	/kawnur/	mail	/mayl/	throe	/throh/
coshes	/kohshiz/	mains	/maynz/	throne	/throhn/
cost	/kawst/	make	/mayk/	throng	/thrawng/
could	/kood/	man	/man/	through	/xhrue/
count	/kownt/	manner	/manur/	throve	/throhv/
country	/kuhntree/	many	/meni/	throw	/xhroh/
course	/kawrs/	March	/mahrch/	thrown	/xhrohn/
cover	/kuhvur/	mark	/mahrk/	thru	/thrue/
cox	/kahks/	market	/mahrkut/	thrust	/thruhst/
crash	/krash/	marque	/mahrk/	thug	/thuhg/
crips	/krips/	marry	/maree/	thumb	/thuhm/
cross	/kraws/	marsh	/mawrsh/	thumbs	/thuhmz/
crowd	/krowd/	master	/mastur/	thus	/thuhs/
crusty	/crestee/	material	/mutiriul/	tiches	/tichiz/
cry	/krie/	maths	/mathz/	tidy	/tiedee/
cup	/kuhp/	matter	/matur/	tie	/tie/
cut	/kuht/	maxim	/maksem/	till	/til/
daddy	/dadee/	May	/May/	time	/tiem/
daily	/daylee/	mayor	/mayur/	to	/tue/
dance	/dans/	me	/mee/	today	/tooday/
dare	/der/	mean	/meen/	together	/toogethur/
dark	/dahrk/	means	/meenz/	toil	/toil/
darts	/dahrtz/	measure	/mezhur/	told	/tohld/
dashes	/dashiz/	meat	/meet/	tommy	/tahmee/

date	/dayt/	meaty	/meetee/	tomorrow	/toomahroh/
daughter	/dawtur/	meet	/meet/	too	/tue/
day	/day/	meeting	/meeting/	took	/took/
dayglo	/daygloh/	member	/membur/	tooth	/tueth/
dead	/ded/	men	/men/	torch	/tawrch/
deal	/deel/	met	/met/	tore	/tawr/
dear	/dir/	method	/mexhud/	touch	/tuhch/
December	/Disembur/	meths	/methz/	toward	/tawrd/
decide	/disied/	middle	/midul/	town	/town/
decks	/dekz/	might	/miet/	toxic	/tawksik/
decoy	/deekoi/	mile	/miel/	toy	/toi/
deep	/deep/	milk	/milk/	tracks	/traks/
degree	/digree/	million	/milyun/	trade	/trayd/
deity	/deetee/	mind	/miend/	train	/trayn/
delight	/diliet/	mine	/mien/	training	/trayning/
demand	/dimand/	minute	/minit/	travel	/travul/
deploy	/deeploi/	miss	/mis/	tree	/tree/
depth	/depth/	mister	/mistur/	tried	/tried/
desire	/dizier/	misty	/mistee/	tries	/triez/
destroy	distroi/	mix	/miks/	trip	/trip/
detach	/deetach/	modern	/mawdurn/	trouble	/truhbul/
detox	/deetahks/	moist	/moist/	trust	/truhst/
device	/divies/	Monday	/Muhnday/	truth	/trueth/
devoid	/deevoid/	money	/mohnee/	try	/trie/
dibs	/dibz/	month	/muhnxh/	Tuesday	/Tuezday/
dicey	/diesee/	moody	/muedee/	turfs	/turfs/
did	/did/	moon	/muen/	turn	/turn/
die	/die.	more	/mawr/	twelve	/twelv/
difference	/difruns/	morning	/mawrning/	twenty	/twentee/
different	/difrunt/	mosque	/mahsk/	twins	/twinz/
difficult	/difukult/	most	/mohst/	twitch	/twich/
dig	/dig/	moth	/mawth/	two	/tue/
digs	/digz/	mother	/muhthur/	udder	/uhdur/
dinner	/dinur/	mountain	/mowntin/	uglier	/uhgleeur/
dioxin	/deeahksin/	mouth	/mowxh/	uglis	/uhglis/
direct	/direkt/	move	/muev/	ugly	/uhglee/
dirty	/dirtee/	movement	/muevmunt/	ulcer	/awlsur/
discover	/diskuhvur/	moxa	/mohksu/	ulema	/ulemu/
dish	/dish/	much	/muhch/	ulna	/awlnu/
dishes	/dishiz/	mud	/muhd/	ultima	/awlteemu/
dismay	/dismay/	munch	/muhnch/	ultra	/awltru/
distance	/distuns/	music	/myuezik/	umbel	/uhmbul/
distant	/distunt/	must	/muhst/	umber	/uhmbur/
ditch	/dich/	my	/mie/	umpire	/uhmpieur/
divide	/divied/	nail	/nayl/	uncle	/uhngkul/
do	/due/	nasty	/nastee/	under	/uhndur/
doctor	/dahktur/	nation	/nayshun/	understand	/uhndurstand/
does	/duhz/	nature	/naychur/	understood	/uhndurstood/
dog	/dawg/	near	/nir/	until	/until/
doing	/dueing/	nearly	/nirli/	up	/uhp/

dollar	/dahlur/	necessary	/nesuseri/	upon	/upahn/
done	/duhn/	neck	/nek	urban	/urban/
donkey	/duhnkee/	need	/need/	urchin	/urchin/
don't	/dohnt/	needle	/needul/	urea	/ureeu/
door	/dawr/	needs	/needz/	urge	/urg/
dormie	/dawrmee/	needy	/needee/	urgent	/urgent/
dormy	/dawrmee/	neighbor	/naybur/	urine	/yurin/
double	/duhbul/	neither	/neethur/	us	/uhs/
doubt	/dowt/	nerve	/nurv/	use	/yuez/
down	/down/	never	/nevur/	use	/yues/
dream	/dreem/	new	/nue/	usual	/yuezhooul/
dregs	/dregz/	news	/nuez/	vac	/vac/
dress	/dres/	next	/nekst/	vacuum	/vakuem/
dried	/dried/	nice	/nies/	vagina	/vahjienu/
drink	/dringk/	niece	/nees/	vague	/vayg/
drive	/driev/	night	/niet/	vain	/vayn/
drop	/drahp/	nine	/nien/	valet	/valay/
dry	/drie/	ninth	/nienth/	valid	/valid/
dubs	/duhbz/	no	/noh/	valley	/valee/
duck	/duhk/	noise	/noiz/	value	/valyue/
dummy	/duhmee/	none	/nuhn/	valve	/valv/
during	/dooring/	noon	/nuen/	vamp	/vamp/
dusk	/duhsk/	nor	/nawr/	van	/van/
dusty	/duhstee/	north	/nawrxh/	vandal	/vandul/
duty	/duetee/	nose	/nohz/	vanish	/vanish/
dwarfs	/dwawrfs/	not	/naht/	vapid	/vaypid/
each	/eech/	notch	/nawch/	vapor	/vaypur/
ear	/ir/	note	/noht/	various	/varius/
early	/urlee/	nothing	/nuhxhing/	vary	/veree/
earth	/urxh/	notice	/nohtis/	vast	/vast/
east	/eest/	November	/NOHvembur/	vault	/vawlt/
easy	/eezee/	now	/now/	vector	/vektur/
eat	/eet/	number	/nuhmbur/	veer	/veer/
edge	/ej/	nutty	/nuhtee/	vegan	/veegin/
effort	/efurt/	nylons	/nielahnz/	vein	/vayn/
egg	/eg/	oath	/ohth/	venom	/venuhm/
eight	/ayt/	oats	/ohts/	vent	/vent/
either	/eethur/	object	/ahbjekt/	verb	/vurb/
electric	/ulektrik/	ocean	/ohshun/	verbs	/vurbz/
electricity	/ulektrisitee/	oches	/ohchiz/	verse	/verz/
elk	/elk/	October	/Ahktohbur/	very	/veree/
else	/els/	odds	/ahdz/	vet	/vet/
Emmy	/emee/	of	/uhv/	via	/veeuh/
employ	/emploi/	off	/awf/	vice	/vies/
end	/end/	offer	/awfur/	video	/videeoh/
ends	/endz/	office	/awfis/	view	/vyue/
enemy	/enumee/	often	/awfun/	view	/vue/
English	/ingglish/	oh	/oh/	vile	/viel/
enjoy	/injoi/	oil	/oil/	visa	/veezu/
enough	/inuhf	oil	/oil/	visit	/vizit/

