ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

**MSc THESIS**

**Yoldaş ERDOĞAN**

# TURKISH TEXT TO SPEECH USING CHILDREN'S VOICES SYLLABLES

**DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING**

**ADANA, 2019**

**ÇUKUROVA UNIVERSITY**
**INSTITUTE OF NATURAL AND APPLIED SCIENCES**

# TURKISH TEXT TO SPEECH USING CHILDREN'S VOICES SYLLABLES

**Yoldaş ERDOĞAN**

**MSc THESIS**

**DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING**

We certify that the thesis titled above was reviewed and approved for the award of degree of the Master of Science by the board of jury on 19/08/2019.

| | | |
|---|---|---|
| ………………………………… | ……………………………. | …………………………........... |
| Assoc. Prof. Dr. Zekeriya TÜFEKÇİ | Assoc. Prof. Dr. Sami ARICA | Assoc. Prof. Dr. Serdar YILDIRIM |
| SUPERVISOR | MEMBER | MEMBER |

This MSc Thesis is written at the Department of Electrical and Electronics Engineering of Institute of Natural and Applied Sciences of Çukurova University.

**Registration Number**:

**Prof. Dr. Mustafa GÖK**
**Director**
**Institute of Natural and Applied Sciences**

## ABSTRACT

**MSc THESIS**

**TURKISH TEXT TO SPEECH USING CHILDREN'S VOICES SYLLABLES**

**Yoldaş ERDOĞAN**

**ÇUKUROVA UNIVERSITY**
**INSTITUTE OF NATURAL AND APPLIED SCIENCES**
**DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING**

Supervisor : Assoc. Prof. Dr. Zekeriya TÜFEKÇİ
Year: 2019, Pages: 135
Jury : Assoc. Prof. Dr. Zekeriya TÜFEKÇİ
: Assoc. Prof. Dr. Sami ARICA
: Assoc. Prof. Dr. Serdar YILDIRIM

Text to speech (TTS) shortly means to convert a written text into audio signals electronically. This written text may be a text document, electronic book, or a web page. An ideal TTS system is expected to be able to process every readable text in the quality of natural human voice. In our country, text to speech studies mostly focus on the production of adult male and female voices. In this thesis, an audio database consisting of children's voices was designed so the synthesized sound is aimed to be children's voices.

In voice synthesis studies, it is seen that the closest sound to naturalness was provided by concatenative voice synthesis methods. Within the scope of this thesis, a TTS system that is based on additive synthesis technique which uses binary syllable as the length of voice unit is implemented. In general, conversion of text to audio signal process consists of two main parts. In the first part, the text to be synthesized is normalized according to language rules and is divided into syllables. A hyphenation algorithm is developed for the designed system and the entered text was separated into syllables. In the second part, audio syllable signals are processed and merged so that the speech synthesizing process is performed. Although there are different techniques in processing the audio signals, they are extended and shortened based on the Synchronous Overlap and Add (SOLA) method in this thesis.

The system generates syllables from the text information it receives as an input. It makes triple syllables to be produced from double syllables. Then, by using the audio files belonging to these syllables, syllables are taken from the recorded files and began to be merged. At this stage, rules determined according to the types of sounds are applied at the junction points of syllables and naturalness is tried to be created similar to the waveforms in real sound files. This naturalness has been tried to be provided by extending and shortening the beginning or end of syllables where necessary.

Although the system uses simple techniques, the selected additive method is very suitable for the structure of Turkish and so produces efficient results.

**Keywords:** Turkish Text to Speech, Additive Synthesis, Child Voice, Syllabification , TTS, Voice Extension, WAV, SOLA

## ÖZ

**YÜKSEK LİSANS TEZİ**

**ÇOCUK SES HECELERİ KULLANARAK TÜRKÇE METİNDEN KONUŞMA SESLENDİRME**

**Yoldaş ERDOĞAN**

**ÇUKUROVA ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**
**ELEKTRİK-ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI**

Danışman : Doç. Dr. Zekeriya TÜFEKÇİ
Yıl: 2019, Sayfalar: 135
Jüri : Doç. Dr. Zekeriya TÜFEKÇİ
: Doç. Dr. Sami ARICA
: Doç. Dr. Serdar YILDIRIM

Metinden Konuşma Sentezleme (MKS) kısaca yazılı haldeki bir metnin elektronik ortama aktarılarak ses sinyallerine dönüştürülmesi demektir. Bu yazılı metin bir belge veya elektronik kitap da olabilir, bir web sayfası da olabilir. İdeal bir TTS sisteminden insanın okuyabildiği her metni doğal insan sesi gibi işleyebilmesi beklenir. Ülkemizde metinden konuşma sentezleme çalışmaları daha çok yetişkin kadın ve erkek seslerinin üretilmesine yoğunlaşmıştır. Bu tezde ise çocuk seslerinden oluşan bir ses veritabanı tasarlanmış ve sentezlenecek sesin çocuk sesi olması hedeflenmiştir.

Ses sentezleme çalışmalarında doğallığa en yakın sesin, eklemeli (concatenative) ses sentezleme yöntemleri ile sağlandığı görülmüştür. Bu tez kapsamında ses verisi olarak ikili heceyi kullanan ve eklemeli sentezleme yöntemine dayanan bir metin seslendirme sistemi gerçeklenmiştir. Metinden konuşma sinyali oluşturma genel olarak iki ana bölümden oluşmaktadır. Birinci bölümde sentezlenecek metin, dil kurallarına uygun olarak normalize edilmekte ve hecelerine ayrılmaktadır. Tasarlanan system için bir heceleme algoritması geliştirilmiş ve girilen metnin hecelerine ayrılması sağlanmıştır. İkinci bölümde ise ses hece sinyalleri işlenerek bir araya getirilmekte ve konuşma sentezleme işlemi gerçekleştirilmektedir. Ses sinyallerinin işlenmesinde farklı teknikler bulunmakla beraber bu tez çalışmasında SOLA(Synchronous Overlap and Add) yöntemi temel alınarak ses sinyalleri uzatılmakta ve kısaltılmaktadır.

Sistem, girişte aldığı metin bilgisinden heceleri oluşturur. Üçlü heceleri ikili hecelerden üretilecek şekle getirir. Daha sonra bu hecelere ait ses dosyalarını kullanarak ikili veye tekli heceleri kayıtlı oldukları dosyalardan alır ve belirli algoritmalar dahilinde birleştirir. Bu aşamada hecelerin birleştiği yerlerde seslerin türlerine göre belirlenen kurallar uygulanır ve gerçek ses dosyalarındaki doğallık elde edilmeye çalışılır. Bu doğallık gerekli yerlerde hecelerin başında ya da sonunda uzatma ve kısaltma yapılarak sağlanmaya çalışılmıştır.

Sistem basit teknikler kullanıyor olmasına rağmen, seçilen eklemeli method Türkçe'nin yapısına çok uygun olduğu için verimli sonuçlar üretmektedir.

**Anahtar Kelimeler:** Türkçe Metin Seslendirme, Eklemeli Sentezleme, Çoçuk Sesi, Heceleme, TTS, Ses Uzatma, WAV, SOLA

## ACKNOWLEDGEMENTS

I would like to express my gratitude to my mother Hatun Erdoğan and my father Aleddin Erdoğan for enabling me to grow up in the difficult life conditions of the past. I would also like to thank my dearest sister Ceylan, her husband Umut and my nephew Asmê Hayal for supporting me at every stage of my life.

My lovely family members; I would like to thank my dear wife Ergül for her patience and for enduring me in this challenging road, my son Rênas Merdan, my daughters Zelal, Zerya and Roza Dilba for allowing me to work during this thesis. I love you very, very much.

I would also like to thank my thesis advisor Zekeriya Tüfekçi and valuable jury members as well as my colleagues Esin, Murat and Erkan for their moral support.

To the beautiful tomorrows where everyone lives peacefully, without missing love, respect, happiness and hope…

**TABLE OF CONTENTS** **PAGE**

**LIST OF TABLES** **PAGE**

**LIST OF FIGURES** **PAGE**

# 1. INTRODUCTION

## 1.1. General Information

Text to Speech (TTS) is converting text into spoken word by a computer. Nowadays, TTS applications are used in many immobile and mobile devices for different purposes such as making the interaction with the user more humanitarian in multimedia tools or facilitating the life of some people, especially the visually impaired ones. Efforts to automatically produce sounds similar to human voices are one of today's popular topics.

Speech synthesis is the process of converting a text into audio signals by a computer and so artificially producing human speech. A computer system used for this purpose is called a “speech synthesizer” and the process can be implemented in computer software or hardware. Linking the recorded speech segments stored in a database to each other is a commonly used speech synthesizing technique

The quality of a speech synthesizer is assessed by the similarity and intelligibility of the synthesized speech to the human voice. There are lots of application areas of speech synthesizing systems nowadays and these areas are increasing day by day.

In this study, a Turkish TTS system is designed and implemented. There are three groups in TTS systems: formant synthesizers, additive synthesizers and discourse synthesizers. The developed synthesizer is in the second group. The additive synthesizers synthesize the audio tracks recorded by a speaker by reassembling them afterwards. For this purpose, first of all, the text that will be converted into speech is divided into words. The words that are separated from the space character are then separated into their diphones (binary phonemes). In this study, diphones are used as sound tracks. Turkish linguistic rules are used in order to correctly identify the diphones forming the word. The diphones called from the database are loaded into the developed interface. The fundamental frequencies and energies of the neighboring diphones that will be combined are equalized as much as possible. Afterwards, all the diphones are combined with the existing tunes to obtain the raw synthesis. The integration process is done by overlapping and inserting. That is, the last part of the first diphone and the first part of the second diphone are softened with a window to make a smooth integration. The main objective here is to give prosedic features to the combined sound tracks in a suitable manner. In this way, a natural synthesis can be obtained.

## 1.2. Sound and Physical Properties of Sound

The most important communication tool between the humans is sound. Voice communication is a more preferred form of communication because it is easier and faster than written communication. Sound is a wave that moves from a point to another point in a medium. It is created by a vibrating object. A sound travels through the air with the motion of air molecules (Parker, 2015).

The sound propagation event is the sum of successive events occurring in several interconnected systems.

**Source** $\longrightarrow$ **Medium** $\longrightarrow$ **Reciever**

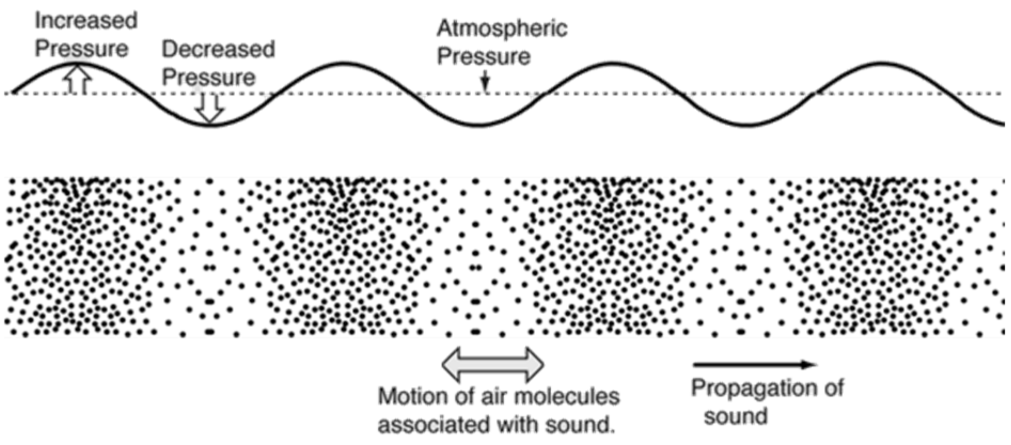


Figure 1.1 The propagation of sound in the air (hyperphysics.phy-astr.gsu.edu)

In the sound event, the acoustic energy emitted by the source affects the receiver while transmitted through the medium and it is perceived by the receiver. In all of these situations there is something that is **propagating** – **transmitting** – **perceiving**. This is energy (Zeren, 1995)

In order to evaluate these sound waves as a sound, they must periodically repeat at least 20 to 20,000 times per second. These limit values are approximate values and vary from person-to-person or age-to-age. The quality of the sound we perceive may vary depending on the source, the medium it emits, and even some features of our hearing system.

Some important concepts related to sound physiology can be described as below:

*Speed of Sound*: The propagation rate of sound waves varies depending on the medium. The speed of sound in air, is about 344 meters per second (344 m / sec) at 21 °C.

*Sound Wave*: Sound is defined as the pressure changes that the human ear can detect between 20 Hz and 20 KHz in liquid, solid, gas mediums. Since the mechanical waves in this frequency range stimulate the hearing, they form sound waves that are particularly important to humans. When a sound wave comes into the human ear, the ear changes the pressure changes in the sound wave to the impulses in the nerves, which are then interpreted by the brain as sounds heard. The simplest sound wave has only one frequency and a constant amplitude. This is called as sine wave. In Figure 1.2 a simple sine wave graph is shown.

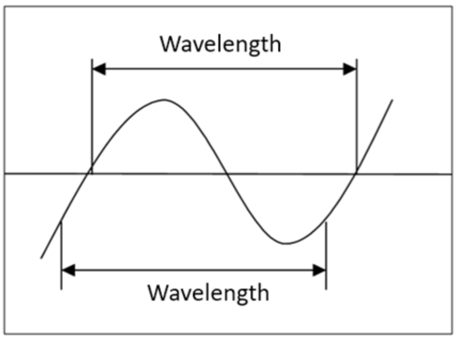


Figure 1.2 A simple sine wave

The length of the sound wave is determined by dividing the propagation speed of the sound by its frequency.

$$\lambda = \frac{c}{f}$$

*Period*: It has been stated that sound can be defined as vibrations in air, or rather air pressure. Each sound has its own vibration character. In order to give a simple example, the simplest voice, a pure tone, will be examined here. In this pure tone, the pressure first increases, then decreases and finally returns to its original state. All of this was completed in 0.1 seconds. The period is the time elapsed for a single repetition of this agitation and it can take different values for different sounds. In Figure 1.3, the period T = 0.1 seconds.

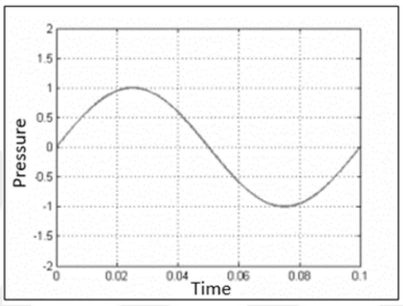


Figure 1.3 Period T = 0.1 sec.

In order to calculate the period of sound waves the following formula can be used.

$$T = \frac{1}{f}$$

*Frequency:* The distance between the two peaks is the wavelength. The number of wave peaks observed per second is called frequency. The frequency determines the pitch of the sound. It is shown as cycles per second (CPS) or hertz (Hz). Low frequencies are bass sounds and high frequencies are higher level sounds. Suppose that the wave defined in Figure 1.3 continues for 1 second. With a period of 0.1 seconds, this wave will have the opportunity to repeat 10 times in a second. In this case, the graph in Figure 1.4 will appear. As you can see, the wave was repeated 10 times in a second. The frequency of repetition per second is expressed in units of Hertz, and therefore the frequency for the above example is 10 Hz.

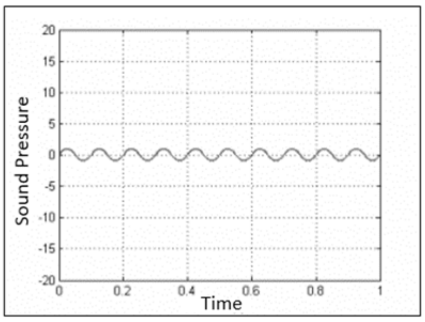


Figure 1.4 Frequency 10Hz.

*Amplitude*: The electrical representation of a sound wave is called a signal or an audio signal. The height of the signal level is called amplitude. The amplitude can be measured in different ways. The top point of the signal level is called as the *peak*. The difference between the positive peak value and the zero point is called the *peak value*, and the difference between the positive and negative peaks is called as *peak-to-peak value* (the value between two peaks).

*Root-Mean-Square (RMS):* RMS is the average amplitude of a signal obtained over a period of time. The average level values in an audio signal do not differ much in themselves, but the peak values change continuously. The RMS value is used to calculate the overall height of the signal over a period of time.

$$RMS = \frac{Peak}{\sqrt[2]{2}} = Peak * 0.707$$

*Noise:* They are non-periodic vibrations. The sounds that push the boundaries in terms of technical sensation of the ear and create psychological disturbance are called as noise

*Phase:* Phase is the relation of the signal or sound waves cycle to the reference time. Phase is expressed in degrees. One cycle is 360$^{o}$. An example sine wave phase state is shown in Figure 1.5.

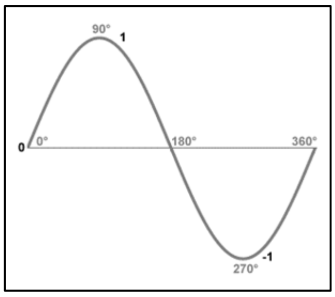


Figure 1.5 Phase

If the amplitude peaks of the two sine waves are in the same place over time, these two waves are in the same phase; in other words, there is no phase difference between these two waves. If the amplitude peaks of the two sine waves are at different locations over time, there is a phase difference between these two waves or in other words a *phase shift*. For example, when there is a $90^o$ phase difference between two sine waves as shown in Figure 1.5; while the first sine wave is at zero point ($0^o$ or $180^o$), the second sine wave is at plus or minus peak point ($90^o$ or $270^o$).

The phase is very important for the combination and mixing of signals or sound waves. If the two sine waves with no phase difference are mathematically added up, they strengthen each other. The total amplitude of the two sine waves which have a $90^o$ phase difference is 1.414 times the amplitude of the waves. Two sine waves with $180^o$ phase difference between them destroy each other. Phase shifts vary according to frequency and time differences. To calculate the phase, shift the following formula can be used:

$\emptyset = \Delta t * f * 360$

$\emptyset$: Phase shift (degrees)

$\Delta t$: Time difference (seconds)

$f$: Frequency (hertz)

In Figure 1.6, a resultant wave, which is formed by combining two sinusoidal waves with frequencies f and 2f, and amplitudes 1 and 0.5, is shown in time and frequency domains.

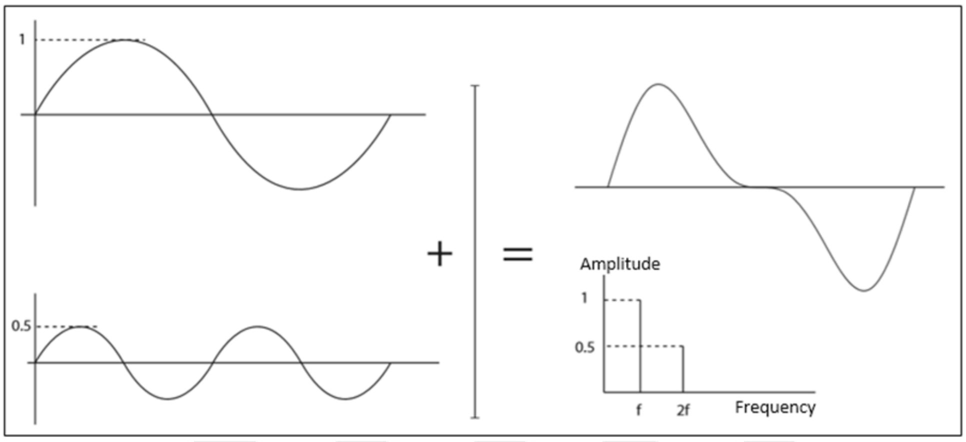


Figure 1.6 Combination of two sinusoidal waves

The amplitude of this wave at any moment can be calculated by the following formula:

$$a(t) = sin(2\pi ft) + \frac{1}{2} sin(2\pi 2ft)$$

*Timbre:* Timbre is a feature that allows people to distinguish the sounds of different musical instruments. With this feature, different instruments playing at the same volume and the same pitch may sound completely different. For example, suppose a guitar and a piano play the same notes from the same octave with the same volume; thanks to their timbre, one can easily distinguish these two instruments. If there was no timbre, in other words, if the notes produced by the instruments were pure sine waves, all the instruments playing from the same note would be perceived as a single instrument. The note produced by the instrument is named as fundamental frequency.

The upper partial frequencies, which are the multiplications of the base frequency, are called harmonics. As an example, let’s take the basic pitch A4 (“La”) sound, 440 Hz as the basic frequency. The first harmonic of 440 Hz is itself (440 Hz x 1 = 440 Hz), the second harmonic (440 Hz x 2) is 880 Hz, third harmonic (440 x 3) is 1320 Hz, and the fourth harmonic (440 Hz x 4) is 1760 Hz. *Octave* means 2:1 frequency ratio. Frequency is doubled if it is increased one octave, and divided by two if it is decreased one octave.

*Envelope:* Envelope is divided into two as acoustic and electronic.

*Acoustic Envelope:* Acoustic envelope is the structure of sound wave in terms of level over time. Acoustic envelope is composed of three parts named as *attack, sustain ve decay*.

*Attack* is the part where the sound starts and then reaches the highest point level. *Sustain* is the section after the attack where the sound extends with small differences in level. *Decay* is the part where the volume decreases and disappears.

*Electronic Envelope:* Electronic envelope is known as ADSR. ADSR is the structure of an electronic musical instrument in terms of amplitude or, in other words, level. ADSR is formed from the first letters of *attack*, *decay*, *sustain* and *release*. *Release* is the section where the level of the signal decreases and disappears.

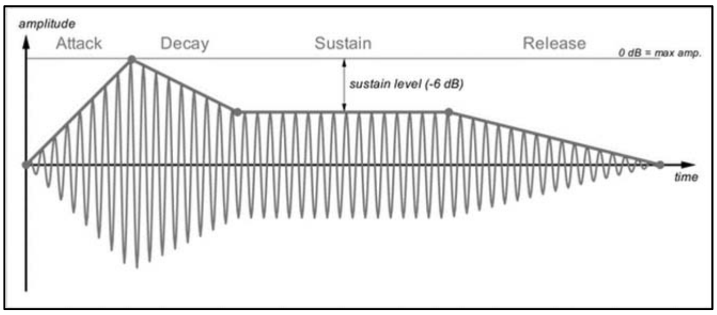


Figure 1.7 ADSR

*Intensity of Sound:* The intensity of sound depends on the vibrational intensity of the vibrating object. Sound intensity, is one of the most important feature of sound like timbre and frequency. Logarithmic scale is used instead of linear scale in sound and signal measurements since large changes in pressure, power and voltage values cause small changes in perceived sound intensity.

*Decibel:* Decibel is a very complicated term for most of the people at first. Decibel is a logarithmic unit used for sound and signal measurements. Because it is logarithmic, it is possible to express very large values with small values using this unit. *Decibel is one tenth of the Bel unit.* In other words, 10 decibels are equal to 1 Bel. The name Bel comes from the name of Alexander Graham Bell; this is why the letter B in the dB abbreviation for decibels is capitalized.

Decibel is always used to express the ratio of two values; decibel itself is not a value. More detailed; decibel is a unit to express the ratio between electrical, acoustic, or other power values logarithmically.

The following formula can be used to express the ratio of two power values in decibels:

$$dB = 10\log(P \div P_{ref})$$

$P$ : Power (Watt)

$P_{ref}$ : Power reference (Watt)

The following formula can be used to convert decibels to power:

$$P = P_{ref} * 10^{(dB \div 10)}$$

*Sound Pressure Level:* The pressure created by sound waves over a certain area is called sound pressure level. Sound pressure level is expressed in Pascal (Pa) or dyne / cm2 units. 10 dyne / cm2 is equal to 1 Pascal.

10 dyne/cm$^2$=1 Pa (Newton/m$^2$)

The threshold of hearing is considered to be 0.0002 Pa. Sound pressure levels are expressed in logarithmic scale and dB SPL (Sound Pressure Level) is used as units. The following formula can be used to convert the sound pressure level in Pascal to dB SPL:

dB SPL = 20 Log (P/Pref)

Table 1.1 Some sound pressure level values

| **Typical sound level in dB** | **Sound source** |
|---|---|
| 140 | Jet engine at 30m |
| 130 | Rivet hammer (pain can be felt at this threshold) |
| 120 | Rock drill |
| 110 | Chain saw |
| 100 | Sheet-metal workshop |
| 90 | Lawn-mower |
| 85 | Front-end loader |
| 80 | Kerbside Heavy traffic<br>Lathe |
| 70 | Loud conversation |
| 60 | Normal conversation |
| 40 | Quiet radio music |
| 30 | Whispering |
| 0 | Hearing threshold |

## 1.3. Speech and Phonetics

The branch of science that studies the sounds of language is called phonetics. Phonology examines what features does the sound has, how it reaches to the listener through sound waves, how the listener perceives that sound, in short, it examines all the features of language related to sound.

In international terminology, sound is called as phonome; sound knowledge is called as phonetics. Phonetics is the basis of phonology which is used for science of sound in international terminology (Coşkun, 2008).

The sound should not be confused with the letter. Sound is related with sound knowledge and letters are related with spelling. The sound is accepted as the cornerstone of oral expression and agreement and the letter as the cornerstone of written expression and agreement. Letters are the ones that make sounds visible in text. The Turkish people use Latin letters but different nations may use different letters (e.g. Iraqis use Arabic letters, Russians use Cyrillic letters). While the letters corresponding to sounds in Turkish are single, in some languages, single sound is formed by one or more letters. Instead of "*ş*" in Turkish, *sh* is used in English and *sch* is used in German; or instead of "*ç*" in Turkish, the letters *tsch* are used in German.

It is not possible to meet all the sounds in languages with the letters in the alphabets. Phonetic experts have created an alphabet that can meet all voices in the world. This alphabet is called as "international phonetic alphabet" (Coşkun, 2008).

## 1.4. Production and Perception of Sound in Humans

Since speech is air-assisted, one of the most important organs that contribute to this is the diaphragm. The diaphragm is located between the lower part of the rib cavity and the upper part of the abdominal cavity. It contracts together with the abdominal muscles to allow air to fill the lungs and discharge from the lungs.

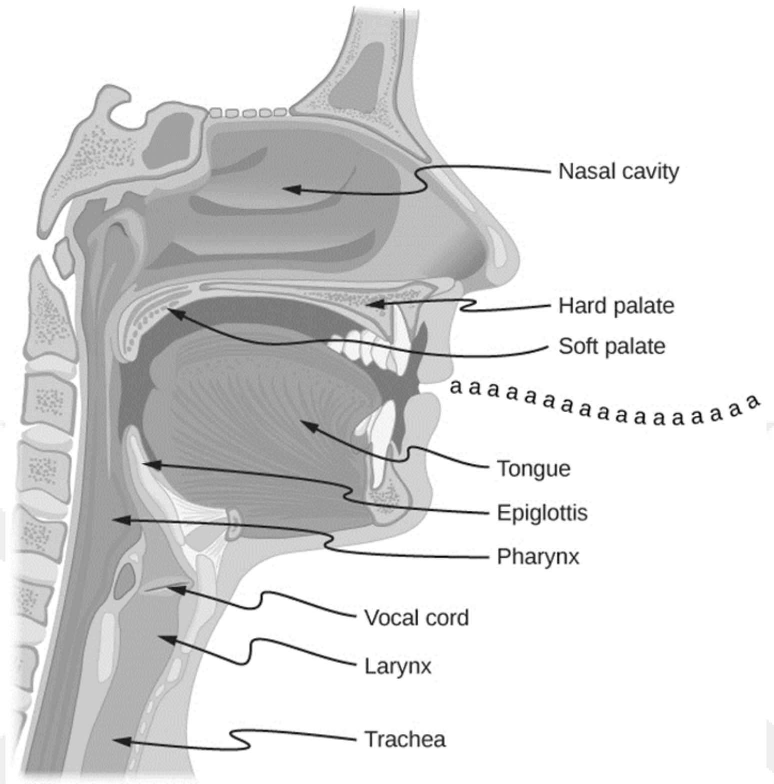


Figure 1.8 Sound production organs of humans(https://courses.lumenlearning.com)

Trachea is above the lungs and it is connected to diaphragm. At the top of the trachea is the larynx. The larynx can move up and down with the help of muscles. In the larynx, there are two mucosal vocal cords, which are located transversely and produce sound with tone. Vocal cords are organs that can vibrate.

One of the most effective organs in the production of sound is the tongue. The root of the tongue is connected to the bone and the tongue changes its shape according to the sound that will be created. Other organs that contribute to the production of sound are the palate, teeth, lips and lower jaw that can move up and down.

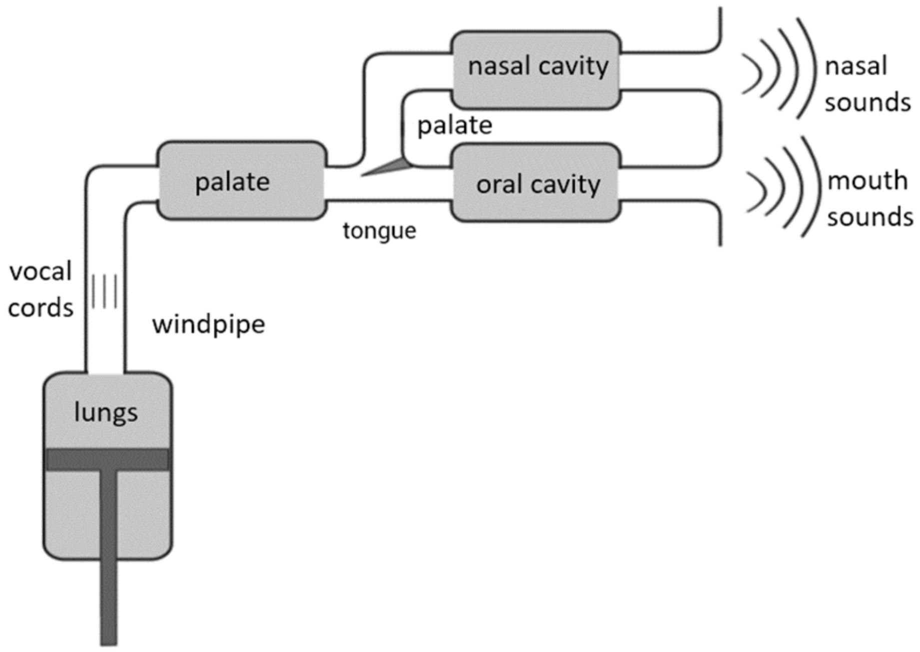


Figure 1.9 Model of human sound production organs (Fant, 1970)

As can be seen in the model of the human sound production organs given in Figure 1.9, the air coming from the lung first passes through the vocal cords. The sound produced by the vibration of the vocal cords contains all the sounds from the thickest to the most treble sounds that can be made by the human. This sound, which includes sound components at every frequency, is actually similar to a wheezing or, in other words, noise. The amplitude of the wheezing sound produced in the vocal cords is high in thick and low in treble sounds. The form of the mouth and nasal cavity produce the human sound by shaping the wheezing sound.

The brain ensures that all of these organs, active or passive, work in harmony in the production of human sound. As mentioned earlier, the sounds that are produced while talking are caused by the displacement of air flow within the respiratory system. At the start of the speech, air is pumped upwards by the lungs, which act as initiators.

The air which moves upward from the air tube passes through the larynx, which has two vocal cords. The air continues its move by going out from the point of larynx opening to the throat. There are two ways in which airflow can move out of the respiratory system: the oral cavity or the nasal cavity. The entrance to these gaps is adjusted by the palate (Özsoy, 2004).

As the airflow moves through the organs and exits from the oral or nose cavities, it encounters some obstacles in the oral cavity. According to the feature, location and degree of these barriers, voices form two main groups:

- Consonants
- Vowels

Consonants are formed as a result of a significant obstruction of the airflow. Vowels, on the other hand, occur as a result of a lesser degree of obstruction of airflow compared to consonants.

### 1.5. Children's Sound Acoustics

Human sounds are divided into three groups as female, male and child sounds. Children's sounds cannot be separated into male or female sounds until adolescence, because they do not have a distinct difference until this time. Sound characteristics of children are different from adults. Anatomical and physiological differences between adult and pediatric larynx cause different sound parameters. These anatomical and physiological differences are:

- While the vocal tract is sufficiently developed for respiratory, swallowing and airway protection in children, the sound characteristics are limited.
- Cartilage structures are softer and more flexible, vocal cords are shorter and density of muscles are less for children.

- The tongue is more bulky, short and has limited movements within the oral cavity for children.

All these anatomical differences cause different sound parameters for children than adults:

- In children, sound is more treble than adults and is more prone to nasality as a resonance feature.
- Fundamental frequency values are above 250 Hz (80-150 Hz for male, 150-250 Hz for female).
- The timbre is high and the sound range is narrow.
- The sound intensity is moderate to high, the sound stability is low and inconsistent, and the voice onset is predominantly hard.

Children's sound width and area, narrow to wide, shows a development line that extends towards more thick sounds until the age of 6 and both thick raw and thin sounds after the age of 6. However, the child's sound width and development line are not the same for all ages and children. The sound width and the development line vary according to the natural, social and cultural environment in which the child lives. However, in a study conducted in Western Europe it is shown that the infant's voice, which is around la 1 (sound of la piano in the 4th octave) during infancy, shows a development line that reaches to fa 1 at the age of 1-2, to mi 1 at the age of 3-5, and to si 1 at the age of 6-8.

In children, the physiological sound range is wider than the musical sound range. In childhood, the physiological sound range (the range between the highest and the lowest sound) remains fairly stable, but the musical sound range develops. In order to sing during the development process, they can generally use a range of around 1.5 octaves (Lunchsinger et. al., 1967).

## 1.6. Turkish and Sound Rules

There are 29 letters in the Turkish alphabet and they are generally classified as vowels and consonants.

***a, b, c, ç, d, e, f, g, ğ, h, ı, i, j, k, l, m, n, o, ö, p, r, s, ş, t, u, ü, v, y, z***

In response to the 29 letters in written language, it can be said that there are 36 sounds (with extra sounds) used during the speech. For example, with a, e, g, k and l letters 10 sounds are tried to be produced as shown in Table 1.2.

Table 1.2 Letters can replace different sounds

| Sound | Sample Word(s) |
|---|---|
| a | zar |
| a | saat, alim |
| e | yemek |
| e | el, vermek |
| g | geri, gemi |
| g | gaga, gayda |
| k | kalın, katı |
| k | keser, kivi |
| l | sal, kalın |
| l | lastik, lazım |

### 1.6.1. Vowels

Vowels are sounds obtained by resonating the vibrations generated by vocal cords in the sound path. They form without encountering an obstacle in any position of the sound path. In contrast to the 16 sounds which are described as famous in Turkish, 8 phonemes are defined as meaning separators.

**/a/, /e/, /ı/, /i/, /u/, /ü/, /o/, /ö/**

Vowels are classified according to the angle of the jaw (tongue-to-palate), the position of the tongue in the mouth and the shape of the lips.

***a) Classification of vowels according to the angle of the jaw (tongue-to-palate):***

The vowels are determined as close or open according to the angle of the jaw (i.e. rictus).

Close vowels: **/ı/, /i/, /u/, /ü/**

Open vowels: **/a/, /e/, /o/, /ö/**

***b) Classification of vowels according to the shape of the lips:***

Vowels are classified as unrounded and rounded according to the shape of the lips.

Unrounded vowels: **/a/, /e/, /ı/, /i/**

Rounded vowels: **/o/, /ö/, /u/, /ü/**

***c) Classification of vowels according to the position of the tongue in the mouth:***

In this classification, when the vowel sound is produced, the point of sound extraction is taken into consideration, not the form taken by the mouth. Tongue plays an important role in making sound. Vowels are classified according to tongue regions as; front, central and back.

Back vowels: **/a/, /o/, /u/**

Central vowels: **/ı/**

Front vowels: **/e/, /i/, /ö/, /ü/**

Front vowels can be classified as unrounded and rounded according to the form of sound while being produced.

Unrounded front vowels: **/e/, /i/**

Rounded front vowels: **/ö/, /ü/**

### 1.6.2. Consonants

Sounds that come into contact with an obstacle at any position of the sound path are called as consonants. Vocal cord vibrations are not important in consonants; they are produced mostly by the cuts in the sound.

There are 21 consonants in Turkish:

**/b/, /c/, /ç/, /d/, /f/, /g/, /ğ/, /h/, /j/, /k/, /l/, /m/, /n/, /p/, /r/, /s/, /ş/, /t/, /v/, /y/, /z/**

Consonants in Turkish are classified according to the point of origin, the form of origin and the presence and absence of vocal cord vibrations.

***a) Classification of consonants according to the form of origin:***

The way the air exits from the vocal organs affects the sound. Accordingly, consonants are divided into 5 classes: *explosion, impact, side constriction, seizure* and *nasal passage*.

Explosion consonants: **/b/, /d/, /g/, /ğ/ /p/, /t/, /k/**

Impact consonants: **/r/**

Side constriction consonants: **/l/**

Seizure consonants: **/c/, /ç/, /f/, /h/, /j/, /s/, /ş/, /v/, /y/, /z/**

Nasal passage consonants: **/m/, /n/**

***b) Classification of consonants according to the point of origin:***

The point from where the air exits also affects the sound. According to this, consonants are divided into 8 classes: *double lip, lip-tooth, tongue tip-back teeth, tongue tip-gums, tongue-front palate, tongue tip-front palate, tongue-art palate* and *larynx.*

Double lip : **/b/, /p/, /m/**

Lip-tooth : **/f/, /v/**

Tongue tip-back teeth: **/d/, /t/**

Tongue tip-gums : **/n/, /r/, /s/, /z/**

Tongue-front palate**: /c/, /ç/, /j/, /ş/, /y/**

Tongue tip-front palate: **/l/**

Tongue-art palate : **/k/, /g/, /ğ/**

Larynx : **/h/**

***c) Classification of consonants according to the vocal cord vibrations:***

Some of the consonant sounds vibrate the vocal cords while occurring. These consonants are called as voiced consonants. If there is no vibration in the vocal cords when producing consonant sounds, such consonant sounds are called as unvoiced consonants.

Voiced consonants: **/b/, /c/, /d/, /g/, /ğ/, /j/, /l/, /m/, /n/, /r/, /v/, /y/, /z/**

Unvoiced consonants: **/p/, /t/, /k/, /ç/, /f/, /s/, /ş/, /h/**

An important point in terms of phonology is that; Although there are certain number of consonants and vowels in the world languages, they are in different qualities. For example, / a / sound in Turkish is different from the / a / sounds in English, French, Persian and Japanese. Some of these / a / sounds are close to / o / and some are longer than nasal passage sounds and / a / sound in Turkish.