enter	/entur/	oily	/oilee/	vocal	/vohkul/
epoxy	/eepawksee/	old	/ohld/	vogue	/vohg/
equal	/eekwul/	on	/ahn/	voice	/vois/
equip	/eekwip/	only	/ohnlee/	void	/void/
escape	/uskayp/	oogamy	/uegamee/	vomit	/vawmit/
etches	/etchiz/	open	/ohpun/	vote	/voht/
ethics	/ethiks/	opinion	/upinyun/	vouch	/vowch/
even	/eevun/	or	/awr/	voyage	/voiyeg/
evening	/eevning/	order	/awrdur/	voyeur	/voiyur/
ever	/evur/	orderly	/awrdurli/	wagon	/wagun/
every	/evree/	other	/uhthur/	wait	/wayt/
exact	/iksakt/	ought	/awt/	walk	/wawk/
exalt	/ikawlt/	our	/owr/	wall	/wawl/
exam	/iksam/	out	/owt/	want	/wawnt/
excel	/ikssel/	outer	/owtur/	war	/wawr/
except	/eksept/	outside	/owtsied/	warm	/wawrm/
exert	/iksurt/	over	/ohvur/	was	/wuhz/
exile	/eksiel/	ovolo	/ohvohlu/	wash	/wahsh/
exit	/eksit/	own	/ohn/	wash	/wahsh/
expect	/ikspekt/	ox	/awks/	washes	/washiz/
experience	/ikspiriuns/	oxen	/awksen/	watch	/wahch/
explain	/eksplayn/	oxide	/awksied/	water	/wahtur/
eye	/ie/	page	/payj/	wave	/wayv/
face	/fays/	paid	/payd/	wax	/waks/
fact	/fakt/	pain	/payn/	way	/way/
fail	/fayl/	part	/pahrt/	we	/wee/
fair	/fer/	partial	/pahrshul/	weak	/week/
faith	/fayth/	party	/pahrtee/	wear	/wer/
fall	/fawl/	pass	/pas/	weather	/wethur/
family	/famulee/	past	/past/	wedge	/wej/
famous	/faymus/	pasty	/paystee/	Wednesday	/Wenzday/
fancy	/fansee/	patch	/pach/	weds	/wedz/
far	/fahr/	pay	/pay/	week	/week/
farm	/fahrm/	peace	/pees/	weight	/wayt/
fast	/fast/	peach	/peech/	welcome	/welkum/
fat	/fat/	pegs	/pegz/	well	/wel/
father	/fahthur/	people	/peepul/	went	/went/
fatty	/fatee/	perfect	/purfikt/	west	/west/
favor	/fayvur/	perhaps	/purhaps/	wet	/wet/
fax	/faks/	period	/piriud/	what	/whuht/
fear	/fir/	person	/pursun/	wheat	/wheet/
February	/Febrooeree/	phag	/fag/	wheel	/wheel/
feed	/feed/	phage	/fayg/	whelk	/welk/
feel	/feel/	pharm	/fahrm/	when	/when/
feet	/feet/	phase	/fays/	where	/wher/
feisty	/fiestee/	phatic	/fatik/	whether	/whethur/
fell	/fel/	phenol	/feenul/	which	/which/
fellow	/feloh/	phenyl	/feeniel/	while	/whiel/
felt	/felt/	phew	/pue/	white	/whiet/
fence	/fens/	phi	/fie/	whole	/hohl/

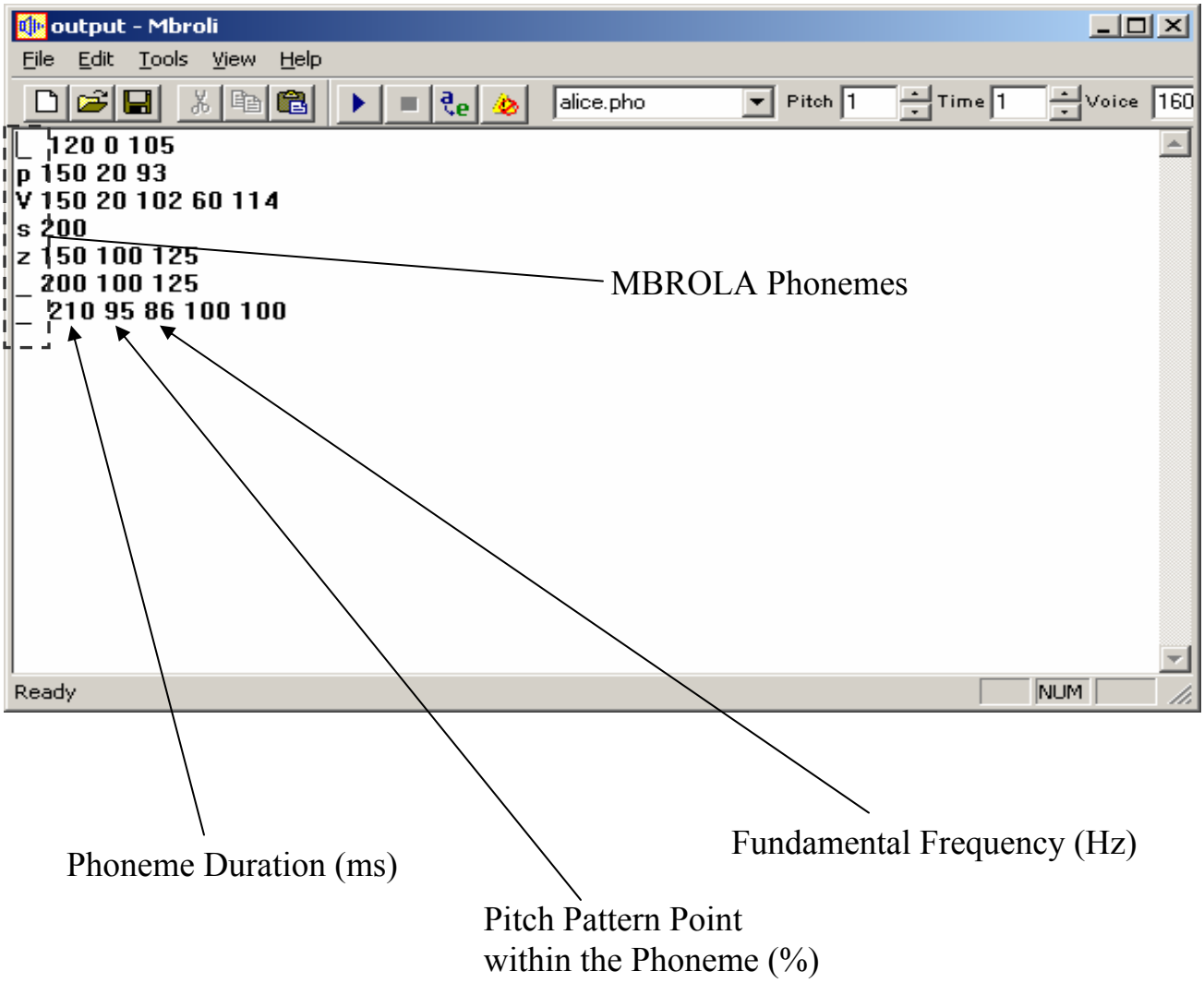
fetch	/fech/	phial	/fieul/	whom	/huem/
few	/fyue/	phiz	/fiz/	whose	/huez/
field	/feeld/	phlox	/flahks/	why	/whie/
fifteen	/fifteen/	phobia	/fohbeeu/	width	/width/
fifth	/fifxh/	phoebe	/feebee/	wife	/wief/
fifty	/fiftee/	phon	/fahn/	wild	/wield/
fight	/fiet/	phone	/fohn/	will	/wil/
figure	/figyur/	phonic	/fahnik/	win	/win/
fill	/fil/	phony	/fohnee/	wind	/wind/
filo	/filoh/	phot	/fawt/	window	/windoh/
filth	/filth/	photo	/fohtoh/	wing	/wing/
find	/fiend/	photon	/fohtahn/	wings	/wingz/
fine	/fien/	phrase	/frays/	winter	/wintur/
finger	/finggur/	phut	/fuht/	wise	/wiez/
finish	/finish/	phyla	/fielu/	wish	/wish/
finish	/finish/	physic	/fisik/	wishes	/wishiz/
fire	/fier/	pick	/pik/	with	/with/
firm	/furm/	picture	/pikchur/	within	/within/
first	/furst/	piece	/pees/	without	/withowt/
fish	/fish/	pinch	/pinch/	witty	/witee/
fishes	/fishiz/	pique	/pik/	woman	/woomun/
fit	/fit/	pith	/pith/	women	/wimin/
five	/fiev/	pity	/pitee/	won	/wuhn/
fix	/fiks/	pixel	/piksul/	wonder	/wuhndur/
fix	/fiks/	place	/plays/	wood	/wood/
fixed	/fikst/	plain	/playn/	word	/wurd/
flabby	/flabee/	plan	/plan/	wore	/wawr/
flier	/flier/	plant	/plant/	work	/wurk/
floor	/flawr/	plaque	/plak/	world	/wurld/
flower	/flowur/	play	/play/	worn	/wawrn/
fly	/flie/	pleasant	/plezunt/	worth	/wurxh/
foamy	/fohmee/	please	/pleez/	would	/wood/
foible	/foibul/	pleasure	/plezhur/	write	/riet/
foil	/foil/	ploy	/ploi/	written	/writun/
folk	/fohlk/	poach	/pohch/	wrong	/rawng/
follow	/fawloh/	point	/point/	wrote	/roht/
food	/fued/	poise	/pois/	yacht	/yaht/
fool	/fuel/	polo	/pohloh/	yack	/yak/
foot	/foot/	poor	/poor/	yah	/yah/
for	/fawr/	position	/puzishun/	yahoo	/yahhue/
force	/fawrs/	possible	/pahsubul/	yajur	/yahjur/
foreign	/fawrun/	pot	/paht/	yak	/yak/
forest	/fawrust/	pouch	/powch/	yakut	/yakuet/
forever	/fawrevur/	power	/powur/	yale	/yayl/
forget	/furget/	pox	/pawks/	yam	/yam/
form	/fawrm/	prepare	/priper/	yang	/yang/
fortieth	/fawrtiuxh/	present	/presunt/	yank	/yank/
forty	/fawrtee/	president	/prezudunt/	Yankee	/yankee/
forward	/fawrwurd/	press	/pres/	yap	/yap/
found	/fownd/	pretty	/pritee/	yappy	/yappee/

four	/fawr/	prey	/pray/	yard	/yahrd/
fousty	/fowstee/	price	/pries/	yarn	/yawrn/
fox	/fahks/	probably	/prahbubli/	yate	/yayt/
foxed	/fahkst/	problem	/prahblum/	yawl	/yawl/
foxy	/fahksee/	produce	/prohdues/	yawn	/yawn/
foyer	/foiyur/	promise	/prahmis/	ye	/yee/
fray	/fray/	proud	/prowd/	year	/yir/
free	/free/	prove	/pruev/	yearly	/yeerlee/
fresh	/fresh/	public	/puhblik	yearn	/yurn/
fresh	/fresh/	pull	/pool/	years	/yirz/
Friday	/Frieday/	pure	/pyoor/	yeast	/yeest/
friend	/frend/	push	/poosh/	yell	/yel/
from	/fruhm/	pushes	/pooshiz/	yellow	/yeloh/
front	/fruhnt/	put	/poot/	yellow	/yelloh/
froth	/frawth/	pygmy	/pigmee/	yelp	/yelp/
full	/fool/	pyramid	/peeramid/	yen	/yen/
further	/furthur/	qua	/kwa/	yes	/yes/
fusty	/fuhstee/	quad	/kwad/	yesterday	/yesturday/
gain	/gayn/	quaff	/kwaf/	yet	/yet/
game	/gaym/	quag	/kwag/	yew	/yue/
garden	/gahrdun/	quail	/kwayl/	yiddish	/yiddish/
gashes	/gashiz/	quake	/kwayk/	yield	/yild/
gate	/gayt/	qualm	/kwawlm/	yo	/yoh/
gather	/gathur/	quart	/kwawrt/	yob	/yahb/
gave	/gayv/	quarter	/kwawrtur/	yodal	/yohdul/