### 1.6.3. Syllable Concept

Phonemes combine to form syllables. This rule applies to all languages. However, since syllables can be in different forms in different languages, defining syllables is a difficult process. The syllables for each language differ in their structure. Therefore, it is not possible to make a common syllable definition covering all languages. There must be one vowel at each syllable in the Turkish language, without a vowel a syllable cannot be formed.

The words consist of syllables and syllables consists of letters. The syllables form the sound structure of words. Although vowels in Turkish can form a syllable on their own, consonant letters cannot form a syllable without taking vowels with them. Therefore, in a Turkish word the number of syllables is equal to the number of vowels. The reason for this is that, it is not possible to have more than one vowel in a Turkish syllable. The consonants form syllables by getting together with the vowels that follow them. In Turkish, syllables form as follows.

- The syllables that consist of a single vowel: *a, e, ı, i, o, ö, u, ü*
- The syllables that consist of vowel-consonant pairs: *al, at, ak, ay...*
- The syllables that consist of consonant-vowel-consonant: *bel, bol, kal, gel...*
- The syllables that consist of consonant-vowel pairs: *ba, da, ka, la* ...
- The syllables that consist of vowel-2 consonants: *alt, üst, ırk...*
- The syllables that consist of consonant-vowel-2 consonants: *kürk, yurt, renk...*
- The syllables that consist of 2 consonants-vowel-consonant: *krem, tren..*
- The syllables that consist of 2 consonants-vowel-2 consonants: *kramp, branş..*

The syllable system rules in Turkish are sometimes broken due to foreign originated words. When the syllable type is examined according to these rules, 1714952 meaningful or meaningless syllables can be formed in Turkish. In Table 1.3, the general structure of Turkish syllables is given, "C" represents consonants and "V" represent vowels.

Table 1.3 The syllable structure in Turkish

| **Syllable Structure** | **Sample Syllables** |
|---|---|
| V | a, e, ı, i, o, ö, u, ü |
| VC | ab, ac, aç, ad, … ,az, eb, ec, … |
| CV | ba, be, bı, bi, … , za, ze, zı, zi, … |
| CVC | bul, göl, köy, ter, … |
| VCC | aşk,ört, ast, ırk, … |
| CCV | gri, bre, tra, ... |
| CVCC | kalp, dört, renk, sarp, … |
| CCVC | tren, gram, krem, … |
| CCVCC | kramp, branş, … |

Table 1.4 The syllable number in Turkish

| **Syllable Structure** | **Sample Syllables** | **Multiplier** | **Total** |
|---|---|---|---|
| V | a, e, ı, i, o, ö, u, ü | 8 | 8 |
| VC | ab, ac, aç, ad, … ,az, eb, ec, … | 8x21 | 168 |
| CV | ba, be, bı, bi, … , za, ze, zı, zi, … | 21x8 | 168 |
| CVC | bul, göl, köy, ter, … | 8x21x21 | 3528 |
| VCC | aşk,ört, ast, ırk, … | 21x8x21 | 3528 |
| CCV | gri, bre, tra, ... | 21x21x8 | 3528 |
| CVCC | kalp, yurt, renk, sarp, … | 21x8x21x21 | 74088 |
| CCVC | tren, gram , krem, … | 21x21x8x21 | 74088 |
| CCVCC | kramp, branş. … | 21x21x8x21x21 | 1555848 |

In this thesis, syllables are very important because by making processes on the recorded syllables new syllables or words are created.

### 1.6.4. Some Phonetic Rules in Turkish

In Turkish, all rules are in accordance with the movements of organs related to sounds, in case of any strain, the sounds in the word change. These rules are followed when combining consonants or making annexations or adapting foreign originated words. These rules play an important role in text analysis and word synthesis. The formation of a syllable on a computer is done by algorithms that comply to phonetic rules.

Two consonants do not appear consecutively at the beginning of Turkish words, whereas it is seen that some consonants come together at the beginning of some foreign words while spelling. For example, in speaking language, the "*standart*" word is vocalized by bringing a suitable vowel in between consonant letters as "*sıtandart*". The vowel that will come between two consonants can be determined according to the rule in Table 1.5. Here, sample rules are given for words like "*kral, grup, tren*".

Table 1.5 Adding vowels into words that start with two consonants

| **The vowel in the syllable** | **The vowel that will come in between two consonants** | **Sample** |
|---|---|---|
| **a, ı, o** | ı | gıramer, kıredi |
| **u** | u | gurup |
| **ü** | ü | bürüt |
| **e, i, ö** | i | tiren, tirilyon |

There are no long vowels in Turkish originated words. The long vowel is seen in Arabic and Persian originated words that are used in Turkish: *şive (şi:ve), şair (şa:ir), kaide (ka:ide), numune (numu:ne), şube (şu:be), adalet (ada:let),* *badem (ba:dem), iman (i:man), beraber (bera:ber), ifade (ifa:de), isaret (isa:ret), rica (rica:), vali (va:li), idare (ida:re), vefa (vefa:).* In these examples, the sound shown by the letter before the two points is long vowel and it is said as long (TDK, 2000).

When words ending with closed syllables containing long vowels, are merged with a affix that starts with a vowel letter or are used with auxiliary verbs, the long vowel comes out again in the syllable*: usul / usulü (usu:lü); esas / esasen (esa:sen); hayat / hayatı (haya:tı); ruh / ruhum (ru:hum); kanun / kanunen (ka:nu:nen); vicdan / vicdanen (vicda:nen)* (TDK, 2000).

In some Arabic originated words, the larynx consonant is located at the end of the syllable. In such words, the larynx consonant is completely removed from the Turkish speaking and leads to the prolongation of the vowel before it: *telif (te:lif), dava (da:va), mana (ma:na), tecil (te:cil), memur (me:mur), tesir (te:sir).* (TDK, 2000)

When a word ending with a consonant letter is followed by a word starting with a vowel letter, they are said by connecting these two sounds. This is called "*ulama*" in Turkish language. For example; whereas "*satın almak*" phrase is divided into syllables as *"sa-tın al-mak"* in spelling, it is splitted as "*sa-tı-nal-mak*" in vocalization (TDK, 2000).

### 1.6.5. Emphasis

In a word or phrase pronounciation of a syllable more strongly than others or in a sentence pronounciation of a word more strongly than others, is called "*emphasis*". The emphasized syllable or word has a stronger and violent pronunciation than other syllables or words. In Turkey Turkish, instead of some exceptions, the emphasis is usually located on the last syllable of the word.

For instance, the importance of emphasis in speech and written language is better understood when the sentence "*Ben senden çok sıkıldım.*" is examined:

1. ***Ben** senden çok sıkıldım*. (I'm more bored than you. You're not the one more bored. It is me.)

2. *Ben **sen**den çok sıkıldım*. (I am so bored of you not anyone else.)

3. *Ben sen**den** çok sıkıldım*. (I'm more bored than you are.)

4. *Ben senden **çok** sıkıldım*. (I'm sick and tired of you.)

5. *Ben senden çok sıkıl**dım***. (I can no longer take you, cannot stand you, cannot live with you.)

As can be seen in the above sentences, emphasis is decisive in meaning and expression of words and sentences. It is extremely important in the use of language.

In proper place names (city, district, etc.), with a few exceptions, emphasis is on the first syllable: ***Kon**ya, **Bur**sa, **Muğ**la, **Or**du.*

While calling, the emphasis is on the first syllable for the people names: ***Ah**met! **Ha**san! **Nes**lihan!*

Emphasis on most of the adverbs is in the first syllable: ***ya**rın, **son**ra, **yi**ne, **şim**di*.

Emphasis on conjunctions is mostly in the first syllable: ***çün**kü, **ya**hut, **yal**nız*.

Emphasis on exclamatory phrases is in the first syllable: ***ey**vah! **hay**di!*

Emphasis on question words is in the first syllable: ***ne**rede? **na**sıl? **ni**çin? **han**gi? **kim**de?*

Emphasis on adjacent words is in the first word and is in the last syllable of that word: *kö**pek**balığı, de**ve**boynu, sa**ray**burma*.

## 1.7. Sound Synthesizing Techniques

Text to speech (TTS) is a natural language modeling process that requires changing units of text into units of speech for audio presentation. This is the opposite of speech to text, where a technology takes in spoken words and tries to accurately record them as text. The text-to-speech (TTS) synthesis procedure consists of two main phases. The first one is text analysis, where the input text is transcribed into a phonetic or some other linguistic representation, and the second one is the generation of speech waveforms, where the acoustic output is produced from this phonetic and prosodic information. These two phases are usually called as high- and low-level synthesis.

Two approaches are seen in Text to Speech Synthesizing Systems. These approaches are;

- Natural sound addition method
- Artificial sound production method

### 1.7.1. Natural Sound Addition Method

The method of adding sounds is based on the principle of using sound recordings produced by human. In the studies carried out in accordance with this method, the method of adding from a largest part to a smallest part is tried. The sounding systems based on the method of adding natural sounds are considered as high quality in terms of sound.

**Sentence, Phrase and Word Based Vocalization**

Sentence, phrase and word based vocalization method is used especially in computerized response systems. In such systems, because the texts that will be converted to speech are certain, the speech form of the response sentences in the database are used. For example; "*Kuruluşumuzu aramış olmanızdan dolayı teşekkür ederiz*".

In some implementations, instead of the whole sentence word patterns that may be included in the sentence are vocalized and stored in the database. Thus, the number of vocal recordings in the database is tried to be reduced. For instance, in the sentence; "***Ankara'ya*** *gidecek yolcuların* ***212*** *numaralı bekleme salonuna gelmeleri beklenmektedir*." the city names and waiting room numbers, which are seen as dark, are stored in one place and other words are stored in another place. Bold words are variables; others are accepted as constants.

By changing the variable words, the same information can be read as: *Adana'ya gidecek yolcuların 344 numaralı bekleme salonuna gelmeleri beklenmektedir*.

There are also vocalization methods in which only vocalized words are kept in databases. For example; according to the morphological features of the language different vocalized forms of the words *"Ankara", "gitmek", "yolcu", "212", " numara", "bekleme", "salonu", "gelmek",* and *"beklenmek"* are also kept in the database. This method, which is seen as a viable solution for Indian European languages that are limited in terms of affixes, is not suitable for languages being rich in affixes such as Turkish.

There are also examples in the sentence vocalization method where only one or more words are used. For example; "***xxxxxx'ya*** *gidecek yolcuların* ***yyyyy*** *numaralı bekleme salonuna gelmeleri beklenmektedir*". In this sentence, the pattern sentence, except the city name and the waiting room number of the plane, is kept in a distinct record than the city name and waiting room number (they are also kept in separate records). When the sentence is vocalized, the city name and waiting room number are added to the appropriate points. In this application, some adjustments should also be made for languages such as Turkish (i.e. rich in affixes).

**Format Unit, Syllable, Pair of Sounds and Voice-based Speech**

In general, it is not feasible to vocalize a text by adding consecutive sentences, phrases or words that are vocalized by humans, since it requires a very large database. Therefore, different methods have been tried. In one of these methods, the word is divided into pieces morphologically. For instance, *gözlükcü* word is splitted as *göz + lük + cü* to its root and affixes. The database consists of vocalized root words and affixes. When the "*gözlükçü"* word is vocalized the root is vocalized with the affixes.

Vocalization by adding format units in languages that are limited in terms of affixes (e.g. English), are easier than adding in languages that are rich of affixes (e.g. Turkish). Meanwhile, the fact that the affixes will change according to the sound adaptation rules in Turkish should not be forgotten.

Converting the possible syllables in a language to speech and making use of them while vocalizing a word is another method that can be used in the process of text-to-speech.

Table 1.4 gives the possible number of syllable patterns. Looking at these numbers, it can be said that there are 1714952 syllable structures in Turkish. According to this, when the 1714952 syllables are converted to speech, it can be said that a Turkish text can be realized. However, the meaningless ones in these syllables must be removed.

A syllable database that will be created by recording all the sounds mentioned in the table will of course be very laborious. Instead, a database consisting of only binary syllables are created within the scope of this thesis and other syllables are tried to be produced by using sound synthesizing techniques.

The outlines of a general algorithm that can be used to separate a Turkish word into its syllables is as follows:

- The word is scanned starting from the end to the beginning.
- When the first vowel is encountered, it is cut to the left of the consonant by going one step to the left.

This algorithm, which is outlined above, may be an error-free algorithm when an editing is made for words that have been passed from foreign languages. In this thesis, a different spelling algorithm has been also tried to be developed.

When we go down to a lower level from the syllable structure, we see examples where individual or binary letters are added. In these examples the first step is to divide words vocalized by people into pairs of sounds and sounds. This study requires intensive labor, and at the end it is seen that texts can be converted to speech.

When combining signals related to syllables and binary letters that are vocalized by humans, it is necessary to use the sound characteristics of the language. When the sound characteristics of the language are not involved in the vocalization process, explosions are heard in the sound produced.

### 1.7.2. Artificial Sound Production Method

Another method used for converting text to speech is to produce artificial sound. The basis of the artificial sound production method is based on modeling the human sound production organ. In this method, firstly the frequency characteristics of the vowels and consonants of the language are produced, then by using these sounds syllables and words are tried to be vocalized.

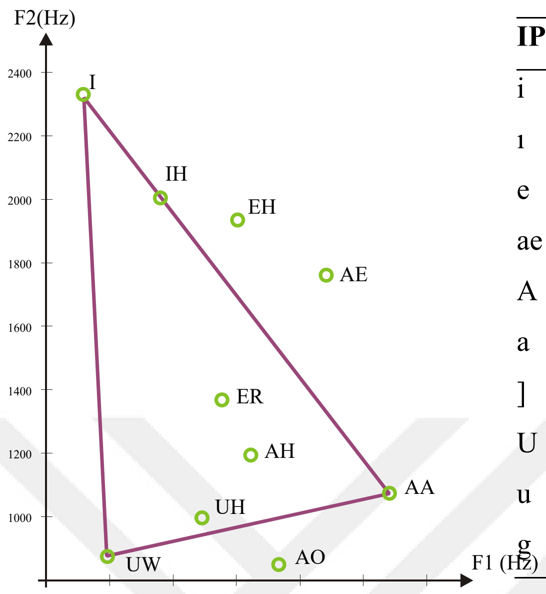
The process of producing artificial sound consists of the following steps:

In the first stage, the text is converted into a phonetic (sound-based, sound-related) alphabet. Although Turkish is accepted as a phonetic language, the fact that our vowels such as "a, e" have thin, thick and long form, it is necessary to convert Turkish text into a phonetic alphabet, too. In languages such as English, where the writing rule is irregular, converting the text into a phonetic alphabet is quite difficult.

In the second stage, the letters in a word written with the phonetic alphabet are tried to be vocalized together. At this stage, letter pairs or syllables can be vocalized.

During the vocalization of a word, two important points to be considered are “emphasis” and “extensions”. In Turkish words, the duration of reading the letters is usually equal. However, words taken from foreign languages break this rule. In Turkish words, it is almost certain where the emphasis will be made in the word and in the sentence; however, the place of emphasis can be changed in spoken language.

"Formant" is a term which was taken from Latin to German in the early 1900s and then into English. It means formative. Phonetic defines the word formant as the frequencies that determine and format the phonetic property of vowels. The lowest formant frequency is labeled as f1. The number of formant frequencies of vowels varies from four to six, but three formant frequencies (F1, F2 and F3) are sufficient for a vowel to be understood. This is the reason why the frequency range of the telephone lines is narrow, the transmitted sound is not of good quality but the conversations can be easily understood. The frequency range of telephone lines is 300 Hz to 3.5 kHz, which covers the first three formant frequencies of all vowel letters.



| IPA | Latin | F1 | F2 | F3 |
|---|---|---|---|---|
| i | IY | 270 | 2290 | 3010 |
| ı | IH | 390 | 1990 | 2550 |
| e | EH | 530 | 1840 | 2480 |
| ae | AE | 660 | 1720 | 2410 |
| A | AH | 520 | 1190 | 2390 |
| a | AA | 730 | 1090 | 2440 |
| ] | AO | 570 | 840 | 2410 |
| U | UH | 440 | 1020 | 2240 |
| u | UW | 300 | 870 | 2240 |
| g | ER | 490 | 1350 | 1690 |

Figure 1.10 Formant Frequencies of Vowels (Coşkun, 2008)

Vowel sounds do not have a single sound frequency; they are combined audio signals. In other words, a vowel sound consists of a basic frequency component and, in addition, lower amplitude frequency components. Although the values of the voice formants vary for each language, the famous formant values given by IPA (International Phonetic Association) are given in Figure 1.10 (Coşkun, 2008).

# 2. RELATED WORKS

Post-graduate and doctoral studies on speech synthesis and related subjects from Turkish texts are given in Table 2.1. When these studies were examined, it was found that the majority of them were on additive synthesizers. In additive synthesizer systems, it is seen that the assembled parts were predominantly phonemes, then syllables and in recent studies mostly diphones (binary phonemes). In addition, the success of different signal processing (combination) methods (e.g. different overlap and splicing methods, sinusoidal model) in additive synthesizers was investigated. Most of the recent studies have tried to develop time and melody models for a more natural speech synthesis.

Table 2.1 Some related works done by year

| Method | M.Sc / PhD | Author | Title |
|---|---|---|---|
| Linear predictive coding based hardware implementation | 1991, M.Sc | Alper GERÇEK | "A TMS 5220 based speech synthesis development system" |
| Linear predictive coding was implemented | 1992, M.Sc | Karen BÜYÜKAŞIKOĞLU | "Konuşma işaretlerinin analiz ve sentezi" |
| Linear predictive coding was implemented | 1992, M.Sc | Enis Sezai BAŞARA | "Yapay ses üretim yöntemleri" |
| Phoneme and diphone concatenation (linear interpolation and implementation with TMS 320 C25) | 1993, M.Sc | İlhan Yaşar ÖZÜM | "A speech synthesis system for Turkish language based on the concatenation of phonemes taken from a speaker" |
| Formant synthesizer for Turkish vowels | 1994, M.Sc | Kamil GÜVEN | "PC based speech synthesis for Turkish" |
| Syllable concatenation at vowel overlaps using PSOLA and its derivatives | 1998, M.Sc | Kerem AYHAN | "Text to speech synthesizer in Turkish using non parametric techniques" |
| Syllable concatenation at consonant overlaps using LP-PSOLA (RELP) | 1999, M.Sc | Özgül SALOR | "Signal processing aspects of text to speech synthesizer in Turkish" |
| TD-PSOLA based implementation | 2000, M.Sc | Barış BOZKURT | "Reading aid for visually impaired (A Turkish text-to-speech system |

| | | | |
|---|---|---|---|
| | | | development)” |
| Formant synthesizer for Turkish words | 2000, M.Sc | Ömer ESKİDERE | “Yazılım tabanlı söz sentezleyici tasarımı” |
| Sinusoidal model based implementation | 2001, M.Sc | Çağla ÖNÜR | “Concatenative speech synthesis based on a sinusoidal speech model” |
| F0 contour synthesis simulations based on syntactic features | 2001, M.Sc | Erkan ABDULLAHBEŞE | “Fundamental frequency contour synthesis for Turkish text to speech” |
| LP-PSOLA based diphone concatenation | 2002, M.Sc | Şifa Serdar ÖZEN | “Türkçe metinden konuşma sentezleme” |
| Diphone concatenation at voiced parts | 2002, M.Sc | Barış EKER | “Turkish text to speech system” |
| Linear additive, mean phoneme/triphone and triphone-tree duration models | 2002, M.Sc | Ömer ŞAYLİ | “Duration analysis and modeling for Turkish text-to-speech synthesis” |
| Intonation models in sentences | 2002, M.Sc | Banu OSKAY | “Automatic modelling of Turkish prosody” |
| Word stress patterns and allophone duration analysis using the Festival synthesizer | 2003, M.Sc | Esra VURAL | “A prosodic Turkish text-to-speech synthesizer” |
| Corpus based harmonic coding concatenation | 2004, M.Sc | Haşim SAK | “A corpus based concatenative speech synthesis system for Turkish” |

| | | | |
|---|---|---|---|
| Diphone concatenation using prototype waveform interpolation | 2005, M.Sc | Asude KARLI | "Örnek bir dizi cümle için Türkçe metinden konuşma sentezleyici" |
| Regression trees for duration and F0 models | 2005, PhD | Özlem ÖZTÜRK | "Modeling phoneme durations and fundamental frequency contours in Turkish speech" |
| LP-PSOLA based diphone concatenation on a mobile device | 2007, M.Sc | İlker ÜNALDI | "Taşınabilir cihazlar için Türkçe metinden konuşma sentezleme sistemi" |
| Microsoft Speech Server TTS based implementation for foreign language learning | 2009, M.Sc | Cansel DEMİR | "Konuşma tanıma sentezleme sistemlerinin okul öncesi dönem yabancı dil eğitiminde kullanılması" |
| Rules for Turkish syllable concatenation | 2009, M.Sc | Kenan GÜLDALI | "Türkçe metin seslendirme" |
| Unit selection based concatenative synthesis using TD-PSOLA | 2009, M.Sc | Zeliha GÖRMEZ | "Implementation of a text-to-speech system with machine learning algorithms in Turkish" |
| TD-PSOLA based concatenative synthesis | 2010, M.Sc | Yücel BİCİL | "Türkçe metinden konuşma sentezleme" |
| Waveform concatenation using at most two-letter syllables | 2010, M.Sc | Cavit ERDEMİR (Erdemir, 2010) | "Türkçe metin seslendirme için doğal konuşma sentezleme" |
| Two-phone and syllable based | 2010, M.Sc | Tuncay ŞENTÜRK | "Türkçe metin seslendirme" |

| | | | |
|---|---|---|---|
| concatenation | | | |
| Syllable based concatenation system for the visually handicapped | 2011, M.Sc | Güray ARIK | “Görme engelliler için bilgisayar kullanımının etkinleştirilmesi, erişilebilirlik ve bir Türkçe hece tabanlı konuşma sentezleme sisteminin geliştirilmesi” |
| Natural speech synthesis using duration, pitch and energy modifications | 2012, PhD | İbrahim Baran USLU | Konuşma işleme ve Türkçenin dilbilimsel özelliklerini kullanarak metinden doğal konuşma sentezleme |
| Word synthesis using SOLA methods | 2013, M.Sc | İLHAMİ SEL | Türkçe metinler için hece tabanlı metinden konuşma sentezleme sistemi |
| Unit Selection and HMM based TTS | 2013, M.Sc | EKREM GÜNER | A hybrid statistical/unit-selection text-to-speech synthesis system for morphologically rich languages |
| Clever Learning System design using speech synthesis and recognition technologies | 2013, PhD | ABDULKADİR KARACI | Ses sentezleme ve tanıma teknolojilerini kullanarak Türkçenin ana dil olarak öğretimi için zeki öğretim sistemi geliştirilmesi |
| Statistical speech synthesis and speaker | 2014, M.Sc | AMİR MOHAMMADİ | Speaker adaptation with minimal data in |

| | | | |
|---|---|---|---|
| adaptation | | | statistical speech synthesis |
| Phonetic Alphabet was used | 2015, M.Sc | TİMUR KARAMEHMET | Implementation of turkish text to speech synthesis with rc8660 voice synthesizer |
| linear predictive coding | 2016, M.Sc | UĞUR AYAZ | Text-to-speech synthesis for Turkish using a DSP board |

# 3. MATERIAL AND METHOD

## 3.1. Material

Materials used in the thesis;

- The programming and database languages used in the preparation of the developed software and the hardware on which the software is developed
- Sound data set consisting of binary syllables selected as a sample application for testing the software to be used in sound synthesis and enriching the content of the thesis.

### 3.1.1. The Software and Hardware Properties

The computer used in the development of the software is a personal computer with Intel i7 processor, 800 GB hard disk and 8 GB RAM. The operating system used is the Windows 10 operating system developed by Microsoft for home and office users. Microsoft's Visual Studio software, which can be downloaded for free from the Internet, was installed on the computer.

For the development of the software, C # programming language with compiler in Visual Studio is preferred. C # is an object-oriented programming language where you can develop both web form and Windows form applications.

In order to record the sound syllables to the computer the software developed within the scope of the thesis was used. The raw form of the recorded syllables are created and the surpluses are trimmed.

### 3.1.2. Formation of Single and Double Syllables

The audio data to be used in the thesis were recorded from external media via a microphone connected to the computer. 344 syllable syllables mentioned in Table 3.1 were transferred to the computer using the program developed within the scope of the thesis.

Table 3.1 The number of single and double syllables

| Syllable structure | Sample syllables | Multiplier | Total |
|---|---|---|---|
| V | a, e, ı, i, o, ö, u, ü | 8 | 8 |
| VC | ab, ac, aç, ad, … ,az, eb, ec, … | 8x21 | 168 |
| CV | ba, be, bı, bi, … , za, ze, zı, zi, … | 21x8 | 168 |

Increasing the recording quality of binary syllables causes the file size to grow. Each syllable occupies approximately 30KB. In total, a 12MB folder was created. New methods have been developed to ensure that the binary syllable data takes up less space. For example, when the digital structure of a syllable recorded in VC format is reversed by a software, a sound close to the CV syllable is obtained. For example, the word "lira" can be formed using only "li" and "ra" syllables. However, instead of using "fi" + "il" syllables for the word "fiil", the first used "li" syllable is reversed to obtain "il". Within the scope of this thesis, this method has not been preferred, but it is an open area for new subjects which may be the continuation of this study.

### 3.1.3. Wav File Format

***Wav*** is a sound format based on Pulse-code Modulation (PCM). Since it is based on the same basis as PCM (apart from minor differences), it can be said that it is also an uncompressed and lossless digital audio format, PCM works with the logic of digitizing the sound wave as it is, and is *an uncompressible format*. Since there is no compression, it takes up a lot of space on the disc. However, the simplicity of the system allows it to be operated via a simple digital sound processor. This audio format is currently used in Blu-ray, CD and DVD formats and it is also preferred in digital telephone systems.

Wav file format is a file format created by taking the first three letters of the word "Wave" in English. It is also known as a file format created by IBM and

Microsoft to play small audio recordings on any computer. Its structure is very simple. Unlike MP3 and other compressed formats, wav files are only digitized sounds.

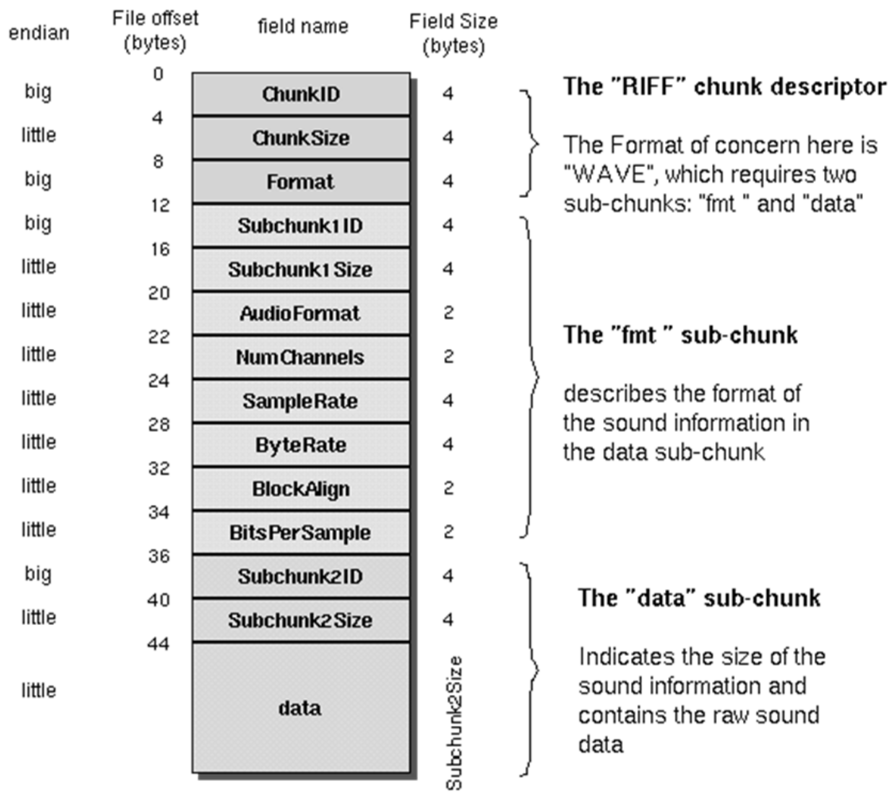
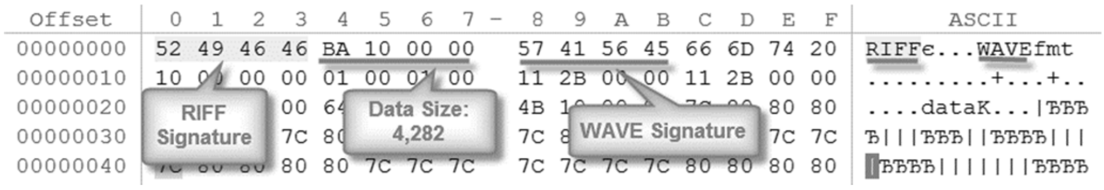


Figure 3.1 Wav file format (http://soundfile.sapp.org)

They are simple and since no compression is applied they do not reduce the quality of sound, but take up a lot of space. Samples in the wav file are kept as uncompressed in the form of raw data. The wav file contains three data regions (chunks). The RIFF (Resource Interchange File Format) chunk is 12 bytes long and it is the region where the file is specified as a “wav” file (Wikipedia).



```
Offset     0  1  2  3  4  5  6  7 -  8  9  A  B  C  D  E  F   ASCII
00000000  52 49 46 46 BA 10 00 00   57 41 56 45 66 6D 74 20   RIFFe...WAVEfmt
00000010  10       00 00 01 00    00   11 2B    00 11 2B 00 00   .........+...+..
00000020           00 64             4B                80 80   ....dataK...|ЂЂЂ
00000030           7C                7C                7C 7C   Ђ|||ЂЂЂ||ЂЂЂЂ|||
00000040              80 80 7C 7C 7C   7C 7C 7C 7C 80 80 80 80   |ЂЂЂЂ||||||ЂЂЂЂ
```

Figure 3.2 Appearance of a Wav file

Table 3.2 RIFF chunk

| Name | Position | Size(byte) | Byte Order | Description |
|---|---|---|---|---|
| Chunk ID | 0 | 4 | Big Endian | Includes the ascii codes of R,I,F,F characters. |
| File Size | 4 | 4 | Little Endian | Includes the size of the data after this section. Value: (File size-8) |
| File Format | 8 | 4 | Big Endian | Includes the ascii codes of W,A,V,E characters. |

The RIFF chunk fields are shown in Table 3.2. The next data region is the FORMAT chunk. In this region format-specific parameters are defined and are 24 bytes long. The FORMAT chunk fields are shown in Table 3.3.

Table 3.3 Format data region

| Name | Position | Size(byte) | Byte Order | Description |
|---|---|---|---|---|
| Chunk 1 ID | 12 | 4 | Big Endian | Includes the ascii codes of f,m,t ve " " (empty space) characters. |
| Chunk 1 Data Size | 16 | 4 | Little Endian | This value is 16 for PCM files. |
| Compression code | 20 | 2 | Little Endian | For uncompressed files, this value is 1. If you read a value other than 1, the value you read shows the format of compression. |
| Number of channels | 22 | 2 | Little Endian | Specifies how many channels are written into the audio file (1,2, ...). |
| Sample rate | 24 | 4 | Little Endian | Holds the value of the sampling frequency. |
| Average bytes per second | 28 | 4 | Little Endian | Indicates how many bytes per second should be read. The value is calculated as: (SampleRate*ChannelNumber *BitForSample/8) |
| Block align | 32 | 2 | Little Endian | The value is calculated as: (ChannelNumber*BitForSample/8) |
| Significant bits per sample | 34 | 2 | Big Endian | Indicates how many bits each sample is expressed. It can take 8, 16, 24 values |

The third data region is the DATA chunk. Actual sampling data is kept in this field. The DATA chunk fields are shown in Table 3.4.

Table 3.4 Data data region

| **Name** | **Position** | **Size(byte)** | **Byte Order** | **Description** |
|---|---|---|---|---|
| Chunk 2 ID | 36 | 4 | Little Endian | Includes the ascii codes of d,a,t,a. |
| Sound data size | 40 | 4 | Little Endian | It shows the size of the data that will come after this part, i.e. the size of the sound data. |

The big endian and little endian rankings mentioned above indicate that in the 2 byte given audio format part, the value "1" is written as 01 00 in the little endian order (hexadecimal). The way it is written according to the big endian rank is 00 01. In the little endian order, the bytes that affect the value least (the least significant) are written at the end, as the name suggests. In the big endian order, the most significant bits are written to the end. This should be taken into consideration when reading audio files byte by byte.

### 3.1.4. Digitizing and Sampling of Sound

The way to convert analog sound to digital form is sampling. The signal is sampled multiple times in one second, the height of the wave is recorded. This is actually the measure of logarithm of the height. Since it is impossible to continuously measure the height of the signal, the height of the signal is measured only at certain sampling times and with a limited number of samples.

Digital sound is the state of analog sound signals marked as "1" and "0" (bits) in the binary system. PCM (*Pulse Code Modulation*) is a sound signal digitized as 64Kbps data. The analog sound signal is sampled as 8000 times per second. Each sample is 8 bits. That's 8bits x 8000/s = 64,000 bits/s in total. This ratio is derived from the Nyquist theorem developed by Harry Nyquist (Nyquist, 1928).

***Nyquist Theorem:***

In order to calculate the number of samples we need in a second, we need to look at Nyquist theory. In the Nyqusit theorem, it has been decided that communication channels can capture and carry high frequency sounds with a sampling of 4000 KHz. According to this theory, it is necessary to take N samples to copy a signal completely. N can be found from the following formula:

N = 2 x signal bandwidth = 2 x 4000 = 8000

8000 samples per second would be a sufficient value. This allows a sample to be taken at 125 microseconds.

Analogue levels of sound are converted to 255 digital levels. 255 discrete numeric levels can be achieved with 8-bit data blocks. Therefore, for one second audio transmission, it requires a bandwidth of 8000 x 8 = 64.000 bps.

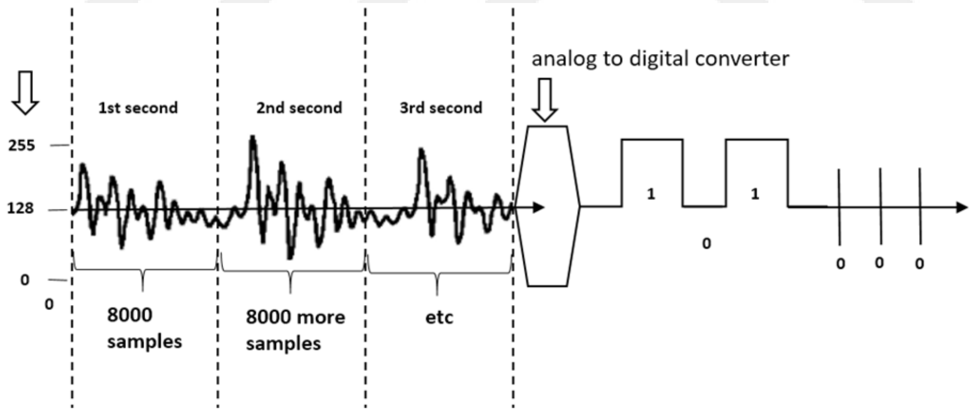


Figure 3.3 Sampling of the sound signal

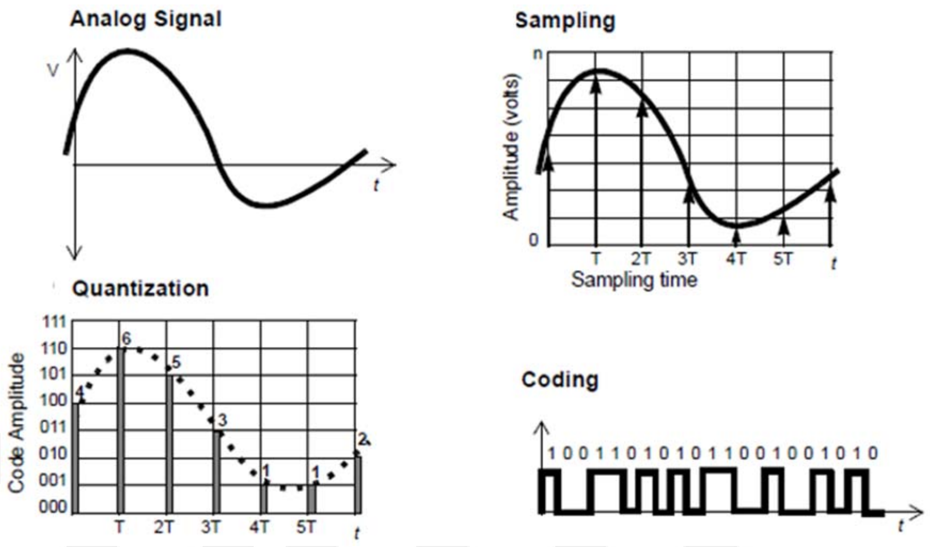


Figure 3.4 Digitizing analog sound signal with PCM

The input sound signal during digital conversion is shown in Figure 3.5. (i.e. http://www.tonmeister.ca/main/ and http://www.terratec.de/4G/2496-en.pdf ):

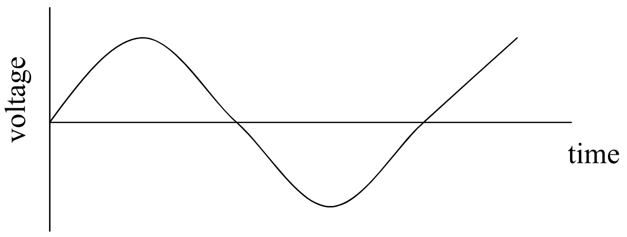


Figure 3.5 Input sound signal

This signal is then divided into specific sampling intervals as shown in Figure 3.6. Here, the voltage value is determined according to the amplitude. Low voltage values are numbered more frequently, and voltage values corresponding to high noises are numbered at longer intervals. This is because the human ear hears logarithmically.