gay	/gay/	quartz	/kwartz/	yoga	/yohgu/
general	/jenurul/	quash	/kwahsh/	Yogi	/yohgee/
gentle	/jentul/	quasi	/kwahzee/	yoke	/yohk/
gentleman	/jentulmun/	quay	/kway/	yolk	/yohlk/
gents	/jents/	queen	/kween/	yonder	/yahndur/
get	/get/	queen	/kween/	yore	/yawr/
gift	/gift/	queer	/kweer/	you	/yue/
girl	/gurl/	quell	/kwel/	young	/yuhng/
give	/giv/	query	/kweeree/	your	/yawr/
glad	/glad/	quest	/kwest/	yours	/yawrz/
glass	/glas/	question	/kweschun/	youth	/yueth/
gloom	/gluem/	queue	/kwue/	yummy	/yuhmee/
gloomy	/gluemee/	quick	/kwik/	zag	/zag/
glossary	/glawsuree/	quiet	/kwieut/	Zaire	/zieir/
gnash	/nash/	quiff	/kwif/	zakat	/zahkat/
go	/goh/	quilt	/kwilt/	zany	/zaynee/
God	/Gahd/	quip	/kwip/	zap	/zap/
goes	/gohz/	quirk	/kwirk/	zeal	/zeel/
gold	/gohld/	quite	/kwiet/	zealot	/zeelawt/
gone	/gawn/	quiz	/kwiz/	zebra	/zeebruh/
good	/good/	quota	/kwohtu/	zebu	/zebue/
goodbye	/goodbie/	quote	/kwoht/	zed	/zed/
got	/gaht/	race	/rays/	zee	/zee/
govern	/guhvurn/	rags	/ragz/	Zen	/zen/
grain	/grayn/	rain	/rayn/	zest	/zest/

grave	/grayv/	raise	/rayz/	zesty	/zestee/
gray	/gray/	ran	/ran/	zeta	/zetu/
great	/grayt/	ranch	/ranch/	ziff	/zif/
green	/green/	ranks	/ranks/	zig	/zig/
grew	/grue/	rather	/rathur/	zilch	/zilch/
grey	/gray/	reach	/reech/	zinc	/zingk/
groin	/groin/	read	/reed/	zine	/zien/
group	/gruep/	ready	/redee/	zip	/zip/
grow	/groh/	real	/reel/	zither	/zixhur/
grown	/grohn/	realize	/reeuliez/	zodiac	/zohdeeak/
guard	/gahrd/	reason	/reezun/	zombie	/zahmbee/
guches	/guhchiz/	receive	/ruseev/	zonal	/zohnul/
guess	/ges/	recoil	/reekoil/	zone	/zohn/
guide	/gied/	record	/rekurd/	zonk	/zahnk/
gummy	/guhmee/	red	/red/	zoo	/zue/
gun	/guhn/	reins	/rayns/	zulu	/zulue/
gushes	/guhshiz/	relax	/reelaks/		
gusty	/guhstee/	remember	/rimembur/		

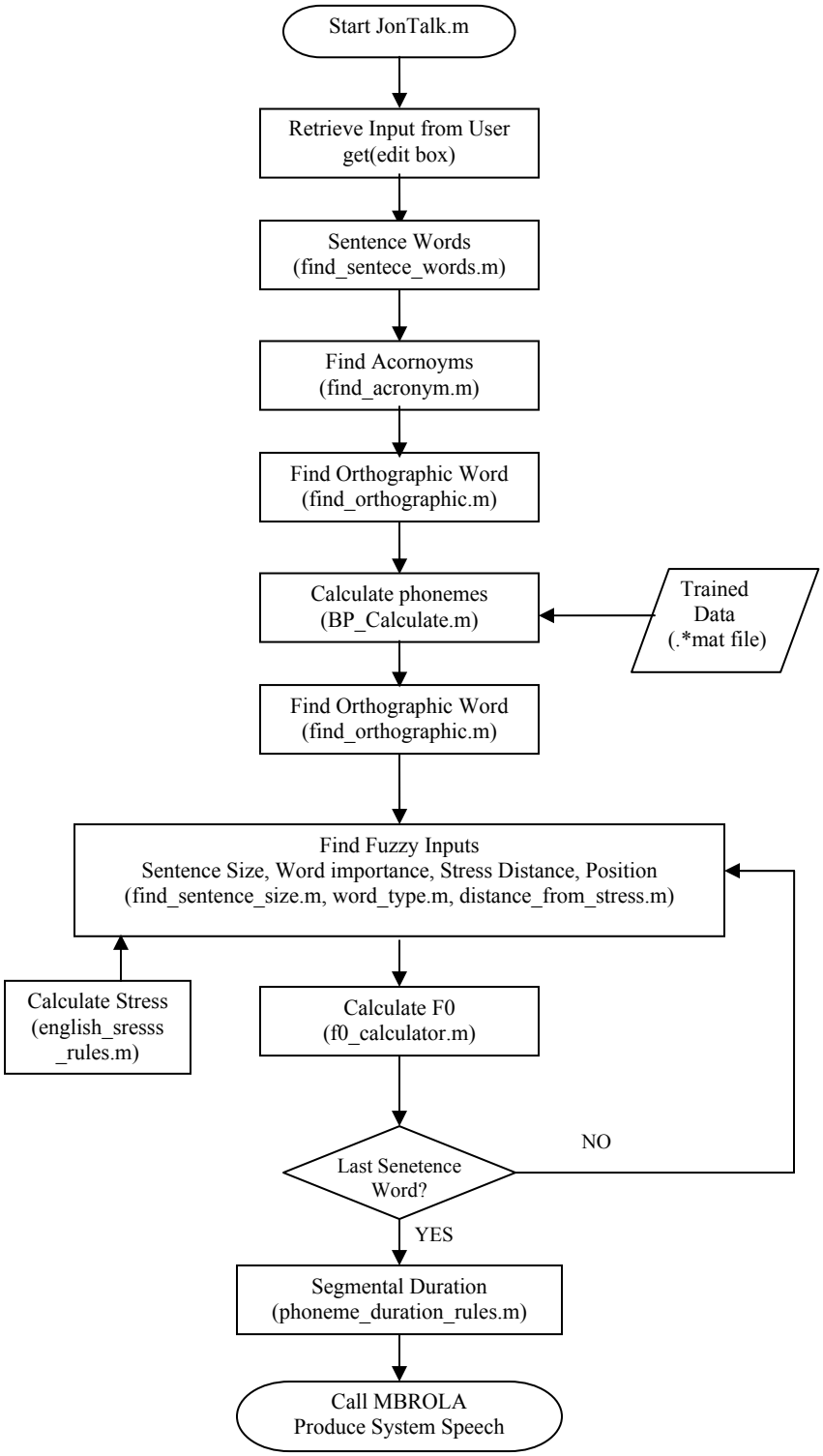
Appendix D – MBROLA Program Description

The MBROLA program is designed to take four variables as inputs. The inputs to the MBROLA program are the phoneme, the phoneme duration, the pitch pattern point, and the fundamental frequency. The program only needs the phoneme and the duration to produce speech. If the user wants to add fundamental frequency values, the program must have a corresponding pitch pattern point. The program can handle up to 20 pitch pattern point and fundamental frequency pairs.



Appendix E – Source Code

JonTalk Flow Chart



```
%--------------------------------------------------------------------
% Function: JonTalk.m
% Purpose: The GUI of the JonTalk text-to-so-speech program
% Description:  This program is the GUI for the TTS system. The program
% handles text input and produces speech output.  The user can control
% the tone of the speech and the speed of the speech.
% Outputs: Speech
%--------------------------------------------------------------------
function varargout = JonTalk(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @JonTalk_OpeningFcn, ...
                   'gui_OutputFcn',  @JonTalk_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before JonTalk is made visible.
function JonTalk_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes JonTalk wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = JonTalk_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
global declarative_sentence_rules
global yes_no_question_rules
global interrogative_question_rules
global wh_question_rules
global rise_fall_rules
global PHONEME_SET
global pitch
global speed

pitch = 1;
```

```
speed = 1;
set(gcf,'Color',[0 0 0]);

linguist_variables{1} = {'useless' 'semi' 'important' 'none'};
linguist_variables{2} = {'small' 'medium' 'large' 'none'};
linguist_variables{3} = {'start' 'middle' 'end' 'none'};
linguist_variables{4} = {'dead-on' 'near' 'far' 'none'};
linguist_variables{5} = {'zero' 'low' 'mid-low' 'mid' 'mid-high' 'high'
'peak'};

%System default rules
yes_no_question_rules = {[1 0 1 0 3] [2 0 1 0 4] [2 3 1 0 3] [0 0 1 2
5] [0 0 1 1 5]...
        [1 0 2 0 3] [1 3 2 0 2] [2 0 2 0 4] [3 0 2 1 5] [0 0 2 1 4] [0
2 2 0 4] [0 3 2 0 3]...
        [0 0 3 1 6] [0 0 3 2 5] [0 3 3 2 6] [1 0 3 2 5]  [3 0 3 1 6]};

declarative_sentence_rules = {[1 0 1 0 2] [2 0 1 0 3] [2 3 1 0 4] [0 1
1 1 6] [0 0 1 1 7]  [0 1 1 0 4]...
        [ 1 0 2 0 2] [1 3 2 0 3] [2 0 2 0 3] [2 3 2 1 4] [0 0 2 1 5] [0
2 2 0 4] ...
        [ 0 0 3 1 2] [3 0 3 2 3] [0 2 3 0 2] [1 0 3 0 2]};

wh_question_rules = {[0 0 1 0 3] [3 0 1 0 4] [2 3 1 0 4] [0 1 1 1 4] [0
0 1 1 5] [0 1 1 0 3] ...
        [ 1 0 2 0 4] [1 3 2 0 3] [0 0 2 2 4] [2 3 2 1 5] [0 0 2 1 5] [3
0 2 1 6] [0 2 2 0 3] ...
        [ 0 0 3 1 3] [0 0 3 2 1] [0 2 3 0 2] [1 0 3 0 1]};

main_menu = uimenu('Label','Options');
uimenu(main_menu,'Label','View Rules','Callback',@Rule_Viewer);
uimenu(main_menu,'Label','Close','Callback','close');

[dummy_variable excel_file_phoneme] = xlsread('Word
Data\phoneme_set.xls');
PHONEME_SET = excel_file_phoneme(:,1);

% --- Executes during object creation, after setting all properties.
function input_screen_CreateFcn(hObject, eventdata, handles)
% hObject    handle to input_screen (see GCBO)
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'))
;
end



function input_screen_Callback(hObject, eventdata, handles)
% hObject    handle to input_screen (see GCBO)
% --- Executes on button press in TTS.