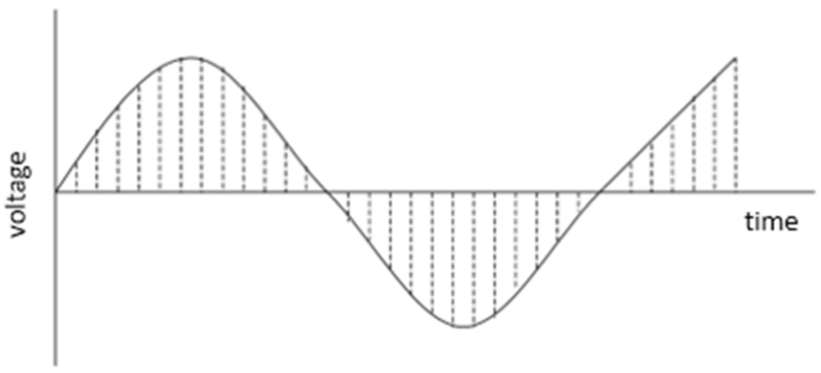


Figure 3.6 Sampling intervals

After sampling, the converter separates the average-value signals from the medium as in Figure 3.7 and rounds the remainder according to the sampling value.

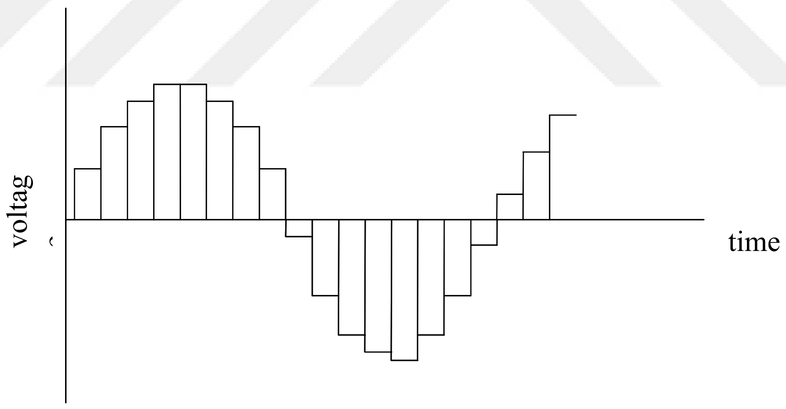


Figure 3.7 Signal with average value

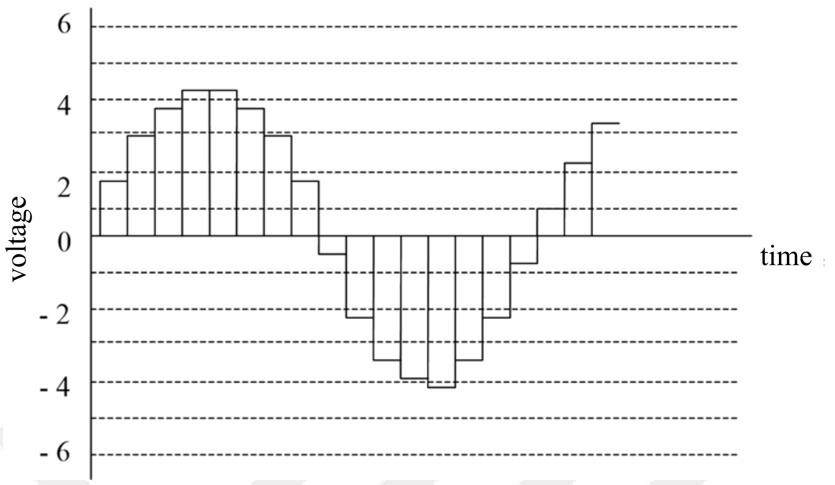


Figure 3.8 Signal and numerical values

Each signal is then assigned a numerical value, as shown in Figure 3.8. This value can range from 0-16536 for 16-bit audio sampling, to 0-16700000 for 24-bit audio sampling. If the value corresponding to the signal does not correspond to an integer, the converter rounds this value to the nearest upper or lower value. This process is called as *quantization*. Figure 3.9 shows this process. As a result of this process, some error margins occurs, namely quantization losses. In contrast to the digitization process, the actual analog sound can never be produced because of these quantization losses that occur during the recovery of the analog sound. However, the human ear is often unable to perceive it.

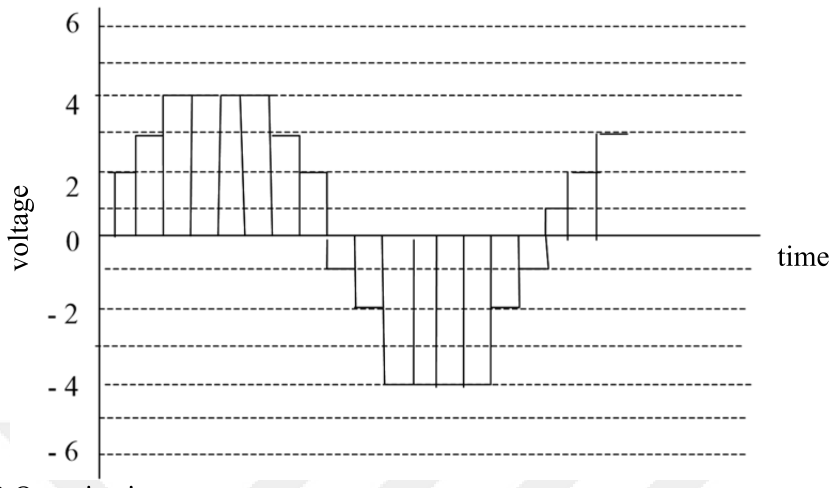


Figure 3.9 Quantization

Finally, the converter tries to create a form closer to the original curve than the above using the reformatting filter. In the example above, the ranges are deliberately large enough to see the event better. The sound curve through the filter is similar to the result in Figure 3.10.

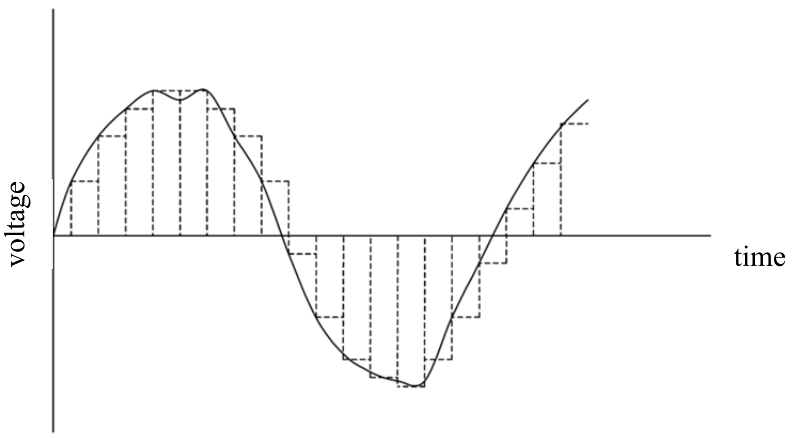


Figure 3.10 Sound passing through the filter

There are basically two different sounds in human speech. These can be called as voiced and unvoiced. When voiced sound is examined in time frame, it can be seen that it shows periodic characteristic. On the other hand, the unvoiced sound shows non-periodic behavior. The waveform of voiced and unvoiced sounds is shown in Figure 3.11.

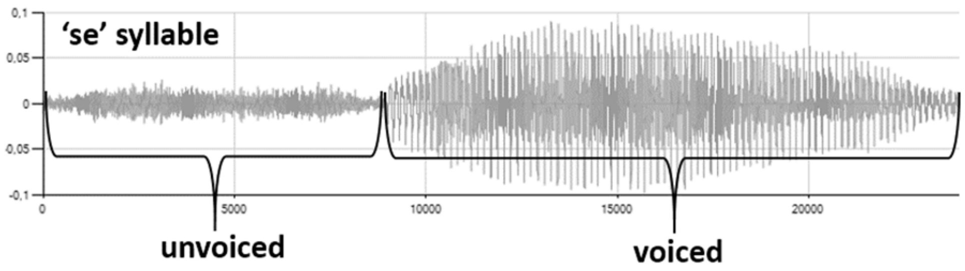


Figure 3.11 Voiced and unvoiced sound

"Pitch" is the name given to a period of sound data and is used in voiced segment which is periodic. Pitch values cannot be mentioned in unvoiced segment because it is not periodic. The pitch value can be calculated by dividing the number of samples in a given sound track by the number of periods. For example, if an audio file contains 1800 samples and 10 periods, the pitch is calculated as 1800/10 = 180. The periodic sound signal and the pitch period are shown in Figure 3.12.

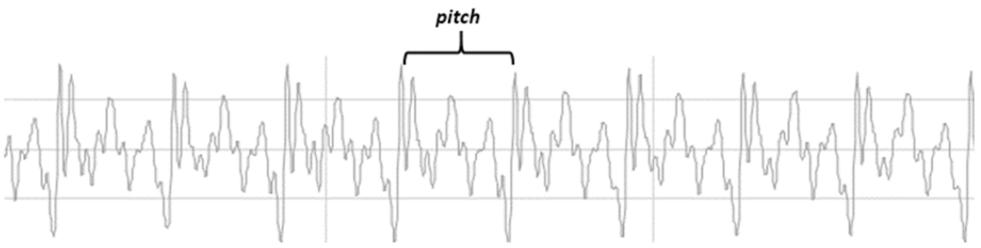


Figure 3.12 Periodic sound signal and pitch period

While the frequency is a single and constant sound wave that vibrates at a certain number of seconds, the *pitch* is composed of complex sound waves that are heard with the overtones and vibrate at a certain number of seconds. The scope of the pitch is broader than the frequency. It carries the amplitude and the timbre which emerges from the combination of upper and lower overtones and also gives the color of sound. These two concepts are not exactly the same. Frequency can be defined as an objective and scientific concept and pitch as a subjective concept. The vibrations of the sound wave at a given period, which is the common point in both concepts, are scaled by frequency and the frequency gives the number of sound vibrations within a second. The perception of the pitch is different from that.

### 3.1.5. Preprocessing Sound Data

Digitized signals are kept as binary words in computer environment. In the signal processing phase, it is often not possible to process all the data that represent the audio expression at once. Therefore, the expression has to be divided into pieces of significant length. Algorithms for operations such as Fast Fourier Transformation (FFT) work on datasets of certain length (a certain number). The data set to be processed at one time is defined as a window in the signal. Since the audio data are the values obtained by sampling, a time unit is obtained by multiplying the number of samples with the sampling period. Therefore, the window can also be considered as a time zone. The process of separating audio expressions into certain number of samples is known as windowing.

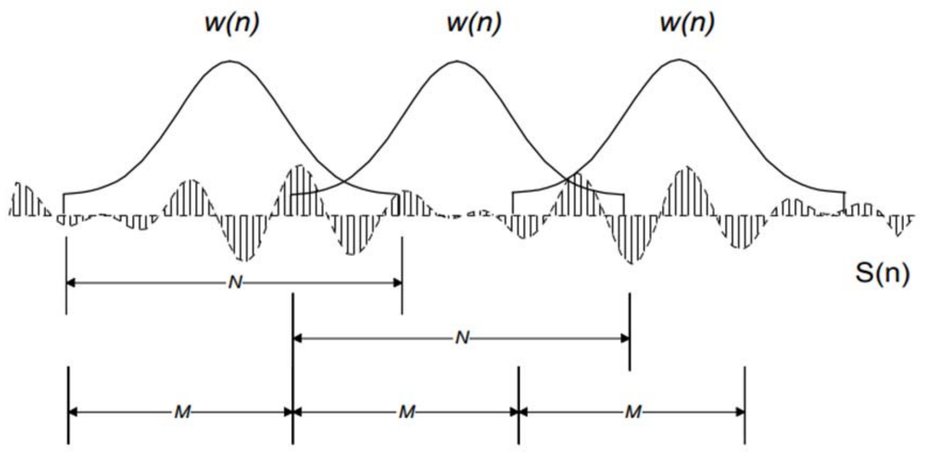


Figure 3.13 Audio signal windowing

Windowing is often performed by creating overlapping clusters (instead of separated data sets) (Morris, January 1989). Window examples created on a digital signal are given in Figure 3.13. In this drawing, (N-M) sample in the N-sampling window consists of samples that overlap the samples in the previous window. For audio expression signals, the width of the windows must be short enough to contain the characteristics of a single sound and long enough not to lose consecutive pitch harmonics. The window widths for audio expression synthesis are about 25-75 ms. For a sample rate of 10 kHz, a windows containing 512 samples corresponds to 51.2 ms signal time period.

Samples representing the audio signal are stored and processed as windows. By keeping the window length larger than the window generation period, successive windows overlap. Compiled windows are processed during the window generation period. Since the window generation period is smaller than the window length (N), the number of samples (M) obtained at the end of the window generation period does not fill a window. The window presented in step $i^{th}$ step includes (M) samples compiled during the $i^{th}$ window generation period. The remaining (N-M) samples are obtained from the samples obtained at the end of the previous ((i-1)th) window generation period. However, it is not possible for the first compiled window (i=1) to do so. Because there is no such (i-1)th window. Some data of the first window would remain empty, because before the first window is completely filled the generation of second window should be started. These datas are filled in with 0. This process is called zero padding.

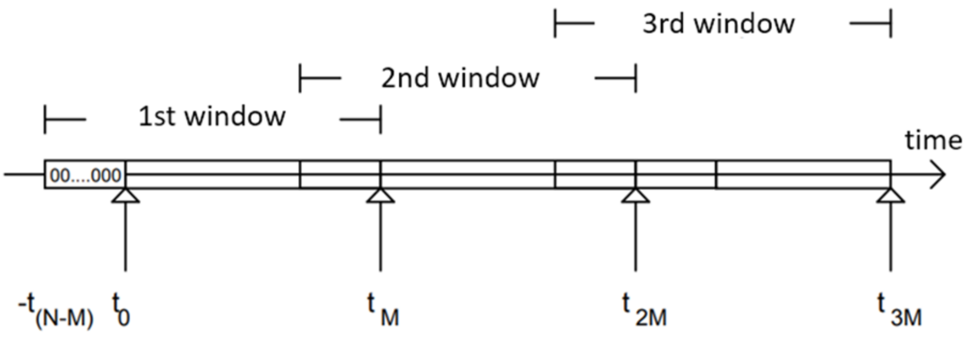


Figure 3.14 Cascade windows

According to the principle described above, the compiled windows consist of a series of (N) samples. These samples are part of the signal being examined. Since the segments in front of and behind this segment are not handled during operation, the form of the signal masked with a rectangle is examined. In this case, treating the signal as if it were at the zero level outside the processed window naturally creates problems. In order to alleviate this problem, windowing is considered, in a more general perspective, as convolution of the signal with a suitable function. Convolution process is a special product of the signal with the window function selected on the time axis.

Convolution process is expressed as given below:

$$y(t) = x(t) \otimes w(t)$$
$$y(t) = \int_{-\infty}^{+\infty} x(t-\tau) w(\tau) dt$$

Here, x represents the signal, w the convolution function, and ⊗ the convolution operation. The format of the convolution process for discrete functions is as follows:

$$y[k] = x[k] * w[k]$$
$$y[k] = \sum_{i=-\infty}^{+\infty} x[i].w[k-i] = \sum_{i=-\infty}^{+\infty} w[i].x[k-i]$$

Some classic functions are used as convolution functions. These special functions are given below with their views. From these, the rectangular window function corresponds to the above-mentioned rectangular window.

***Rectangular window:***

$$w[i] = \begin{cases} 1\ if\ 0 \le i \le n-1 \\ 0\ other\ ways \end{cases}$$

***Barlett window:***

$$w[i] = \begin{cases} \frac{2i}{n-1}\ if\ 0 \le i \le \frac{n-1}{2} \\ 2 - \frac{2i}{n-1}\ if\ \frac{n-1}{2} < i \le n-1 \\ 0\ if\ not \end{cases}$$

***Hanning window:***

$$w[n] = \begin{cases} \frac{1}{2}\left\{1 - \cos\left(\frac{2\pi n}{L-1}\right)\right\}\ if\ 0 \le n \le L-1 \\ 0\ if\ not \end{cases}$$

***Hamming window:***

$$w[i] = \begin{cases} 0.54 - 0.46\cos\left(\frac{2\pi i}{n-1}\right)\ if\ 0 \le i \le n-1 \\ 0\ if\ not \end{cases}$$

***Blackman window:***

$$w[i] = \begin{cases} 0.42 - 0.5\cos\left(\frac{2\pi i}{n-1}\right) - 0.08\cos\left(\frac{4\pi i}{n-1}\right) & if\ 0 \le i \le n-1 \\ 0\ if\ not \end{cases}$$

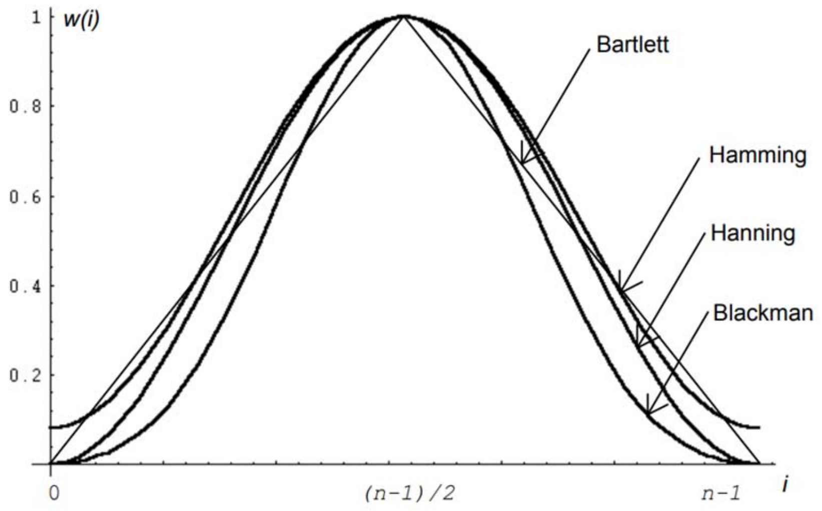


Figure 3.15 Commonly used windowing functions

## 3.2. Method

### 3.2.1. Developed Software

The logic of the Turkish text-to-speech system presented in this thesis is shown in Figure 3.16. Although all stages are shown as sequential and independent stages, such sequence and independence is not applicable for the codes that make up the system. With the compactness of the codes that make up the system, the system still works with the logic shown in the figure. Within the scope of this thesis, a software that can process a sound from every aspect have been developed.