function TTS_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to TTS (see GCBO)

global listbox_handles
global declarative_sentence_rules
global yes_no_question_rules
global wh_question_rules
global rise_fall_rules
global MF_descriptors
global MF_descriptors_rise_fall
global test_axes
global PHONEME_SET
global pitch
global speed

%Membership functions decsriptors
MF_descriptors{1} = [-5 0 5; 0 5 10; 5 10 15];
MF_descriptors{2} = [-5 0 5; 0 5 10; 5 10 15];
MF_descriptors{3} = [-5 0 5; 0 5 10; 5 10 15];
MF_descriptors{4} = [-5 0 5; 0 5 10; 5 10 15];
MF_descriptors{5} = [-2 0 2; 0 2 4; 2 4 6; 4 6 8; 6 8 10; 8 10 12; 10
12 14];

typed_input = get(handles.input_screen,'String');
wh_words = {'how' 'what' 'when' 'where' 'who' 'whom' 'whose' 'why' };

if isempty(typed_input) == 1
    return;
end

%Find the sentence type, depend on pucutation and the first word
if isempty(strfind(typed_input,'?')) == 0
question_marks = strfind(typed_input,'?');
typed_input(question_marks) = ' ';
rule_list = yes_no_question_rules;
for count = 1:length(wh_words)
    if isempty(strfind(lower(typed_input),wh_words{count})) == 0
       rule_list = wh_question_rules;
       break;
    end
end

else
rule_list = declarative_sentence_rules;
end

%Load the trained Back Propagation 85 epoch mat file
loaded_data = load('BPU85epochs.mat');
hidden_weights = loaded_data.hidden_weights;
output_weights = loaded_data.output_weights;

hidden_size = size(hidden_weights);
HIDDEN_NEURONS = hidden_size(1);


typed_input(isspace(typed_input)) = '%';
```

```
for count=1:length(typed_input)
    input_letters{count} = typed_input(count);
end

global COMMAS
COMMAS = strfind(typed_input,',');

%Converts words to individual cell strings
actual_words = find_sentence_words(input_letters,COMMAS);

%Find acronyms, words like NASA or FBI
actual_words = find_acronyms(actual_words);

%Find orthographic words like Mr. Mrs. and numbers
actual_words_normalized = orthographic_converter(actual_words);


sentence_words = '';
output_phonemes = '';

%For each word within the cellstring, calculate the weight output
for main_count=1:length(actual_words_normalized)
word = char(actual_words_normalized{main_count});

%Set the front string to 7 silences (for alignment purposes)
input_size = length(word);
input_string = {'%' '%' '%' '%' '%' '%' '%'};
for count=1:input_size
input_string(7+count) = cellstr(word(count));
end

%Place seven silences at the beginning of the input string
size_string = 7 + input_size;
input_string(size_string+1:size_string+7) = {'%'};
%Send input string, weights, and the other variables to the function
BP_calculate
phonetic_word =
lower(BP_calculate(input_string,output_weights,hidden_weights,PHONEME_S
ET,HIDDEN_NEURONS));
spaces = strmatch('%',phonetic_word);
phonetic_word(spaces) = '';
sentence_words{main_count} = phonetic_word;
output_phonemes = [output_phonemes {'%'} sentence_words{main_count}];
end

% Call this function to remove double phonemes
[output_phonemes sentence_words]  =
double_phoneme_handler(output_phonemes,sentence_words);
sentence_size = find_sentence_size(output_phonemes);


seperated_sentence_indexes = 0;
all_phonemes = '';
temp_output_phonemes = output_phonemes;
```

```
intonation_count = 1;
if length(sentence_words) > 0
for count=1:length(sentence_words)

    word = sentence_words{count};
    real_word = actual_words_normalized(count);
    %Find importance
    [word_def, word_importance] = word_type(real_word);
    %Find sentence position
    offset = 10/length(sentence_words);
    sentence_position = count*offset;
    phonemes = sentence_words{count};

    for phoneme_count=1:length(sentence_words{count})

        %Find distance from stress
        distance =
distance_from_stress(sentence_words(count),{phonemes},phonemes{phoneme_
count});
        %With all of the inputs calcualted, they are are sent to the
fuzzy
        %controller with rules to produce a crisp output
        f0_output =
f0_calculator(word_importance,sentence_size,sentence_position,distance,
rule_list);

        if isempty(f0_output) == 0
        plotting_data{intonation_count} = f0_output*pitch;
        all_phonemes{intonation_count} = phonemes{phoneme_count};
        else
        plotting_data{intonation_count} = [];
        all_phonemes{intonation_count} = phonemes{phoneme_count};
        end

        f0_data{intonation_count} = f0_output*pitch;
        intonation_count = intonation_count + 1;
        spot = strmatch(phonemes(phoneme_count),temp_output_phonemes);
        spot = spot(1);

        temp_output_phonemes(1:spot) = {'*'};
        phonemes{phoneme_count} = '/';

    end


end

else
        return;
end
f0_contour = 0;
%data =  smooth_f0_contour(f0_contour,f0_data);
data =  smooth_f0_contour(f0_data);
spaces = strmatch('%',output_phonemes,'exact');
```

```
phoneme_spot = 1:length(output_phonemes);
phoneme_spot(spaces) = '';
real_intonation_data(1:length(output_phonemes)) = {''};
real_intonation_data(spaces) = {''};
real_intonation_data(phoneme_spot) = data;

phoplayer_text =
phoneme_duration_rules(output_phonemes,real_intonation_data,seperated_s
entence_indexes,speed);

%Plotting the f0 contour
h = figure;
for count=1:length(plotting_data)-1
    point = plotting_data{count};
    next_point = plotting_data{count+1};
    if isempty(point) == 0 & isempty(next_point) == 0
    plot_one = plot([count count+1],[point
next_point],'bs:','MarkerFaceColor','c','MarkerEdgeColor','k');
    elseif isempty(point) == 0
    plot_one =
plot(count,point,'bs:','MarkerFaceColor','c','MarkerEdgeColor','k');
    end
    hold on;
end

%useless stuff
limit_data = plotting_data;

ax = gca;

if length(limit_data) > 1
set(ax,'xlim',[1 length(limit_data)]);
end
if length(limit_data) > 1 & isempty(cell2mat(limit_data))== 0
set(ax,'ylim',[min(cell2mat(limit_data))-1
max(cell2mat(limit_data))+1]);
end

set(ax,'XTick',1:length(limit_data));
set(ax,'XTickLabel',all_phonemes);
title('Output Phonemes vs. Fundamental Frequency');
ylabel('Fundamental Frequency (Hz)');



% --- Executes on button press in clear.
function clear_Callback(hObject, eventdata, handles)
% hObject    handle to clear (see GCBO)
set(handles.input_screen,'String','');



% --- Executes during object creation, after setting all properties.
function pitch_slide_CreateFcn(hObject, eventdata, handles)
% hObject    handle to pitch_slide (see GCBO)
```

```
usewhitebg = 1;
if usewhitebg
    set(hObject,'BackgroundColor',[.9 .9 .9]);
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'))
;
end


% --- Executes on slider movement.
function pitch_slide_Callback(hObject, eventdata, handles)
% hObject    handle to pitch_slide (see GCBO)
global pitch
pitch = 1.5 - get(handles.pitch_slide,'Value');

% --- Executes during object creation, after setting all properties.
function speed_slide_CreateFcn(hObject, eventdata, handles)
% hObject    handle to speed_slide (see GCBO)
usewhitebg = 1;
if usewhitebg
    set(hObject,'BackgroundColor',[.9 .9 .9]);
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'))
;
end


% --- Executes on slider movement.
function speed_slide_Callback(hObject, eventdata, handles)
% hObject    handle to speed_slide (see GCBO)
global speed
speed = 1.5 - get(handles.speed_slide,'Value');
```

```
%-------------------------------------------------------------------------
% Function: phoneme_duration_rules.m
% Purpose: Calculate segmental duration
% Description:  This program calculates the phonetic segmental
% duration.
% This done using the Klatt duration rules ("From Text to Speech: The
% MiTalk").  Each section below shows which rule corresponds to which
% section.  The program computes the segmental duration and then calls
% the MBROLA program with the segments, intonation, and phonemes
% Outputs: text(unused)
%-------------------------------------------------------------------------
function phoplayer_text =
phoneme_duration_rules(old_output_phonemes,intonation_data,seperated_se
ntence_indexes,speed)

phoplayer_text = '';
sentence_words = '';

num_of_phonemes = length(old_output_phonemes);
%Vowels an consonants
VOWELS_DIPHTHONGS = {'a' 'e' 'i' 'u' 'ah' 'aw' 'ay' 'ee' 'ie' 'oi' 'oo'
'oh' 'ow' 'ue' 'uh' 'ur'};
CONSONANTS = {'b' 'd' 'f' 'g' 'h' 'j' 'k' 'l' 'm' 'n' 'p' 'r' 's' 't'
'v' 'w' 'y' 'z' 'ch' 'ng' 'sh' 'th' 'xh' 'zh'};

%Manner of articulaion classes for consonants
NASALS = {'m' 'n' 'ng'};
LIQUIDS = {'r' 'l'};
GLIDES = {'w' 'y'};
SONORANT_CONSONANTS = {'h' 'l' 'r' 'w' 'y'};
SYLLABIC_CONSONANTS = {'n' 'l' 'r'};
VOICED_FRICATIVE = {'v' 'th' 'z' 'zh'};
VOICED_PLOSIVE = {'b' 'd' 'g'};
VOICEDLESS_PLOSIVE = {'p' 't' 'k'};

DURATION_COLUMN = 2;

[excel_file_durations excel_file_phonemes] = xlsread('Word Data\Phoneme
Duration.xls');
[dummy_data text_data] = xlsread('Word Data\Phoneme Convert List.xls');
phoneme_conversion_list(:,1) = text_data(:,1);
phoneme_conversion_list(:,2) = text_data(:,4);

%Rule #1  All Pauses at end of sentence or phrase need to be length 200
MS
phoplayer_text{1} = '_  200';

%Find the initial values for the phoneme durations
silent_count = 1;
for count=1:length(old_output_phonemes)

    spot =
strmatch(lower(old_output_phonemes(count)),excel_file_phonemes,'exact')
;
    duration_list(count) = excel_file_durations(spot,DURATION_COLUMN);
```

```
end
%duration_list

%Find the words in the sentences
sentence_words = find_sentence_words(old_output_phonemes);


%Rule #2  Vowels in the last syllable before the end needs to be
lengthened
syllables =
convert_phonemes_to_syllable(sentence_words{length(sentence_words)});
last_syllable = syllables{length(syllables)};
vowel_spot = 0;
for count=1:length(last_syllable)
    if
isempty(strmatch(lower(last_syllable(count)),VOWELS_DIPHTHONGS,'exact')
) == 0
        vowel_spot = count;
    end
end
if vowel_spot > 0
spot = strmatch(lower(last_syllable(vowel_spot)),old_output_phonemes);
spot = spot(end);
duration_list(spot) = duration_list(spot)*1.1;
if spot+1 <= length(old_output_phonemes)
if
isempty(strmatch(lower(old_output_phonemes(spot+1)),CONSONANTS,'exact')
) == 0
   duration_list(spot+1) = duration_list(spot+1)*1.4;
end;
end;
end;

%Rule #3 Vowels are shorted by .60 if not in a phrase final syllable
%A phrase final postvocalic liquid or nasal is lengthend by 1.4
temp_sentence_words = sentence_words;
temp_old_output_phonemes = old_output_phonemes;
final_word = sentence_words{length(sentence_words)};

if length(sentence_words) > 1
temp_sentence_words(length(sentence_words)) = '';
final_word_offset = length(temp_old_output_phonemes);
for word_count=1:length(temp_sentence_words)
word = temp_sentence_words{word_count};
    for letter_count=1:length(word)
        if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*.60;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end;
```

```
    end;
end;
end
if isempty(strmatch(old_output_phonemes(end),NASALS)) == 0 |
isempty(strmatch(old_output_phonemes(end),LIQUIDS)) == 0
   spot = length(old_output_phonemes);
   duration_list(spot) = duration_list(spot)*1.4;
end



%Rule #4  Vowels are shorten by .85 if not in a word final syllable
temp_sentence_words = sentence_words;
temp_old_output_phonemes = old_output_phonemes;
temp_phoneme_letter_spots = find(strcmp('%',temp_old_output_phonemes)
== 0);
temp_phoneme_spot = temp_phoneme_letter_spots(1);
temp_phoneme_space_spots =  find(strcmp('%',temp_old_output_phonemes)
== 1);
temp = find(temp_phoneme_space_spots > temp_phoneme_spot);
if isempty(temp) == 0
temp_phoneme_spot = temp_phoneme_space_spots(temp(1));
end
%Take into account 2 spaces ina row
SKIP = 1;
if SKIP == 0
for word_count=1:length(temp_sentence_words)

original_word = temp_sentence_words{word_count};
word_syllables = convert_phonemes_to_syllable(original_word);
if length(word_syllables) > 1
word_syllables(length(word_syllables)) = '';
word = '';
for count=1:length(word_syllables)
word = [word word_syllables{count}];
end

    for letter_count=1:length(word)
        if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        temp_old_output_phonemes(real_location)
        pause;
        duration_list(real_location) =
duration_list(real_location)*.85;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end;
    end;
end;
temp_old_output_phonemes(1:temp_phoneme_spot) = {'/'}
temp_phoneme_starting_point = find(strcmp('%',temp_old_output_phonemes)
== 1 & strcmp('/',temp_old_output_phonemes) == 0)
pause;
```

```
end;

end
%Rule #5 Vowels are shorten by .80 in all words with multiple syllables
%I think it's fixed!
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
syllables = convert_phonemes_to_syllable(word);
if length(syllables) > 1
    for letter_count=1:length(word)
        if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*.80;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end;
    end;
else
real_location1 =
strmatch(lower(word(end)),temp_old_output_phonemes,'exact');
if length(word) > 1
real_location2 = strmatch(lower(word(end-
1)),temp_old_output_phonemes,'exact');
else
real_location2 = real_location1 - 1;
end
real_location = real_location1(find(real_location1 - 1 ==
real_location2(1)));
temp_old_output_phonemes(1:real_location) = {'/'};
end;
end;

%Rule #6 Consonants that are not the first letter of the word are
shorten by .85
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
if length(word) > 1
    for letter_count=2:length(word)
        if
isempty(strmatch(lower(word(letter_count)),CONSONANTS,'exact')) == 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*.85;
        temp_old_output_phonemes(1:real_location) = {'/'};
        else
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
```

```
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
        end;
    end;
end;
end;

%Rule #7 Unstressed segments are compressed compared to stressed
elements
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
stresses = english_stress_rules(word);
for letter_count=1:length(word)
    if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0 & (stresses(letter_count) == 0 | stresses(letter_count) == 2)
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) = duration_list(real_location)*.9;
        temp_old_output_phonemes(1:real_location) = {'/'};
    else
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
end
end


%Rule #8 An emphasized vowel is lengthend by 1.4
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
stresses = english_stress_rules(word);
for letter_count=1:length(word)
    if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0 & stresses(letter_count) == 1
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*1.10;
        temp_old_output_phonemes(1:real_location) = {'/'};
    else
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
end
end
```

```
%Rule #9 Alter duration of vowels that are affected by postvocalic
consonants
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
for letter_count=1:length(word)
    if
(isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0 |
isempty(strmatch(lower(word(letter_count)),SONORANT_CONSONANTS,'exact')
) == 0) & letter_count + 1 <= length(word)
        %Before a voiced fricative
        if
isempty(strmatch(lower(word(letter_count+1)),VOICED_FRICATIVE,'exact'))
== 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*1.40;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end
        %Before a voiced plosive
         if
isempty(strmatch(lower(word(letter_count+1)),VOICED_PLOSIVE,'exact'))
== 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*1.20;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end
        %Before a nasal
        if
isempty(strmatch(lower(word(letter_count+1)),NASALS,'exact')) == 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*.85;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end
        %Before a voiceless plosive
        if
isempty(strmatch(lower(word(letter_count+1)),VOICEDLESS_PLOSIVE,'exact'
)) == 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) = duration_list(real_location)*.7;
        temp_old_output_phonemes(1:real_location) = {'/'};
        end

    else
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
```

```
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
end
end


%Rule #10 Alter duration of vowel-vowel combinations and consonant-
consonant combinations
%Vowel-vowel combinations
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
for letter_count=1:length(word)
    if
isempty(strmatch(lower(word(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0 & letter_count + 1 <= length(word)
    if
isempty(strmatch(lower(word(letter_count+1)),VOWELS_DIPHTHONGS,'exact')
) == 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*1.20;
        duration_list(real_location+1) =
duration_list(real_location+1)*.7;
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
    else
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
end
end
%Consonant-consonant combinations
temp_old_output_phonemes = old_output_phonemes;
for word_count=1:length(sentence_words)
word = sentence_words{word_count};
for letter_count=1:length(word)
    if isempty(strmatch(lower(word(letter_count)),CONSONANTS,'exact'))
== 0 & letter_count + 1 <= length(word)
    if
isempty(strmatch(lower(word(letter_count+1)),CONSONANTS,'exact')) == 0
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        duration_list(real_location) =
duration_list(real_location)*1.20;
        duration_list(real_location+1) =
duration_list(real_location+1)*.7;
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
    else
```

```
        real_location =
strmatch(lower(word(letter_count)),temp_old_output_phonemes,'exact');
        real_location = real_location(1);
        temp_old_output_phonemes(1:real_location) = {'/'};
    end
end
end


%Rule #11 My rule, if there are two silences in a row the reduce the
second silence
temp_old_output_phonemes = old_output_phonemes;
for phoneme_count=1:length(old_output_phonemes)
    if
isempty(strmatch(temp_old_output_phonemes(phoneme_count),'%','exact'))
== 0 & phoneme_count + 1 <= length(old_output_phonemes)
    if
isempty(strmatch(temp_old_output_phonemes(phoneme_count+1),'%','exact')
) == 0
        duration_list(phoneme_count+1) =
duration_list(phoneme_count+1)*.4;
        temp_old_output_phonemes(1:phoneme_count) = {'/'};
    end
    else
        temp_old_output_phonemes(1:phoneme_count) = {'/'};
    end
end



%Create the file *pho ouput file
for count=1:length(old_output_phonemes)
    index =
strmatch(lower(char(old_output_phonemes(count))),phoneme_conversion_lis
t(:,2),'exact');
    output_phonemes(count) = phoneme_conversion_list(index,1);
end

%adjust the speed
duration_list = duration_list*speed*1.15;

for count=1:length(old_output_phonemes)
phoplayer_text{count+1} = [char(output_phonemes(count)),'
',num2str(duration_list(count)),' ', num2str(intonation_data{count})];
end

phoplayer_text = [phoplayer_text cellstr('_  210')];

fid = fopen('output.pho','w');
for count=1:length(phoplayer_text)
fprintf(fid,'%s\n',char(phoplayer_text(count)));
end
OK = fclose(fid);

dos('mbrola us1 output.pho output.wav');
```

```
[wav_data,FS,NBITS,OPTS]= wavread('output.wav');
wavplay(wav_data,FS);
```

```
%-------------------------------------------------------------------------
% Function: BP_calculate.m
% Purpose: Weights Back Propgation Calculation
% Description: Function calculates the network output with inputs and
% weights. The for loop finds the actual output of the weights
% the maximum of the weights are supposed to be the correct output
% and phonemes
% Outputs: calculated phonemes
%-------------------------------------------------------------------------
function output =
BP_calculate(input_string,output_weights,hidden_weights,PHONEME_SET,HID
DEN_NEURONS)

alphabet_size = 27;
INPUT_SIZE = alphabet_size*7;
OUTPUT_NEURONS = 43;

NUM_CHARACTERS = length(input_string);

converted_input_string =
convert_to_numbers(input_string,NUM_CHARACTERS);
input_vector = converted_input_string(1:INPUT_SIZE);
%reshape(converted_input_string,27,NUM_CHARACTERS)
%reshape(input_vector,27,7)
character_count = 8;
count = 1;
output_string = '';
while  character_count <= NUM_CHARACTERS;
    for hide = 1:HIDDEN_NEURONS;
    hidden_layer_output(hide) = 1/(1 + exp(-
1*sum(input_vector.*hidden_weights(hide,:))));
    end

    for out = 1:OUTPUT_NEURONS;
    output_layer_output =
sum(output_weights(out,:).*hidden_layer_output);
    output(out) = 1/(1 + exp(-output_layer_output));
    end;

output_phoneme_number = find(output == max(output));
output_string{count} = char(PHONEME_SET(output_phoneme_number));
input_vector =
BP_calculate_moving_window(input_vector,converted_input_string,characte
r_count);
character_count = character_count + 1;
count = count + 1;
%input_string_count = input_string_count + 1;
end
output = output_string(5:end-3);
```

```
%-------------------------------------------------------------------------
% Function: english_stress_rules.m
% Purpose: Assigns Stress
% Description: Function places stress using the Halle and Keyer stress
% rules from the MITalk System.  The rules are just numbered.  The
% actual rules are located in the book "From Text to Speech: The MITalk
% System"
% Outputs: array of stresses for the inputted word
%-------------------------------------------------------------------------
function stresses = english_stress_rules(unstressed_word)
stresses(1:length(unstressed_word)) = 0;
stress_count = 1;
syllable_list = '';
%Vowels an consonants
VOWELS_DIPHTHONGS = {'a' 'e' 'i' 'u' 'ah' 'aw' 'ay' 'ee' 'ie' 'oi' 'oo'
'oh' 'ow' 'ue' 'uh' 'ur'};
CONSONANTS = {'b' 'd' 'f' 'g' 'h' 'j' 'k' 'l' 'm' 'n' 'p' 'r' 's' 't'
'v' 'w' 'y' 'z' 'ch' 'ng' 'sh' 'th' 'xh' 'zh' '%'};
%Short and Long Vowels
SHORT_VOWELS = {'a' 'e' 'i' 'u'};
LONG_VOWELS = {'ah' 'aw' 'ay' 'ee' 'ie' 'oi' 'oo' 'oh' 'ow' 'ue' 'uh'
'ur'};

%First find all of the vowels
syllables = convert_phonemes_to_syllable(unstressed_word);
num_of_syllables = length(syllables);
no_vowel = 0;
for count=1:length(unstressed_word)
if isempty(strmatch(lower(unstressed_word{count}),VOWELS_DIPHTHONGS))
== 1
    no_vowel = no_vowel + 1;
end
end
if no_vowel == length(unstressed_word)
    return;
end

%Rule #1 A and B
BEEN_STRESSED = 0;
last_syllable_has_short = 0;
vowel = '';
last_syllable = syllables{end};
for count=1:length(last_syllable)
if isempty(strmatch(lower(last_syllable(count)),SHORT_VOWELS,'exact'))
== 0
   last_syllable_has_short = 1;
end
end
        if num_of_syllables >= 3
        if length(syllables{end-1}) == 1 & length(syllables{end}) >= 1
& last_syllable_has_short == 1
        silly = syllables{end-2};
        for letter_count=1:length(silly)
          if
isempty(strmatch(lower(silly(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
```

```
          vowel = silly(letter_count);
          end
        end

        if isempty(vowel) == 0
        vowel_spots = strmatch(vowel,unstressed_word,'exact');
        silly_spots = 0;
        for count=1:length(silly)
            silly_spots = [silly_spots
strmatch(lower(silly(count)),unstressed_word,'exact')'];
        end

        final_vowel_location = find(vowel_spots(1) == silly_spots);