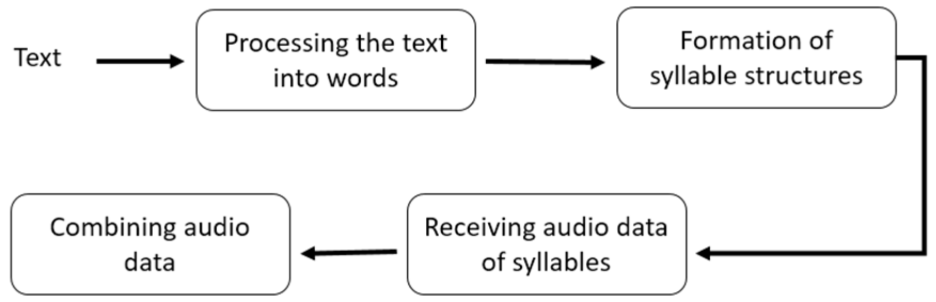

Text
Processing the text into words
Formation of syllable structures
Receiving audio data of syllables
Combining audio data

Figure 3.16 Working structure of the developed system

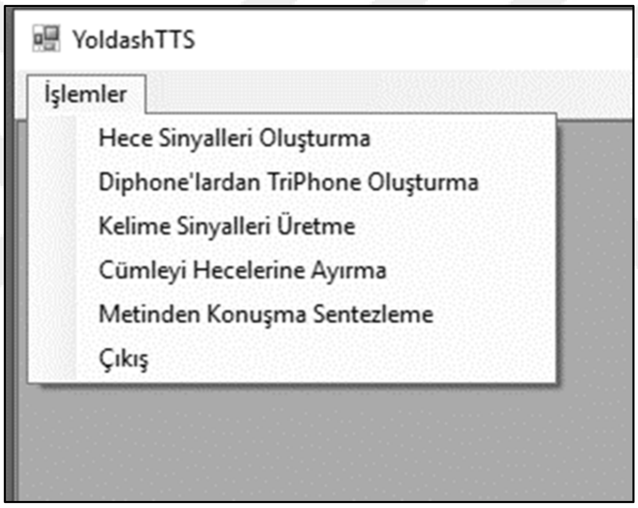
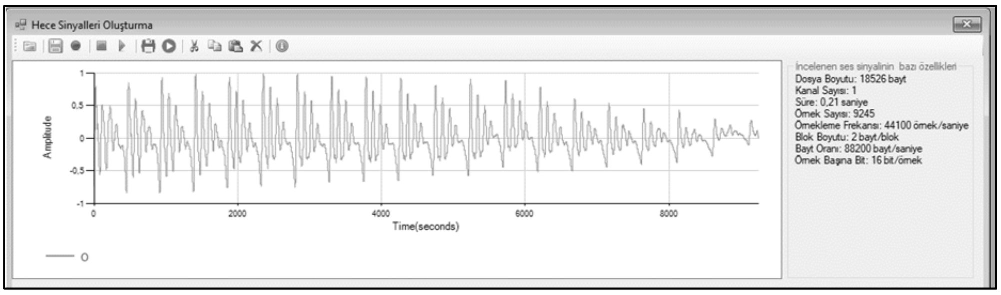

YoldashTTS
İşlemler
Hece Sinyalleri Oluşturma
Diphone'lardan TriPhone Oluşturma
Kelime Sinyalleri Üretme
Cümleyi Hecelerine Ayırma
Metinden Konuşma Sentezleme
Çıkış

Figure 3.17 Menu structure of the software

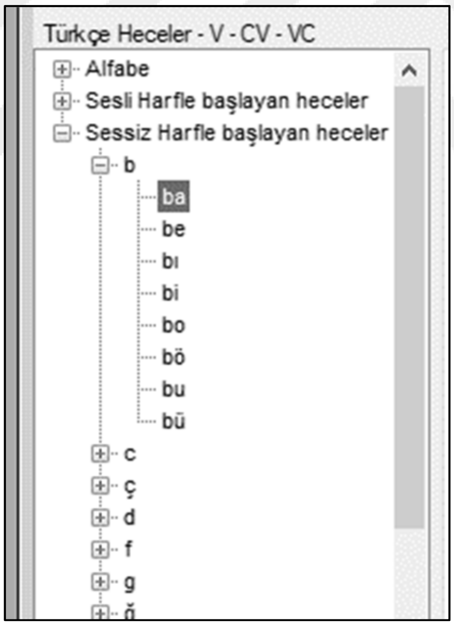
Within the software, the sections that enable the recording of phonemes and diphones have been developed first.



İncelenen ses sinyalinin bazı özellikleri
Dosya Boyutu: 18526 bayt
Kanal Sayısı: 1
Süre: 0,21 saniye
Örnek Sayısı: 9245
Örnekleme Frekansı: 44100 örnek/saniye
Blok Boyutu: 2 bayt/blok
Bayt Oranı: 88200 bayt/saniye
Örnek Başına Bit: 16 bit/örnek

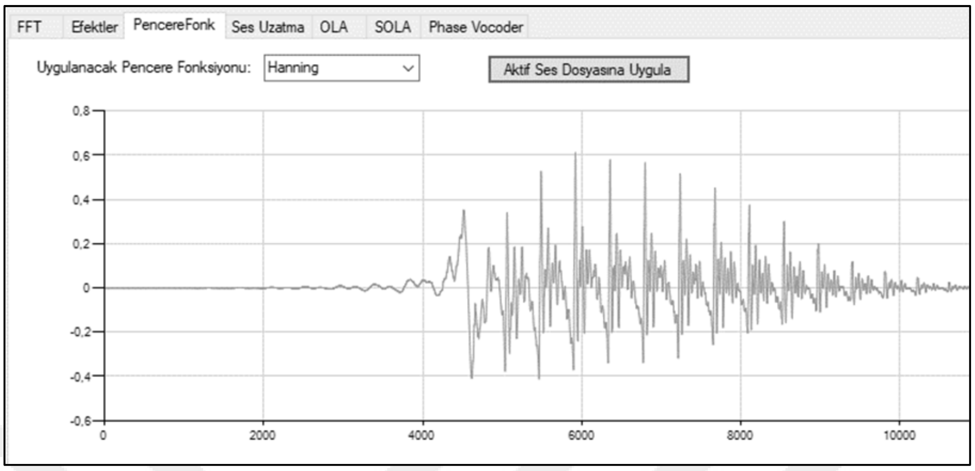
Figure 3.18 Demonstration of the property and wave spectrum of recorded sounds

The user can make all kinds of arrangements related to the sound recorded from the microphone. Clipping of excess sound data can also be performed from this screen.



Türkçe Heceler - V - CV - VC
- Alfabe
- Sesli Harfle başlayan heceler
- Sessiz Harfle başlayan heceler
  - b
    - ba
    - be
    - bı
    - bi
    - bo
    - bö
    - bu
    - bü
  - c
  - ç
  - d
  - f
  - g
  - ğ

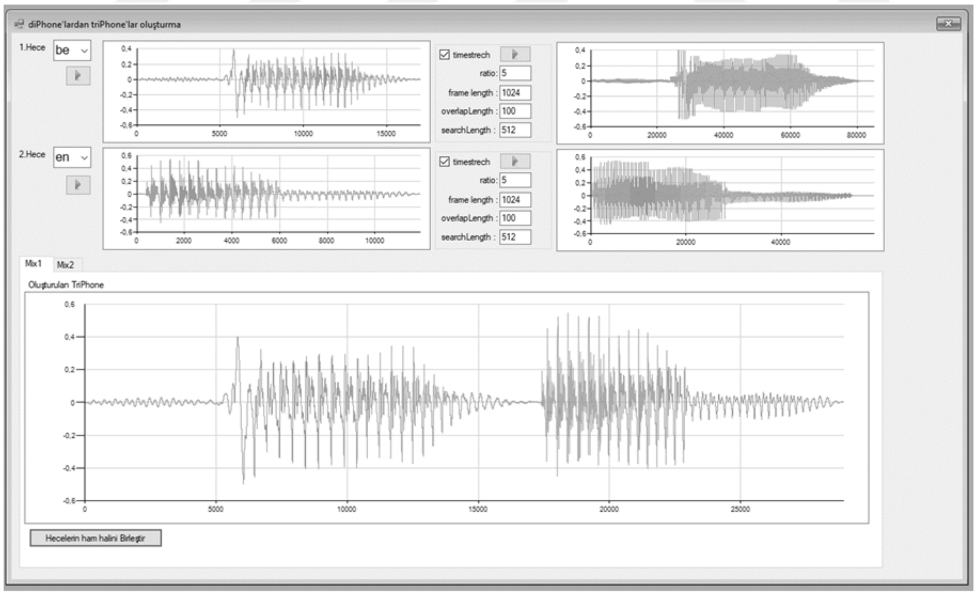
Figure 3.19 The part where syllables are listed and reviewed



Uygulanacak Pencere Fonksiyonu: Hanning

Figure 3.20 The part where various effects are applied to the active sound syllable

There is also a place in the software in order to do tests by applying various effects and observe the results.


diPhone'lardan triPhone'lar oluşturma
Mix1
Mix2
Oluşturulan TriPhone
Hecelerin ham halini Birleştir

1.Hece: be

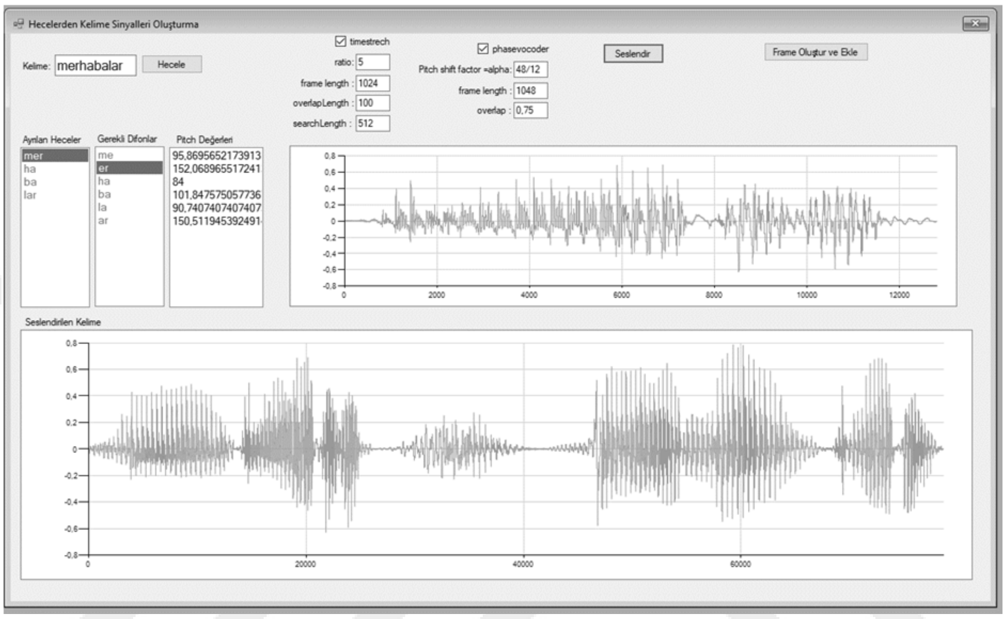
☑ timestrech
ratio: 5
frame length : 1024
overlapLength : 100
searchLength : 512

2.Hece: en

☑ timestrech
ratio: 5
frame length : 1024
overlapLength : 100
searchLength : 512

Figure 3.21 Screen for obtaining triple syllables using binary syllables

The screen in which words are created from syllables and converted to speech is shown in Figure 3.22.



Kelime: merhabalar

☑ timestrech
ratio: 5
frame length : 1024
overlapLength : 100
searchLength : 512

☑ phasevocoder
Pitch shift factor =alpha: 48/12
frame length : 1048
overlap : 0,75

| Ayrılan Heceler | Gerekli Difonlar | Pitch Değerleri |
|---|---|---|
| mer | me | 95,8695652173913 |
| ha | er | 152,068965517241 |
| ba | ha | 84 |
| lar | ba | 101,847575057736 |
| | la | 90,7407407407407 |
| | ar | 150,511945392491 |

Figure 3.22 Text to speech conversion screen

### 3.2.2. The Spelling Algorithm Developed

In order to spell a sentence first of all it should be separated into words. This can be done by referring to various characters (spaces, commas, periods, etc.) that can separate words. In addition, the numbers in the sentence should be reintroduced into the sentence according to their reading.

In the spelling process, the first two letters of the word are important. Firstly, the first two letters of the word are checked as if they are consonants or not. If the first two letters of the word are consonants, a new vowel is added between the two consonants according to the Turkish vowel rule. For example, if the vowel after the first two consonants is a, i and o, vowel i is placed between the two consonants. E.g., the word "*t-rak-tör*"is converted to the form "*t (ı) -rak-tör*". This rule and added vowels are given in detail in Table 1.5.

After this process, all the vowels and consonants in a word are replaced with "V" and "C", respectively. It is not necessary to know each letter in order to separate Turkish words into syllables. For this task, it is sufficient to distinguish the vowels and the consonants. Within the scope of this thesis, such a method has been developed. For example, according to this rule, the word "*balcalı*" is represented as "CVCCVCV". This is the first step of the spelling algorithm. After the word is converted to vowel and consonant notation, grammar rules are applied to the word.

A number of pre-definitions have been made inside the software. According to these pre-definitions, if a word starts with a vowel (V) letter, it will be searched within the word in the order specified by the patterns in Table 3.5 and the first syllable determination will be made.

Table 3.5 Patterns to be search for the words starting with vowels

| **Order** | **Pattern** | **The syllable is** |
|---|---|---|
| 1 | vcccv | Before the 2nd letter |
| 2 | vccc | Before the 4th letter |
| 3 | vccv | Before the 3rd letter |
| 4 | vcc | Before the 4th letter |
| 5 | vcv | Before the 2nd letter |
| 6 | vc | Before the 3rd letter |
| 7 | vv | Before the 2nd letter |

Table 3.6 Patterns to be search for the words starting with consonants

| **Order** | **Pattern** | **The syllable is** |
|---|---|---|
| 1 | ccvcc | Before the 5th letter |
| 2 | ccvcv | Before the 4th letter |
| 3 | cvccc | Before the 4th letter |
| 4 | cvccv | Before the 4th letter |
| 5 | cvcc | Before the 5th letter |
| 6 | ccvv | Before the 4th letter |
| 7 | ccvc | Before the 5th letter |
| 8 | cvcv | Before the 3rd letter |
| 9 | cvc | Before the 4th letter |
| 10 | ccv | Before the 4th letter |
| 11 | cvv | Before the 3rd letter |
| 12 | ccc | Before the 2nd letter |
| 13 | cv | Before the 3rd letter |
| 14 | cc | Before the 2nd letter |

If the word starts with a consonant letter (C), the syllables are made using the patterns in Table 3.6. In this way, each derived syllable is separated from the word and the remaining part is evaluated as a new word so the spelling process continues in this way. For example, the patterns in Table 3.6 are searched in the word "*balcalı*" which is represented as "CV CVCCVCV". Since the "CVCCV" pattern in the 4th row in the table is in "CVCCVCV", the first syllable is created as the letters before the 4th letter and the rest is taken as a new word and subjected to the same rules again (i.e., "CVC"+"CVCV" ("*bal" + "calı"*)). After the first syllable is separated, the word "CVCV" follows the 8th rule in the table, according to which the "CV" before the 3rd letter is separated as a syllable (i.e., "CVC" + "CV" + "CV" ("*bal*" + "*ca*" + "*lı*")).

The codes of the developed spelling algorithm in C # language are presented in the appendix.

### 3.2.3. Creation, Reading, Showing and Vocalization of Wav Audio File

A wav class has been created in the software according to the wav file structure described in section 3.1.3. This wav class and other codes related to this class are presented in the appendix. The RIFF, FMT and DATA sections of all wav sound files are read and placed in an array. Thus, the actual data of the sound files that will be processed are ready on the memory. It is possible to manipulate the sound data that is digitally stored in memory by using various algorithms.

The first 45 bytes of the wav sound packets contain format information and the bytes after the 45th byte form the data block. The bits per sample data is decisive for reading the data block. The data block of the packet with 8 bits per sample is read as 1 byte and the data block of the packet with 16 bits per sample is read by 2 bytes.

The a.wav file used in this thesis is taken as an example.

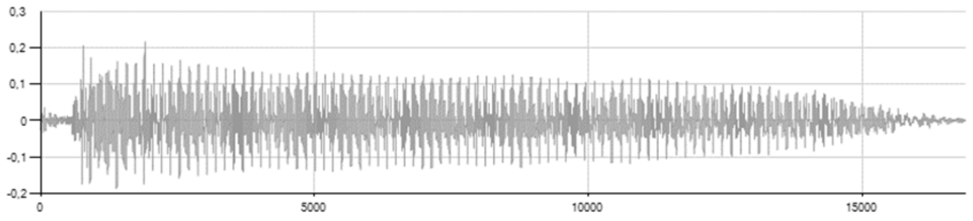


Figure 3.23 The graphical view of a.wav sound file

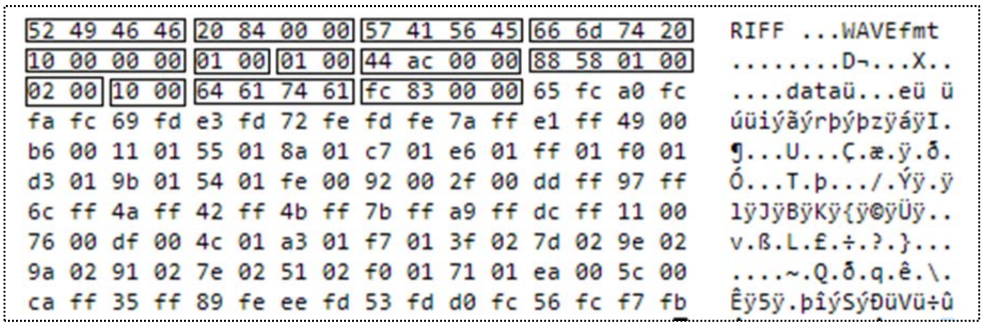
The wav class created within the scope of the thesis is summarized below:

```
public partial class Wav
   {
        public class Header
      {
        /***RIFF*******************/
        public byte[] chunkID;
        public uint fileSize;
        public byte[] riffType;
        /***fmt*******************/
        public byte[] fmtID;
        public uint fmtSize;
        public ushort fmtCode;
        public ushort channels;
        public uint sampleRate;
        public uint fmtAvgBPS;
        public ushort fmtBlockAlign;
        public ushort bitDepth;
        /***data*******************/
        public byte[] dataID;
        public uint dataSize;
        public double wavInSec;
         }
         private byte[] data;
    ….
    }
```

The sound file is a digital data file consisting of bits on the disk. It is necessary to read and understand this data file and reach the data section where the actual data is located. Therefore, wavreader and wavwriter classes have been created to read wav sound files and create new wav sound files. The wavreader class also includes an example function as follows.

```
public static void readFile(Stream stream, out Wav file)
    {
     Wav waveFile = new Wav();
     BinaryReader reader = new BinaryReader(stream);
         waveFile.head.chunkID              =              reader.ReadBytes(4);
         waveFile.head.fileSize      =      36      +      reader.ReadUInt32();
         waveFile.head.riffType             =              reader.ReadBytes(4);
         waveFile.head.fmtID               =               reader.ReadBytes(4);
         waveFile.head.fmtSize              =              reader.ReadUInt32();
         waveFile.head.fmtCode              =              reader.ReadUInt16();
         waveFile.head.channels             =              reader.ReadUInt16();
         waveFile.head.sampleRate            =             reader.ReadUInt32();
         waveFile.head.fmtAvgBPS             =             reader.ReadUInt32();
         waveFile.head.fmtBlockAlign          =            reader.ReadUInt16();
         waveFile.head.bitDepth             =              reader.ReadUInt16();
         waveFile.head.dataID              =               reader.ReadBytes(4);
         waveFile.head.dataSize             =              reader.ReadUInt32();
         waveFile.setData(reader.ReadBytes((int)waveFile.head.dataSize));
         waveFile.head.dataSampleCount =  (int)waveFile.head.dataSize / 2;
         waveFile.head.wavInSec        =        (double)waveFile.head.dataSize/
                                                  (double)waveFile.head.fmtAv
                                                  gBPS;
…….
…..    }
```