        stresses(silly_spots(final_vowel_location)) = 1;
        BEEN_STRESSED = 1;
        end
        end
        end
%Rule #1 C and D

        if num_of_syllables >= 2 & BEEN_STRESSED == 0
        if length(syllables{end}) >= 1 & last_syllable_has_short == 1
        silly = syllables{end-1};
        for letter_count=1:length(silly)
          if
isempty(strmatch(lower(silly(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
          vowel = silly(letter_count);
          end
        end

        if isempty(vowel) == 0
        vowel_spots = strmatch(vowel,unstressed_word,'exact');
        silly_spots = 0;
        for count=1:length(silly)
            silly_spots = [silly_spots
strmatch(lower(silly(count)),unstressed_word,'exact')'];
        end

        final_vowel_location = find(vowel_spots(1) == silly_spots);
        stresses(silly_spots(final_vowel_location)) = 1;
        BEEN_STRESSED = 1;
        end
        end
        end

  %Rule #2 A and B
  any_syllable_has_short = 0;
  if num_of_syllables >= 2 & BEEN_STRESSED == 0
  for syl_count=2:num_of_syllables
      syl = syllables{syl_count};
      for letter_count =1:length(syl)
        if
isempty(strmatch(lower(syl(letter_count)),SHORT_VOWELS,'exact')) == 0
        any_syllable_has_short = 1;
        short_vowel_syl = syl_count;
```

```
        break;
        end
        if any_syllable_has_short == 1
            break;
        end
  end
  end

        if num_of_syllables >= 2 & any_syllable_has_short == 1
        silly = syllables{short_vowel_syl-1};
        for letter_count=1:length(silly)
          if
isempty(strmatch(lower(silly(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
          vowel = silly(letter_count);
          end
        end

        if isempty(vowel) == 0
        vowel_spots = strmatch(vowel,unstressed_word,'exact');
        silly_spots = 0;
        for count=1:length(silly)
            silly_spots = [silly_spots
strmatch(lower(silly(count)),unstressed_word,'exact')'];
        end

        final_vowel_location = find(vowel_spots(1) == silly_spots);
        stresses(silly_spots(final_vowel_location)) = 1;
        BEEN_STRESSED = 1;
        end
        end;
        end

   %Rule #3 Place Stress on last vowel and syllable

        if num_of_syllables >= 1 & BEEN_STRESSED == 0
        silly = syllables{num_of_syllables};
        for letter_count=1:length(silly)
          if
isempty(strmatch(lower(silly(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
          vowel = silly(letter_count);
          end
        end

        if isempty(vowel) == 0
        vowel_spots = strmatch(vowel,unstressed_word,'exact');
        silly_spots = 0;
        for count=1:length(silly)
            silly_spots = [silly_spots
strmatch(lower(silly(count)),unstressed_word,'exact')'];
        end

        final_vowel_location = find(vowel_spots(1) == silly_spots);
        stresses(silly_spots(final_vowel_location)) = 1;
```

```
        %return;
        end
        end

  %Stress Exceptions
  %Rules #1-3 Place Stress on First Syllable
        if num_of_syllables >= 2
        silly = syllables{1};
        for letter_count=1:length(silly)
          if
isempty(strmatch(lower(silly(letter_count)),VOWELS_DIPHTHONGS,'exact'))
== 0
          vowel = silly(letter_count);
          end
        end

        if isempty(vowel) == 0
        vowel_spots = strmatch(vowel,unstressed_word,'exact');
        stresses(vowel_spots(1)) = 1;
        %return;
        end
        end

  %Compond Stress Rules (Retaining Rules)
  %Rules #1
  if isempty(strmatch(lower(last_syllable(end)),'ee','exact')) == 0 &
num_of_syllables >= 3 & length(find(stresses == 1)) >= 2
      stress_spots = find(stresses == 1);
      stresses(stress_spots(2)) = 2;
  end
  %Rule #2 and Rule #3 Retain 1-stress vowel if it is followed by a
string of syllable
  %without primamry stresses.  If only one syllable or stress skip
  if num_of_syllables >= 2 & length(find(stresses == 1)) >= 2
  temp_unstressed_word = unstressed_word;
  temp_unstressed_word(1:length(syllables{1})) = {'/'};
  stress_spots = find(stresses == 1);
  second_syllable = syllables{2};
  for count=1:length(second_syllable)
  spots = strmatch(lower(second_syllable(count)),temp_unstressed_word);
  second_syllable_spots(count) = spots(1);
  end
  second_stress_spot = stress_spots(2);
  if isempty(find(second_stress_spot == second_syllable_spots)) == 0
      stresses(stress_spots(1)) = 2;
  else
      stresses(stress_spots(2)) = 2;
  end
  end

%Strong first syllable rules
%Rule #1 assign 2-stress to the first vowel of the word if it is long
if num_of_syllables >= 2 & length(find(stresses == 1)) >= 2
for count=1:length(unstressed_word)
  if isempty(strmatch(lower(unstressed_word(count)),VOWELS_DIPHTHONGS))
== 0
```

```
      if isempty(strmatch(lower(unstressed_word(count)),LONG_VOWELS))
== 0
      stresses(count) = 2;
      end
  break;
  end
end
end

%Rule #1 assign 2-stress to the first vowel of the word if it is
followed by two syllables
if num_of_syllables >= 2 & length(find(stresses == 1)) >= 2
for count=1:length(unstressed_word)
  if isempty(strmatch(lower(unstressed_word(count)),VOWELS_DIPHTHONGS))
== 0 & count+2 <=length(unstressed_word)
      if isempty(strmatch(lower(unstressed_word(count+1)),CONSONANTS))
== 0 & isempty(strmatch(lower(unstressed_word(count+2)),CONSONANTS)) ==
0
      stresses(count) = 2;
      end
  break;
  end
end
end
```

```
%-------------------------------------------------------------------------
% Function: convert_phonemes_to_syllable.m
% Purpose: Syllable Parser
% Description: This function parses a string into syllables.
% The output to this function is the syllable arrainged into a cell
% string
% Outputs: syllables (cell string)
%-------------------------------------------------------------------------
function syllable = convert_phonemes_to_syllable(word)

syllable = '';
WORD_CONSONANT_LOCATE_ARRAY = 0;
WORD_VOWEL_LOCATE_ARRAY = 0;
word = lower(word);
VOWELS = {'a' 'e' 'i' 'u'};
CONSONANTS = {'%' 'b' 'd' 'f' 'g' 'h' 'j' 'k' 'l' 'm' 'n' 'p' 'r' 's'
't' 'v' 'w' 'y' 'z' 'ch' 'ng' 'sh' 'th' 'xh' 'zh'};
DIPHTHONGS = {'ah' 'aw' 'ay' 'ee' 'ie' 'oi' 'oo' 'oh' 'ow' 'ue' 'uh'
'ur'};
SHORT_VOWELS = {'a' 'e' 'i' 'u'};

WORD_BEGINS_WITH_VOWEL = 0;
INITIAL_VOWEL_IS_SHORT = 0;


%Count the number of vowels and diphthongs in the word
vowel_count = 0;
syllable_count = 0;
word_length = length(word);

for count=1:word_length
    if isempty(strmatch(lower(word(count)),VOWELS,'exact')) == 0
        vowel_count = vowel_count + 1;
    end
end



for count=1:word_length
    if isempty(strmatch(lower(word(count)),DIPHTHONGS,'exact')) == 0
        vowel_count = vowel_count + 1;
    end
end

%Number of syllables equal the number of vowels and dipthongs
syllable_count = vowel_count;
if syllable_count == 1 | syllable_count == 0;
   syllable{1} = word;
   return;
end


%Check to see if the word begins with a vowel
WORD_BEGINS_WITH_VOWEL =
isempty(strmatch(word(1),CONSONANTS,'extact'));
```

```
%Find all and locate of the consonants in the word
word_c_count = 1;
for count=1:length(word)
    CON = strmatch(lower(word(count)),CONSONANTS,'extact');
    if isempty(CON) == 0
        WORD_CONSONANT_LOCATE_ARRAY(word_c_count) = count;
        word_c_count = word_c_count + 1;
    end
end

%Find and locate all of the vowels in the word
word_v_count = 1;
for count=1:length(word)
    VOW = strmatch(lower(word(count)),VOWELS,'extact');
    DIP = strmatch(lower(word(count)),DIPHTHONGS,'extact');
    if isempty(VOW) == 0 | isempty(DIP) == 0
        WORD_VOWEL_LOCATE_ARRAY(word_v_count) = count;
        word_v_count = word_v_count + 1;
    end
end

%Check to see if there are two vowels in a row and subtract from the
total count of syllables
syllable_count = syllable_count -
length(find(diff(WORD_VOWEL_LOCATE_ARRAY) == 1));
if syllable_count == 1 | syllable_count == 0;
   syllable{1} = word;
   return;
end


%Splitting two middle consonants
MIDDLE_C = find(diff(WORD_CONSONANT_LOCATE_ARRAY) == 1);
if isempty(MIDDLE_C) == 0 & syllable_count == 2 & length(MIDDLE_C) == 1
    syllable{1} = word(1:WORD_CONSONANT_LOCATE_ARRAY(MIDDLE_C));
    syllable{2} = word(WORD_CONSONANT_LOCATE_ARRAY(MIDDLE_C+1):end);
return;
end


%Test to see if the first vowel is short or long
if syllable_count == 2
    if isempty(strmatch(word(1),SHORT_VOWELS,'extact')) == 0 |
isempty(strmatch(word(2),SHORT_VOWELS,'extact')) == 0
    INITIAL_VOWEL_IS_SHORT = 1;
    end
end
%Splitting before single middle consonant in a 2 syllable word
%Intital vowel is not short
if INITIAL_VOWEL_IS_SHORT == 0 & syllable_count == 2
if isempty(MIDDLE_C) == 1 & WORD_BEGINS_WITH_VOWEL == 0
    syllable{1} = word(1:WORD_CONSONANT_LOCATE_ARRAY(2)-1);
    syllable{2} = word(WORD_CONSONANT_LOCATE_ARRAY(2):end);
return;
end;
if isempty(MIDDLE_C) == 1 & WORD_BEGINS_WITH_VOWEL == 1
```

```
    syllable{1} = word(1);
    syllable{2} = word(2:end);
return;
end;
end;
%Splitting before single middle consonant in a 2 syllable word
%Intital vowel is short
if INITIAL_VOWEL_IS_SHORT == 1 & syllable_count == 2
if isempty(MIDDLE_C) == 1 & WORD_BEGINS_WITH_VOWEL == 0
    syllable{1} = word(1:WORD_CONSONANT_LOCATE_ARRAY(2));
    syllable{2} = word(WORD_CONSONANT_LOCATE_ARRAY(2)+1:end);
return;
end;
if isempty(MIDDLE_C) == 1 & WORD_BEGINS_WITH_VOWEL == 1
    syllable{1} = word(1:2);
    syllable{2} = word(3:end);
return;
end;
end;

%Seperate word syllables that have more than 2 syllables or do not
follow
%the above criteria
if syllable_count >= 2

if WORD_BEGINS_WITH_VOWEL == 0
    %If the word begins with consonant syllable is 3 letters long
initally
    syllable_starting_point = 1;
    syllable_ending_point = 3;
else
    %If the word begins with vowel syllable is 2 letters long
    syllable_starting_point = 1;
    syllable_ending_point = 2;
end

num_of_syllables = syllable_count;
for syl_cell_count=1:num_of_syllables
     %Initially store the syllable from the starting and ending letters
     %syllable_starting_point
     %syllable_ending_point
%     word(syllable_starting_point:syllable_ending_point)
     if syllable_ending_point <= length(word)
     syllable{syl_cell_count} =
word(syllable_starting_point:syllable_ending_point);
     end
         %Find the location of the next two letters
     next_letter = syllable_ending_point + 1;
     next_next_letter = syllable_ending_point + 2;
     next_next_next_letter = syllable_ending_point + 3;
     if syl_cell_count == 1
     %For the first iteration, check to see if the next letter at the
end of the 3 letter long syllable is a vowel.
     %If so, then make the first syllable only 2 letters long and set
the location for the next syllable
     %If not then set the location for the next syllable; if the word
```

```
     %begins with a vowel then automatically goto this step
     if isempty(find(next_letter == WORD_VOWEL_LOCATE_ARRAY)) == 0 &
WORD_BEGINS_WITH_VOWEL == 0
         syllable = '';
         syllable{syl_cell_count} =
word(syllable_starting_point:syllable_ending_point-1);
         syllable_starting_point = syllable_ending_point;
         syllable_ending_point = syllable_ending_point + 1;

     else
        syllable_starting_point = syllable_ending_point + 1;
        syllable_ending_point = syllable_ending_point + 2;
     end
     end

     if syl_cell_count > 1
     %For the rest of rest of the iterations, check to see if the
second
     %letter after the lat letter in the 2 letter syllable is a vowel
     %If so, then update the syllable location for the next syllable
     %If the letter is a consonant then make the syllable 3 letters
long
     %and update the syllable location for the next syllable
     if isempty(find(next_next_letter == WORD_VOWEL_LOCATE_ARRAY)) == 0
        syllable_starting_point = syllable_ending_point + 1;
        syllable_ending_point = syllable_ending_point + 2;
     else
        if syllable_starting_point < length(word) &
syllable_ending_point+1 <= length(word)
        syllable{syl_cell_count} =
word(syllable_starting_point:syllable_ending_point+1);
        syllable_starting_point = syllable_ending_point + 2;
        syllable_ending_point = syllable_ending_point + 3;

        if syllable_ending_point > length(word)
        syllable_starting_point = syllable_starting_point - 1;
        syllable_ending_point = syllable_ending_point - 1;
        syllable{syl_cell_count} =
word(syllable_starting_point:syllable_ending_point-1);
        end

        end
     end
     end

%for loop
end

output_length = 0;
syllable_start = '';
for count=1:length(syllable)
output_length = output_length + length(syllable{count});
syllable_start{count} =  output_length - length(syllable{count});
end
```

```
if output_length < word_length
temp_word = word(syllable_start{end}+1:word_length);
syllable{end} = temp_word;
end


%syllable_count > 2
end


%END OF PROGRAM
```

```
%-------------------------------------------------------------------------
% Function: orthographic_converter.m
% Purpose: Orthographic Converter
% Description: This program handle numbers and abbreviations for the
% TTS program.  It converts these orthgraphic words into regular text
% Outputs: words
%-------------------------------------------------------------------------
function actual_words = orthographic_converter(actual_words);

[abbreviations regular_words] = textread('Word
Data\Abbreviations.txt','%s%s');

%Abbreviations Conversion
for count=1:length(actual_words)
    word = word_normalizer(actual_words{count});
    spot = strmatch(word,abbreviations,'exact');
    if isempty(spot) == 0
    actual_words{count} = {regular_words{spot}};
    end
end


%Numeric Conversion
[numbers numeric_words] = textread('Word Data\Numbers.txt','%s%s');

%temp_actual_words = actual_words;
temp_actual_words = '';
%temp_actual_words{1:end}
count = 1;
temp_count = 1;

while count <= length(actual_words)

    word = char(word_normalizer(actual_words{count}));
    %if isempty(str2num(word)) == 0
    if sum(isletter(word)) == 0
    %if isnumeric(word) == 1

    numbered_word = floor(str2num(word));

    if numbered_word < 0
    temp_actual_words{temp_count} = 'negative';
    temp_count = temp_count + 1;
    numbered_word = floor(abs(str2num(word)));
    end

    if numbered_word == 0
    temp_actual_words{temp_count} = 'zero';
    end
     % For words between 1 and 10
    if numbered_word >= 1 & numbered_word <= 10
        worded_number = num2str(numbered_word);
        temp_actual_words{temp_count} =
numeric_words{strmatch(worded_number,numbers,'exact')};
    end
```

```
    % For words between 10 and 100
    if numbered_word > 10 & numbered_word < 100
        worded_number = num2str(numbered_word);
        spot = strmatch(worded_number,numbers,'exact');
            if isempty(spot) == 0
            temp_actual_words{temp_count} = numeric_words{spot};
            else
            temp_actual_words{temp_count} =
numeric_words{strmatch([worded_number(1),'0'],numbers,'exact')};
            temp_actual_words{temp_count+1} =
numeric_words{strmatch(worded_number(2),numbers,'exact')};
            temp_count = temp_count + 1;
            end
    %End for words between 10 and 100
    end


    % For words between 100 and 1000
    if numbered_word >= 100 & numbered_word < 1000
        worded_number = num2str(numbered_word);
        % hundreds place
        temp_actual_words{temp_count} =
numeric_words{strmatch(worded_number(1),numbers,'exact')};
        temp_actual_words{temp_count+1} = 'hundred';
        temp_count = temp_count + 1;

        %tens and ones place
        tens_place_number = str2num(worded_number(2:3));

            % For words between 1 and 10
            if tens_place_number >= 1 & tens_place_number <= 10
            worded_number = num2str(tens_place_number);
            temp_actual_words{temp_count+1} =
numeric_words{strmatch(worded_number,numbers,'exact')};
            temp_count = temp_count + 1;
            end

            % For words between 10 and 100
            if tens_place_number > 10 & tens_place_number < 100
            worded_number = num2str(tens_place_number);
            spot = strmatch(worded_number,numbers,'exact');
            if isempty(spot) == 0
            temp_actual_words{temp_count+1} = numeric_words{spot};
            temp_count = temp_count + 1;
            else
            temp_actual_words{temp_count+1} =
numeric_words{strmatch([worded_number(1),'0'],numbers,'exact')};
            temp_actual_words{temp_count+2} =
numeric_words{strmatch(worded_number(2),numbers,'exact')};
            temp_count = temp_count + 2;
            end
            end

    %end for words between 100 and 1000
```

```
    end

    %between 1000 and 1,000,0000
    if numbered_word > 1000 & numbered_word < 1000000
        worded_number = num2str(numbered_word);
        % hundreds thousand place
        temp_actual_words{temp_count} =
numeric_words{strmatch(worded_number(1),numbers,'exact')};
        temp_actual_words{temp_count+1} = 'hundred';
        temp_count = temp_count + 1;

        %tens and ones thousand place
        tens_place_number = str2num(worded_number(2:3));

            % For words between 1 and 10
            if tens_place_number >= 1 & tens_place_number <= 10
            worded_number = num2str(tens_place_number);
            temp_actual_words{temp_count+1} =
numeric_words{strmatch(worded_number,numbers,'exact')};
            temp_count = temp_count + 1;
            end

            % For words between 10 and 100
            if tens_place_number > 10 & tens_place_number < 100
            worded_number = num2str(tens_place_number);
            spot = strmatch(worded_number,numbers,'exact');
            if isempty(spot) == 0
            temp_actual_words{temp_count+1} = numeric_words{spot};
            temp_count = temp_count + 1;
            else
            temp_actual_words{temp_count+1} =
numeric_words{strmatch([worded_number(1),'0'],numbers,'exact')};
            temp_actual_words{temp_count+2} =
numeric_words{strmatch(worded_number(2),numbers,'exact')};
            temp_count = temp_count + 2;
            end
            end

            temp_actual_words{temp_count+1} = 'thousand';


    %end for words between 100 and 1000
    end



    %Decimal points
    if abs(rem(str2num(word),1)) > 0
        decimal_word = num2str(abs(rem(str2num(word),1)));
        temp_actual_words{temp_count+1} = 'point';
        temp_count = temp_count + 1;
        for decimal_count = 3:length(decimal_word)
        temp_count = temp_count + 1;
        temp_actual_words{temp_count} =
numeric_words{strmatch(decimal_word(decimal_count),numbers,'exact')};
```

```
        end
    %End to Decimal points
    end


    %Else to the first if statement in the while loop
    else

        temp_actual_words{temp_count} =  word;
    %End of the first if statement in the while loop
    end



count = count + 1;
temp_count = temp_count + 1;
%End of the while loop
end

actual_words = temp_actual_words;
```

```
%-------------------------------------------------------------------------
% Function: f0_calculator.m
% Purpose: Fuzzy Inference Computation
% Description: Function calculates the f0 output.  The is basically the
% fuzzy interference system with calls to the MF-Calculator. Using the
% MF decsriptior (global) and input variables the function calculates
% the crisp f0 output for a given phoneme
% Outputs: f0 (fundamental frequency) output
%-------------------------------------------------------------------------
function f0_output =
f0_calculator(variable1,variable2,variable3,variable4,rules)

global MF_descriptors
global test_axes

variables(1) = variable1;
variables(2) = variable2;
variables(3) = variable3;
variables(4) = variable4;

GRAPH = 0;

%Get each rule in from the list
for rule_count=1:length(rules)

%First get the numberes that represent the linguist variables of the
rule
rule_numbers = rules{rule_count};

MF_count = 1;
%Then for each linguistic variable excpet the output, get the MF for
that
%input, the calculate the output that variable produces with that input
%If the rule number is 0 then that means that input has no bearing on
the
%rule.