Let's look at the a.wav file with a binaryhex editor and by taking the first 160 characters as an example let’s see how it is read (Figure 3.24).



```
52 49 46 46 20 84 00 00 57 41 56 45 66 6d 74 20   RIFF ...WAVEfmt
10 00 00 00 01 00 01 00 44 ac 00 00 88 58 01 00   ........D¬...X..
02 00 10 00 64 61 74 61 fc 83 00 00 65 fc a0 fc   ....dataü...eü ü
fa fc 69 fd e3 fd 72 fe fd fe 7a ff e1 ff 49 00   úüiýãýrþýþzÿáÿI.
b6 00 11 01 55 01 8a 01 c7 01 e6 01 ff 01 f0 01   ¶...U...Ç.æ.ÿ.ð.
d3 01 9b 01 54 01 fe 00 92 00 2f 00 dd ff 97 ff   Ó...T.þ.../.Ýÿ.ÿ
6c ff 4a ff 42 ff 4b ff 7b ff a9 ff dc ff 11 00   lÿJÿBÿKÿ{ÿ©ÿÜÿ..
76 00 df 00 4c 01 a3 01 f7 01 3f 02 7d 02 9e 02   v.ß.L.£.÷.?.}...
9a 02 91 02 7e 02 51 02 f0 01 71 01 ea 00 5c 00   ....~.Q.ð.q.ê.\.
ca ff 35 ff 89 fe ee fd 53 fd d0 fc 56 fc f7 fb   Êÿ5ÿ.þîýSýÐüVü÷û
```

Figure 3.24 The view of the first 160 bytes of the a.wav file with the hex editor

waveFile.head.chunkID = reader.ReadBytes(4);

The code reads the first 4 bytes of the file. The 52, 49, 46, 46 hexadecimal values are converted to decimal as 82, 73, 70, 70 and they correspond to the characters 'R', 'I', 'F', 'F' as can be seen from the ASCII code table (https://en.wikipedia.org/wiki/ASCII).

waveFile.head.fileSize = 36 + reader.ReadUInt32();

The code reads the next 4 bytes of the file (20, 84, 00, 00). In the wav file format structure this section contains the size of the whole file. According to the Little Endian rank, the 00008420 hexadecimal number corresponds to 33824 as a decimal. 36 should also be added to this as the header size. This file appears to be 33860 bytes.

waveFile.head.riffType = reader.ReadBytes(4);

The data read by this code is 57, 41, 56, 45 and it is expressed as decimal 87, 65, 86, 69. In the ASCII code table, the corresponding characters are ‘W’, ’A’, ’V’, ’E’.

```
waveFile.head.fmtID = reader.ReadBytes(4);
```

The 66, 6d, 74, 20 hexadecimal number read with this code expresses 102, 109, 116, 32 as decimal, which means ‘f’, ‘m’, ‘t’, ‘ ’ (one space) in the ASCII table.

```
waveFile.head.fmtSize = reader.ReadUInt32();
```

This code reads the next byte sequence 10, 00, 00, 00. The 00000010 data indicates 16 as decimal. This value is assumed to be 16 for PCM in wav files.

```
waveFile.head.fmtCode = reader.ReadUInt16();
```

This line reads the next two bytes represented by 01, 00. This value is converted to decimal 0001 and this corresponds to 1. A value of 1 in wav files indicates that this file is an uncompressed file. If you read a value other than 1, this value indicates the format of the compression.

```
waveFile.head.channels = reader.ReadUInt16();
```

This code reads the next two bytes represented by 01, 00. This value is converted to decimal 0001 and corresponds to 1. In a way, the number of channels in sound files indicates, with how many different samples does the data inside a sound file is created. So each channel can be thought of as a separate sound file. When the file is played, the sound data sequence on each channel is played simultaneously. Thus, sound sounds more realistic and it is possible to get an idea about the direction of sound. A value of 1 indicates mono channel sound data.

```
waveFile.head.sampleRate = reader.ReadUInt32();
```

This line reads 4 bytes of 44, ac, 00, 00 hexadecimal data. The number 0000AC44 corresponds to 44100 as decimal. This is the sampling frequency value. In order to obtain CD quality sound, 44100 samples should be collected from a sound signal per second. This data indicates that the sound is sampled with a frequency of 44100 hertz. The sampling frequency is independent of the actual frequency of the signal.

```
waveFile.head.fmtAvgBPS = reader.ReadUInt32();
```

With this code, the next 4 bytes are read from the wav file. The hexadecimal data is seen as 88, 58, 01, 00 and is read as 00015888. After that it was converted to decimal number 88200. This value shows the number of bytes that should be read in a second. The following formula is used to calculate this value:

*SampleRate*ChannelNumber *BitForSample/8*

```
waveFile.head.fmtBlockAlign = reader.ReadUInt16();
```

By this code 02, 00 bytes were read and 0002 data was converted to 2 as decimal. This refers to the size of the data block. The block consists of samples that are prepared to be instantly converted into sound on all channels. That is, each block contains as many samples as the number of channels. The block size is calculated by multiplying the number of channels and the size allocated to each sample.

*ChannelNumber *BitForSample/8*

```
waveFile.head.bitDepth = reader.ReadUInt16();
```

10, 00 bytes are read with this line of code. This hexadecimal byte array represents 16 as decimal. This shows how many bits each sample is represented by. This value can be 8, 16 and 24. The data block of the packet with a bit rate of 8 is read as 1 byte and the data block of the packet with a bit of 16 is read as 2 bytes.

```
waveFile.head.dataID = reader.ReadBytes(4);
```

This code reads the 64, 61, 74, 61 bytes in the data shown in Figure 3.24 and converts them to decimal numbers as 100, 97, 116, 100. These values correspond to the characters ‘d’, ’a’, ’t’, ’a’ in the ASCII code table.

```
waveFile.head.dataSize = reader.ReadUInt32();
```

With this line of code, fc, 83,00,00 byte sequence is read and 000083FC hexadecimal number is converted to 33788 as decimal. This number shows the data size of the file after this section. That is, it expresses the size of the actual raw sound data.

```
waveFile.setData(reader.ReadBytes((int)waveFile.head.dataSize));
```

This line of code then reads all raw sound data and transfers it to a byte array. This data is the most important part for this study. Since each sound sample is represented by 2 bytes in the file, when we divide the total number of data (33788) into 2, we can say that the sound in the a.wav sound file actually contains 16894 samples.

Above, the number of bytes that should be read per second for this example wav file is read from the file and it is seen that this value is 88200. If we divide the total sound data (33788) by this value, we will find out how many seconds does this file is. This file contains a sound data of 0.38 seconds.

All the operations on processing the sound is done by playing on every byte of data after the first 36 bytes of the wav file. If you copy the same 33788 data in the sound (a.wav) file one after the other and add only these 33788 data to the file size and data size sections of the file and save the file again, the wav file will be converted to a file containing two audio from the same sound.

Within the scope of this thesis, wav files can be created in the memory with a desired ratio and structure and then stored anywhere using the following structure:

```
private byte[] data;
public Wav(byte[] newData)
{
   head = new Header();
   head.chunkID = System.Text.Encoding.ASCII.GetBytes("RIFF");
   head.fileSize = 36 + (uint)newData.Length;
   head.riffType = System.Text.Encoding.ASCII.GetBytes("WAVE");
   head.fmtID = System.Text.Encoding.ASCII.GetBytes("fmt ");
   head.fmtSize = 16;
   head.fmtCode = 1;
   head.channels = 1;
   head.sampleRate = 44100;
   head.fmtAvgBPS = 88200;
   head.fmtBlockAlign = 2;
   head.bitDepth = 16;
   head.dataID = System.Text.Encoding.ASCII.GetBytes("data");
   head.dataSize = (uint)newData.Length;
   head.wavInSec = (double)head.dataSize / (double)head.fmtAvgBPS;
   data = newData;
   ……
     }
```

When the audio files in the memory are sent to the playback device of the computer, the commands of the winmm.dll file of the Windows operating system are used. This file serves as an interface between the hardware device and the sound file that is waiting in the memory to be played. The header structure containing the steps described above is sent to the winmm.dll file and so the playback device is made ready by the operating system. Then the raw sound data is sent from the buffer to the device byte by byte and so the sound is heard.

The visual representation of the spectrum of sound data, as seen in Figure 3.22, is also derived after processing the raw sound data. The chart component in visual studio is used for graphical representation of sound file. However, the sound samples represented by 2 bytes in the file are converted to double data type. The following codes are used for this translation:

```
public double[] dataToDouble()
    {
      handle = new Handler();
      double[] result = new double[data.Length / 2];
      for (int i = 0, pos = 0; pos < data.Length - 2; i++, pos++)
      {
              result[i] = handle.byteToDouble(data[pos], data[++pos]);
         }
      return result;
      }

public double byteToDouble(byte firstByte, byte SecondByte)
    {
      short s = (short)((SecondByte << 8) | firstByte);
      return s / 32768.0;
      }
```

Each sample of the raw sound data of the a.wav sound file in Figure 3.22 is accessed with the dataToDouble() function. For example, the byte codes of the first sample of the sound data are 65, FC hex numbers. These numbers are expressed in binary form as 01100101 and 11111100. The byteToDouble function shifts the second byte to 8 bit left and creates the 1111110000000000 number and then by applying the "and" operation to this number with the first byte (01100101) founds 1111110001100101. After that it converts it to decimal number as -923. The short variable can store values between -32768 and 32767. When a sound data is displayed, it is reduced to 1 to 0. The obtained -923 value was divided by 32768 to obtain the value of -0.028167724609375. This value is the first point of the sound wave drawn on the screen in Figure 3.25. In this way, the waveform of the sound is drawn on the screen by reading all bytes forming the sound data.

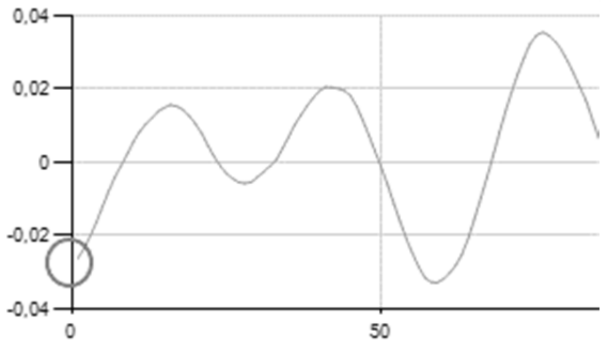


Figure 3.25 Displaying the first sample data in the a.wav sound file

### 3.2.4. Extending the Sound Data

The previously recorded syllable sound data is used in additive and syllable based voice synthesis studies. A syllable is composed of two parts: vowel and consonant. In Figure 3.26 the waveform of the "*çü*" syllable is shown. The part where the letter '*ü*' is located in the syllable consists of similar waveforms that periodically repeat each other.

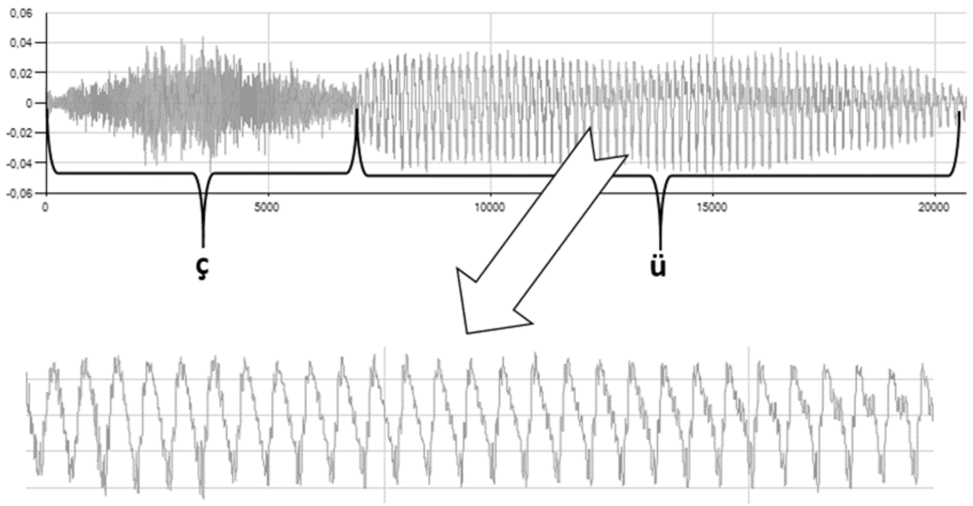


Figure 3.26 Waveform of "*çü*" syllable

Extending or cropping a syllable can be done by interfering with the audible and periodic part of the syllable. The first method that comes to mind is the idea that a long sound can be obtained by adding these periodic parts one after the other. In practice, however, this method is problematic because each segmented sound file has different energy and frequency values and cannot be combined from the right place. The sound obtained by this method is more intermittent than a long "*ü*" sound and it is not fully understood.

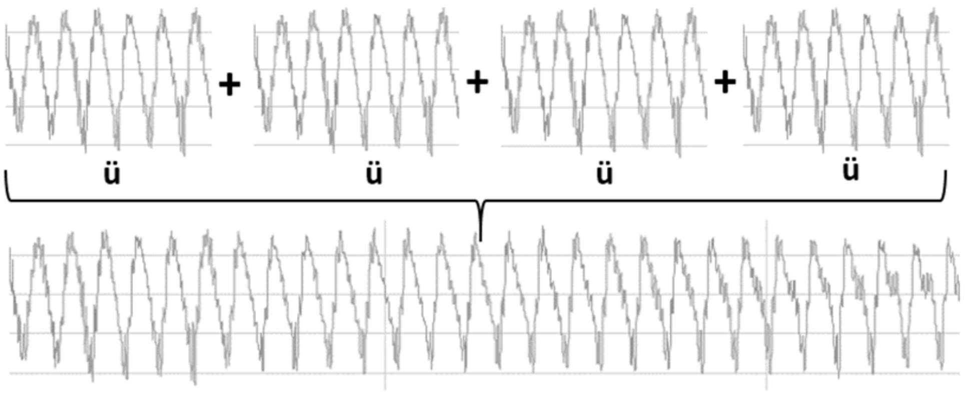


Figure 3.27 Partial addition of ”$\ddot{u}$” sound

To change the speed of speech, it is not enough to keep the recording and playback speed of the sound different. For example, it can be noticed by everyone who listens that when a speed-controlled recording is played back quickly, the sounds vary considerably and when the speed doubles the sounds cannot be understood.

The relation between the original (analog) sound ($x_a(t)$ ) and the accelerated state of this sound in time domain $\alpha$ times ($y_a(t)$) can be expresses as follows:

$$y_a(t) = x_a(\alpha t)$$

When the Fourier transforms of this equation are taken, the representation of it in the frequency domain can be shown as follows:

$$y_a(\Omega) = \frac{1}{|\alpha|} x_a(\frac{\Omega}{\alpha})$$

As seen from this equation, a compression or expansion in the time domain causes an expansion or compression in the frequency domain, respectively.

Therefore, both short-term (envelope or LPC coefficients) and long-term (pitch period and other long-time parameters) relationships of the speech are changed by this process. Since the perception of sound is in the frequency domain for the humans, the intelligibility is impaired. Therefore, in the first designed algorithms, the input sound is divided into 20-30ms timed frames that do not overlap with each other to obtain the desired time scale. The input sound is accelerated or decelerated by discarding or repeating the appropriate ones. This structure, which is called as cut and paste has a simple and fast algorithm, but produces some distortions in sound. These distortions occur from the corruption of the continuity and pitch periods of cut-and-paste points.

In order to eliminate these distortions, cut and paste algorithms considering the continuity have been developed (frames according to local peaks or pitch period) and the quality of the sounds (the time scale of it has been changed) has been improved.

#### 3.2.4.1. Overlap and Add (OLA)

In this method, the sound data is divided into equal blocks. The blocks selected with the time shift $Sa$ are repositioned with the time shift $Ss = \alpha Sa$. Thus, a longer sound data is obtained.

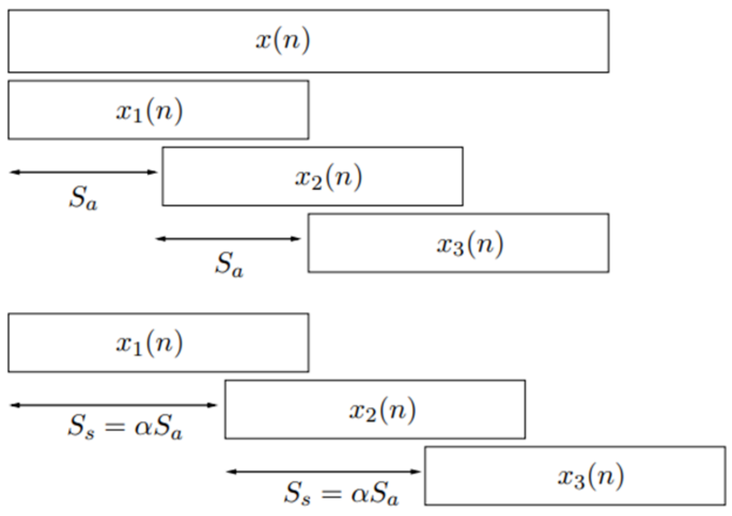


Figure 3.28 OLA algorithm

It is also possible to shorten the sound by selecting a smaller $Ss = \alpha Sa$ value with the same method. The OLA destroys the original phase relationships of the connected input signal particles and then interpolates between the unaligned signal particles to form the new output signal. This causes irregularities and distortions that affect signal quality in pitch periods.

Even if the cut and paste methods pay attention to the continuity of the pitch period, adding or removing a frame may not provide the continuity in the signal at the desired level. The reason for this is, in a normal conversation sounds may seem the same, but the tone and emphasis may change in consecutive pitch periods. This distortion becomes more pronounced in transitions (vowel-consonant or consonant-vowel transitions) since the tone and emphasis change is even greater there.

Instead of adding/subtracting, OLA (OverLap Add) is used to correct these distortions. In the OLA method, frames with a size N of $\Delta n$ distances from each other are used. The distances between the frames are then changed to $\alpha\Delta n$ as the sound is synthesized. Since there will be overlap in the last parts of the previous frames and in the first parts of the frames behind it, *decreasing weight* is given to the values of the previous frame at the overlapping points, and an *increasing weight* is given to the overlapping values of the frame behind it. The values of the overlapping points are found with this weighted sum. In this process, the sum of the weights used at all overlapping points of the two consecutive parts will be equal to 1, these weights will be *decreasing monotone* for the overlapping part of the previous frame and *increasing monotone* for the other. Linear weight window (first and second half of Barlet window) is used mostly, but as an alternative Hamming, Hanning and etc. windows can also be used. This OLA process is shown in Figure 3.29.