for count=1:length(rule_numbers)-1
if rule_numbers(count) > 0
membership_functions = MF_descriptors{count};
MF(MF_count) =
MF_calculator(variables(count),membership_functions(rule_numbers(count)
,:));
MF_count = MF_count + 1;
end
end

rule_output_numbers(rule_count) = min(MF);

end

output_MF = MF_descriptors{5};

fired_rules = find(rule_output_numbers > 0);

if isempty(fired_rules) ==  0
```

```
         for fire_count=1:length(fired_rules)
         firing_strength =
rule_output_numbers(fired_rules(fire_count));
         rule_spot = rules{fired_rules(fire_count)};
         triangles = output_MF;
         output_tri = triangles(rule_spot(5),:);
         trapezoid =
create_MF_trapezoid(firing_strength,output_tri(1),output_tri(2),output_
tri(3),12);
         all_trapezoids(fire_count,:) = trapezoid;
        end


 else
        f0_output = [];
        return;
 end

 if length(fired_rules) > 1
 trape_max = max(all_trapezoids);
 centroid = sum((0:.1:12).*(trape_max))/sum(trape_max);
 f0_output =  130 + centroid*9;
else
 trape_max = all_trapezoids;
 centroid = sum((0:.1:12).*(trape_max))/sum(trape_max);
 f0_output =  130 + centroid*9;
end
```

```
%-------------------------------------------------------------------------
% Function: find_acronyms.m
% Purpose: Find the Acronyms
% Description: This program handle acronyms. It converts these acronyms
% words into regular text
% Outputs: ACRONYMS converted into words
%-------------------------------------------------------------------------
function actual_words = find_acronyms(actual_words)

[capital_letters column1 column2] = textread('Word Data\Capital Letter
to Sound List.txt','%s%s%s');
word_count = 1;
temp_word_count =  1;
FOUND_ALL_CAPS = 0;
number_of_words = length(actual_words);
temp_actual_words = actual_words;

while word_count <= number_of_words

    word = actual_words{word_count};
    upper_count=0;
for another_count=1:length(word)
    if isempty(strmatch(word{another_count},capital_letters)) == 0
    upper_count = upper_count + 1;
    end
end

if upper_count==length(word)
    for capital_count=1:length(word)
    letter_spot = strmatch(word(capital_count),capital_letters);
    temp_actual_words{(temp_word_count + capital_count)-1} =
column1(letter_spot);
    %temp_word_count = temp_word_count + 1;
        if letter_spot == 23
        temp_actual_words{(temp_word_count + capital_count)} =
column2(letter_spot);
        %word_count = word_count + 1;
        end
    end

temp_word_count = temp_word_count + length(word) - 1;
%end to if statement
else
temp_actual_words{temp_word_count} = actual_words{word_count};
end
%number_of_words = length(actual_words);
%actual_words{word_count+1}
word_count = word_count + 1;
temp_word_count = temp_word_count + 1;
end

actual_words = temp_actual_words;
```

```
%-------------------------------------------------------------------------
% Function: word_type.m
% Purpose: Find the type of word
% Description: This program finds the word importance of a given word.
% The input is the word and the program first checks to see if it is a
% function or content word. Then it assigns a value between 1 and 10
% based on the word size.
% words into regular text
% Outputs: interger (1-10) and word type (function or content)
%-------------------------------------------------------------------------
function [word_type, word_importance] = word_type(first_word)
word_type = 'CONTENT';

ARTICLES = {'a' 'an' 'the' 'some'};
CONJUNCTIONS = {'and' 'but' 'or' 'so' 'because'  'although' 'nor'
'neither' 'either'};
OTHER_FUNCTIONS = { 'about' 'across' 'against' 'am' 'among' 'any'
'anybody' ...
        'anyone' 'anything' 'are' 'around' 'as' 'at' 'be' 'been'
'before' 'behind' 'below'...
        'beneath' 'beside' 'between' 'beyond' 'by' 'can' 'could' 'did'
'do' 'does' 'down' 'during' ...
        'each' 'ever' 'every' 'everybody' 'everyone' 'everything' 'for'
'from' 'going' 'had' ...
        'has' 'have' 'he' 'her' 'hers' 'herself' 'him' 'himself' 'his'
'however' 'I' 'if' 'in' ...
        'into' 'is' 'it' 'its' 'itself' 'like' 'may' 'me' 'might'
'mine' 'my' 'myself' 'never'...
        'no' 'nobody' 'noone' 'not' 'nothing' 'off' 'on' 'onto' 'or'
'ought' 'our' 'ours' 'ourselves' ...
        'over' 'shall' 'she' 'should' 'since' 'so' 'somebody' 'someone'
'something' 'than' 'that' 'the' ...
        'their' 'them' 'themselves' 'then' 'therefore' 'therfore'
'these' 'they' 'this' 'those' 'though' 'through' ...
        'to' 'under' 'unless' 'until' 'up' 'us' 'was' 'we' 'were'
'whatever' 'whenever' ...
        'wherever' 'whether' 'which' 'while' 'whose'  'will' 'with'
'without' 'would' 'you' ...
        'your' 'yours' 'yourself'};

%empty_cells = strmatch('',prep_size2,'exact');
%double_word_start = empty_cells(end) + 1;

%'how' 'what' 'when' 'where' 'who' 'whom' 'whose' 'why'

%single_prepositions = prep_size1(1:double_word_start-1);
%double_prepositions = [prep_size1(double_word_start:end)
prep_size2(double_word_start:end)];
%Check to see if the word is an article
for count=1:length(ARTICLES)
    if isempty(strmatch(lower(first_word),ARTICLES(count),'exact')) ==
0
       word_type = 'FUNCTION';
       word_importance = length(first_word{1}) - 1;
       if word_importance > 1.5
          word_importance = 1.5;
```

```
        end
        return;
    end
end


%check for conjunctions
for count=1:length(CONJUNCTIONS)
    if isempty(strmatch(lower(first_word),CONJUNCTIONS(count),'exact'))
== 0
        word_type = 'FUNCTION';
        word_importance = length(first_word{1});
        if word_importance > 2.5
           word_importance = 2.5;
        end

        return;
    end
end

%Check to see if word is a other function words
for count=1:length(OTHER_FUNCTIONS)
    if
isempty(strmatch(lower(first_word),OTHER_FUNCTIONS(count),'exact')) ==
0
        word_type = 'FUNCTION';
        word_importance = length(first_word{1})+.15;
        if word_importance > 2.75
           word_importance = 2.75;
        end
        return;
    end
end


word_importance = length(first_word{1}) + 1.75;
if word_importance > 10;
    word_importance = 10;
end
```

```
%-------------------------------------------------------------------------
% Function: distance_from_stress.m
% Purpose: Find the word distance from stress
% Description: This program finds the word distance form stress and
% then it assigns a value between 1 and 10
% Outputs: interger (1-10)
%-------------------------------------------------------------------------
function stress_distance =
distance_from_stress(sentence_word,temp_phonemes,phoneme)

VOWELS = {'a' 'e' 'i' 'u'};
CONSONANTS = {'b' 'd' 'f' 'g' 'h' 'j' 'k' 'l' 'm' 'n' 'p' 'r' 's' 't'
'v' 'w' 'y' 'z' 'ch' 'ng' 'sh' 'th' 'xh' 'zh'};
DIPHTHONGS = {'ah' 'aw' 'ay' 'ee' 'ie' 'oi' 'oo' 'oh' 'ow' 'ue' 'uh'
'ur'};

phoneme_spot = strmatch(phoneme,temp_phonemes{1},'exact');
phoneme_spot = phoneme_spot(1);
phoneme_word_stresses = english_stress_rules(sentence_word{1});
primary_stresses = find(phoneme_word_stresses == 1);
if isempty(primary_stresses) == 1
   stress_distance = 10;
   return;
end

if primary_stresses(1) == phoneme_spot
    stress_distance = 0;
    return;
end

if length(sentence_word) == 1
   offset = 10/length(sentence_word{1});
   stress_distance = (abs(primary_stresses(1) -
phoneme_spot)+1)*offset;
   return;
end
```

```
%-------------------------------------------------------------------------
% Function: distance_from_stress.m
% Purpose: Calculates position in sentence
% Description: Calculates position in sentence and then it assigns a
% value between 1 and 10
% Outputs: interger (1-10)
%-------------------------------------------------------------------------

function sentence_position =
find_sentence_position(output_phonemes,phoneme)

sentence_size = length(output_phonemes);
offset = 10/sentence_size;

found_letter = strmatch(phoneme,output_phonemes,'exact');
found_letter = found_letter(1);

sentence_position = found_letter*offset;
```

```
%-------------------------------------------------------------------------
% Function: distance_from_stress.m
% Purpose: Calculates sentence size
% Description: Calculates sentence size and then it assigns a
% value between 1 and 10
% Outputs: interger (1-10)
%-------------------------------------------------------------------------

function sentence_size = find_sentence_size(output_phonemes);

sentence_words = find_sentence_words(output_phonemes);
syllable_count = 0;

for count=1:length(sentence_words)
    word = sentence_words{count};
    syllable = convert_phonemes_to_syllable(word);
    syllable_count =  syllable_count + length(syllable);
end
off_set = .5;
sentence_size = syllable_count*off_set;
if sentence_size > 10
    sentence_size = 10;
end
```

```
%-------------------------------------------------------------------------
% Function: distance_from_stress.m
% Purpose: Finds the words in sentence
% Description: Finds the words in sentence based on the spaces.
% Outputs: cell string of sentence words
%-------------------------------------------------------------------------
function sentence_words =
find_sentence_words(old_output_phonemes,COMMAS);

spaces = strmatch('%',old_output_phonemes,'exact');
if isempty(spaces) == 0;
temp_phonemes = old_output_phonemes;
spaceless_word = old_output_phonemes;
spaceless_word(spaces) = '';
for count=1:length(spaceless_word);
spot = strmatch(spaceless_word(count),temp_phonemes);
non_space_letter_indices(count) = spot(1);
temp_phonemes(1:spot(1)) = {'\'};
end

word_spot = 1;
word_count = 1;
word{word_spot} =
char(old_output_phonemes(non_space_letter_indices(1)));
word_spot = word_spot + 1;
differences = diff(non_space_letter_indices);


for count=1:length(differences)
if differences(count) == 1;
word{word_spot} =
char(old_output_phonemes(non_space_letter_indices(count+1)));
word_spot = word_spot + 1;
else
sentence_words{word_count} = word;
word_count = word_count + 1;
word = '';
word_spot = 1;
word{word_spot} =
char(old_output_phonemes(non_space_letter_indices(count+1)));
word_spot = word_spot + 1;
end
end

sentence_words{word_count} = word;
end

if isempty(spaces) == 1;
sentence_words{1} = old_output_phonemes;
end
```