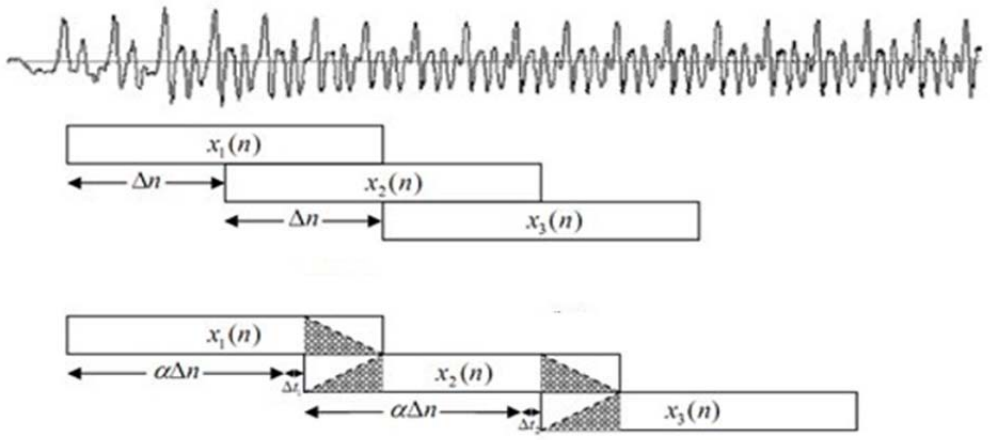


Figure 3.29 OLA and cut and paste

The OLA shown in Figure 1 is expressed as follows;

$$y(m.\alpha\Delta n + j) := \big(1 - f(j)\big)y(m.\alpha\Delta n + j) + f(j)x(m\Delta n + j) \quad ;0 \le j \le L - 1$$
$$y(m.\alpha\Delta n + j) = x(m\Delta n + j) \quad ;L \le j \le N - 1$$

In this equation, f (j) function is folding weights, and L is the number of overlapping samples:

The OLA process forms the basis for many other successful time-scale changing techniques, but because the overlapping parts cannot be similar to each other, there are distortions in the synthesized sound due to the continuity and shift of the pitch period.

### 3.2.4.2. Synchronous Overlap and Add (SOLA)

SOLA is very similar to OLA. The main difference between the two is that SOLA relies on correlation techniques to develop the time extension algorithm. In the SOLA method, the best overlap point should be determined when the blocks are overlapped. In this method, the input signal is divided into overlapping blocks of fixed length, and each block is shifted according to the time scale factor *Sa* constant. The discrete time delay with the highest reciprocal correlation value is then investigated over the "$k_m$" overlap range. At the point where the maximum similarity is found, the overlap blocks are given weight values by the help of fade-in and fade-out functions. Basic SOLA process is shown in Figure 3.30.

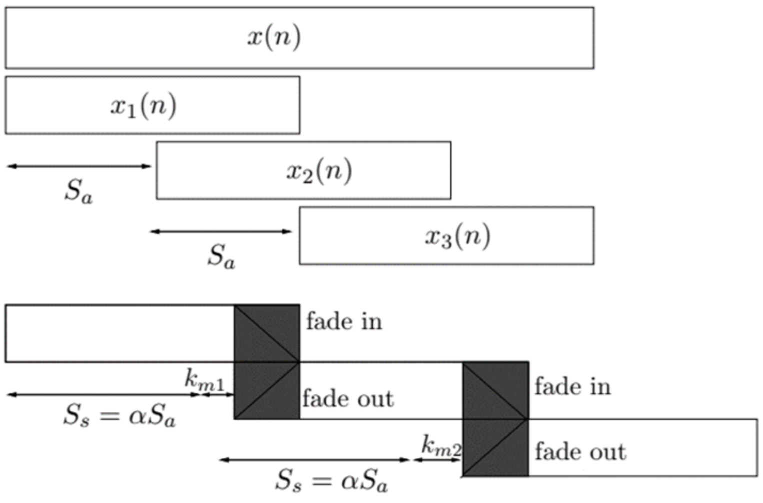


Figure 3.30 SOLA algorithm basic

Since the signal continuity is corrupted due to the change in the distance between consecutive frames in the OLA technique, harmonizing the overlapped consecutive frames can eliminate these problems. For this reason, it is aimed to overlap similar places in SOLA.

In SOLA, the input sound $x$ is divided into N-length frames at $S_a$ sample intervals ($\Delta n = S_a$) (analysis step). In the synthesis sound, the frames are overlapped in the $k$ neighborhood, which provides similarity between the sample intervals $S_z$ ($\alpha\Delta n = S_z$) (synthesis step). Since k values are calculated separately for all frames, the distance between two consecutive synthesis frames (the frame "*m*" and "*m-1*") is calculated as $S_z + k_m - k_{m-1}$. The optimum $k_m$ value is searched in the interval $k_{min} \le k_m \le k_{max}$. To calculate the optimal $k_m$, equations like the lowest value of the difference square mean is used:

$$E_m(k) = \frac{\sum_{j=0}^{L_k-1}(y(mS_s + k + j) - x(mS_a + j))^2}{L_k}$$

In the above equations, $L_k$ is the number of overlapping samples according to k. After selecting the $k_m$ value according to these equations, the time-scale sound synthesis is performed as follows:

$$y(mS_z + j) \coloneqq \big(1 - f(j)\big)y(m.S_z + j) + f(j)x(mS_a + j)\,, \qquad 0 \le j \le L_k - 1$$
$$y(mS_z + j) = x(mS_a + j)\,, \qquad L_k \le j \le N - 1$$

The sign "≔" in the equation above indicates that $y(mS_z + j)$ value will be updated. The new value on the left side of the mark will be updated by also using the old value on the right side of the mark. $f(j)$ is a linearly increasing function and its value is:

$$f(j) = \begin{cases} 0, & j < 1 \\ j/(L_k - 1), & 0 \le j < L_k \\ 1, & j \ge 1 \end{cases}$$

The quality of the sound synthesized by SOLA also depends on the choice of parameters. In the first applications the parameters are chosen as follows: $N$ is 30 ms for speech sound and 40 ms for music sound, $S_a = N/2$, $k_{min} = N/2$, $k_{max} = N/2$ .

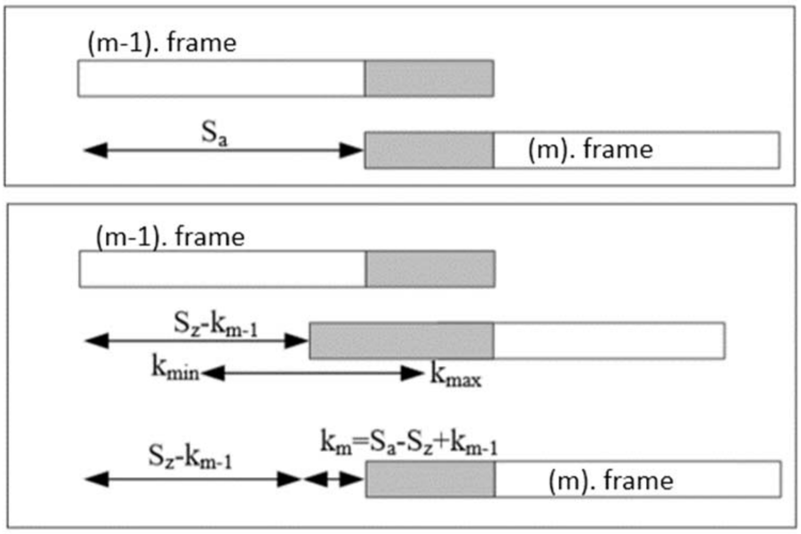


Figure 3.31 Overlap estimation in SOLA operation

First part of the figure: Distance between frames in original voice. Second part: Distance between frames in SOLA synthesis sound

### 3.2.4.3. PSOLA

PSOLA is produced as a variation of the SOLA algorithm and it is based on the hypothesis that the signal is characterized by a tone (e.g., human voice and monophonic musical instruments). A high-quality time-domain modification can be achieved in this technique by applying a synchronized tone to the overlap function. One of the main problems of this method is the difficulty of estimating the base tone period of the signal when the base frequency is lost. The algorithm can be divided into two parts. First, the input signal is analyzed and segmented. In the second part, the segmented signal is combined with the overlapping methods and a tone is added.

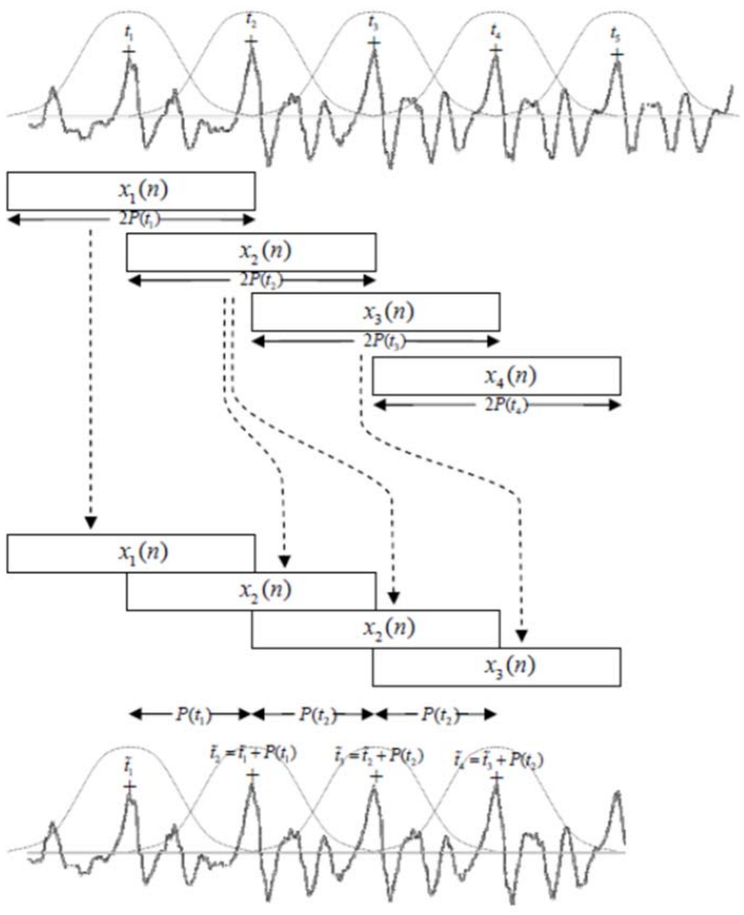


Figure 3.32 PSOLA algorithm (Sanjaume, 2002)

The sound signals (monophonic instrument or speech) have pitches for long-term relationship. PSOLA (Pitch synchronous overlap and add) finds the pitches of the sound signal and divides it into 2 pitch lengths parts and makes overlapping adding according to the detected pitches. The PSOLA algorithm first finds the voiced and unvoiced parts of the analysis sound. It finds and marks the pitch of the sound for the voiced parts, and puts the pitch marks at constant distances for the unvoiced parts.

PSOLA makes pitch detection in time domain. Since the OLA procedure is performed in synchronous pitch, the synthesized sound is of high quality. PSOLA algorithm consists of two parts: analysis and synthesis. In the analysis part, the input sound is divided into frames. In the synthesis part, these frames are combined to synthesize the time scale modified sound.

The first step in the analysis is to put the pitch mark $t_i$ on the maximum points of the periodic parts of the sound and the fixed distance samples of the unvoiced parts. The pitch period $P(t_i)$ is calculated as $P(t_i) = t_{i+1} - t_i$ from the distance between samples marked with time. Then the samples with the input sound pitch marks are divided into pieces so that they are centered. Each piece is windowed with a Hanning window with 2 pitch periods. In synthesis, the acceleration/deceleration factor α determines the length, $\widetilde{P}\,(\tilde{t})$ determines the pitch periods of the synthesis signal.

$$\widetilde{P}\,(\widetilde{t}) = \widetilde{P}\,(\alpha t) = P(t)$$

Synthesis pitch times are calculated from the previous pitch periods as follows:

$$\tilde{t}_{k+1} = \tilde{t}_k + \widetilde{P}\,(t_k) = \tilde{t}_k + P(t_i)$$

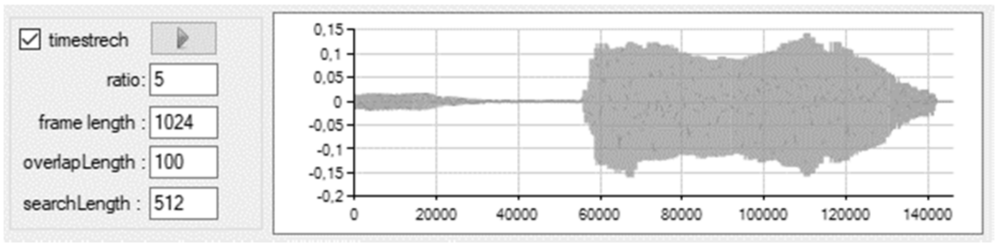
The $\tilde{t}_k$ in the equation shows the $k^{th}$ pitch sign in the synthesized sound and $t_i$ shows the $i^{th}$ pitch sign in the analyzed sound . So the $k^{th}$ part that will it will be added to the synthesis is obtained by overlapping the $i^{th}$ part in analysis. Here, $t_i$ is selected in order to minimize $|\alpha t_i - \tilde{t}_k|$.

In the equation, the acceleration/deceleration factor $\alpha$ is a constant. However, with $\tilde{t} = \int_0^t \alpha(\tau) d\tau_k$, it can be generalized for the non-constant acceleration/deceleration factor $\alpha$. As can be seen from Figure 3.32, according to the above equation some analysis parts for $\alpha > 1$ can be used in more than one overlap adding. Likewise, some parts of analysis may not be used for $\alpha < 1$.

### 3.2.4.4. The Used Algorithm

The algorithm developed within the scope of this thesis is based on SOLA. The algorithm includes three functions. The first function takes the parameters and the sound data shown in Figure 3.31 and processes them. The similar blocks identified using the FindConnectStartIndex function are also copied here one after the other.

```
public double[] Stretch(double[] src, double ratio, int frameLength, int
                                    overlapLength, int searchLength)
    {
      double[] dst = new double[(int)(ratio * src.Length)];
      Array.Copy(src, dst, frameLength);
      int curDstEndIndex = frameLength;
      while (true)
      {
        int srcStartIndex = (int)((double)curDstEndIndex / dst.Length *
                                                    src.Length);
        int connectStartIndex = FindConnectStartIndex(dst, curDstEndIndex, src,
                    srcStartIndex, overlapLength, searchLength);
        for (int t = 0; t < overlapLength; t++)
        {
          if (connectStartIndex + t == dst.Length || srcStartIndex + t ==
                                                              src.Length)
            return dst;
          double a = (double)t / overlapLength;
          dst[connectStartIndex + t] = a * src[srcStartIndex + t] + (1 - a) *
                                                  dst[connectStartIndex + t];
        }
        for (int t = overlapLength; t < frameLength; t++)
        {
          if (connectStartIndex + t == dst.Length || srcStartIndex + t ==
                                                               src.Length)
            return dst;
          dst[connectStartIndex + t] = src[srcStartIndex + t];
        }
        curDstEndIndex = connectStartIndex + frameLength;
      }
        }
```



☑ timestrech
ratio: 5
frame length : 1024
overlapLength : 100
searchLength : 512

Figure 3.33 Parameters sent to the algorithm

Here the *ratio* indicates the extension value of the sound. The sound will be extended longer if this value is chosen large enough. *Frame length* specifies the length of the sound blocks. *Overlap length* refers to the amount of data to be used when comparing two blocks. *Seach length* refers to the amount of sub-blocks that these comparisons will be made in the main block.

The sound data is transferred to the function named Stretch within the array named *src*. An array named *dst* is defined in the function. The size of this array is defined as the product of the *ratio* value with the size of the *src* array. Then the data specified as much as *frameLength* value is copied from the *dst* array to the *src* array.

The following function is used to determine the comparison blocks in the algorithm. The index of the blocks that are closest (similar) to each other in the compared blocks is also determined here.

```
private static int FindConnectStartIndex(double[] dst, int dstEndIndex, double[]
            src, int srcStartIndex, int overlapLength, int searchLength)
    {
       int dstSearchStartIndex = dstEndIndex - overlapLength - searchLength;
       double minDiff = double.MaxValue;
       int minDiffIndex = new int();
       for (int t = 0; t < searchLength; t++)
       {
        double diff = CalcDifference(dst, dstSearchStartIndex + t, src,
                                    srcStartIndex, overlapLength);
        if (diff < minDiff)
        {
          minDiff = diff;
          minDiffIndex = dstSearchStartIndex + t; // 412+t
        }
      }
      return minDiffIndex;
        }
```

These two data were then compared with the specified initial index values. The data from *src* array up to *Overlap length* value is compared with the data blocks from *dst* array specified up to *Overlap length* while making comparisons. The CalcDifference function is used for comparisons. The MSE (Mean Square Error) value of the blocks is determined in each comparison. Thus, the block with the lowest MSE value in the *dst* array is considered to be the closest block to the block in the *src* array.

One of the methods that can be used to measure the level of overlapping of the sound data blocks that are compared, is the Mean Square Error (MSE) method. The MSE helps us to digitize the closeness of the estimated and actual response values of a particular observation. MSE is expressed as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(E_{estimated} - E_{real})^2$$

According to the closeness of the predicted values to the actual values the MSE gets smaller or larger. The minDiffIndex value in the above code holds the initial index of the block with the lowest MSE value in the data array.

The codes where the MSE calculation is performed are as follows.

```
private static double CalcDifference(double[] dst, int dstStartIndex, double[] src,
                                        int srcStartIndex, int overlapLength)
    {
      double sum = 0;
      for (int t = 0; t < overlapLength; t++)
      {
        double d = dst[dstStartIndex + t] - src[srcStartIndex + t];
        sum += d * d;
      }
      return sum;
       }
```

### 3.2.5. Obtaining Triphones from Diphones

In Text to Speech studies, vocalization of two letter syllables consisting of a vowel and a consonant (VC, CV) can be performed without any problem. Because the syllable has already been recorded as a sound file. The operations that can be done for binary syllables are extending or shortening the voiced part of a syllable inside a word.

One of the important issues in enhancing the naturalness of vocalization is the creation of triple syllables (CVC). In forming triple syllables, the method of adding binary syllables to each other (CVC -> CV + VC) is used. For instance, lets process the "bul "syllable.

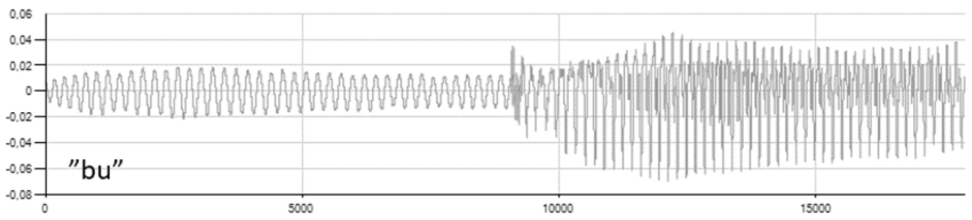


Figure 3.34 "bu" syllable

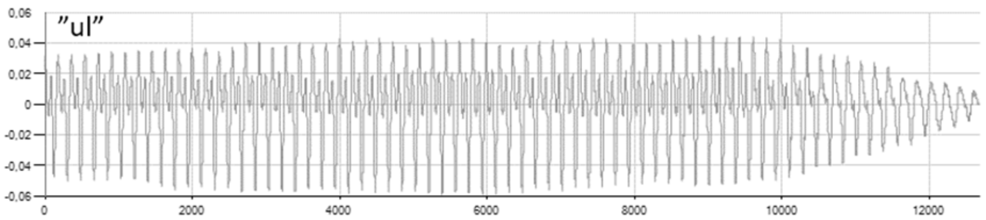


Figure 3.35 “ul” syllable

The “bul” syllable is obtained by overlapping “bu” and “ul” syllables. The voiced part of the “bu” syllable is the last part where the letter “u” exits as seen in Figure 3.34. On the other hand, the voiced and periodic part of the “ul” syllable is the beginning where the letter “u” exists as seen in Figure 3.35. By overlapping the periodic parts of these two syllables, a new sound is tried to be obtained.

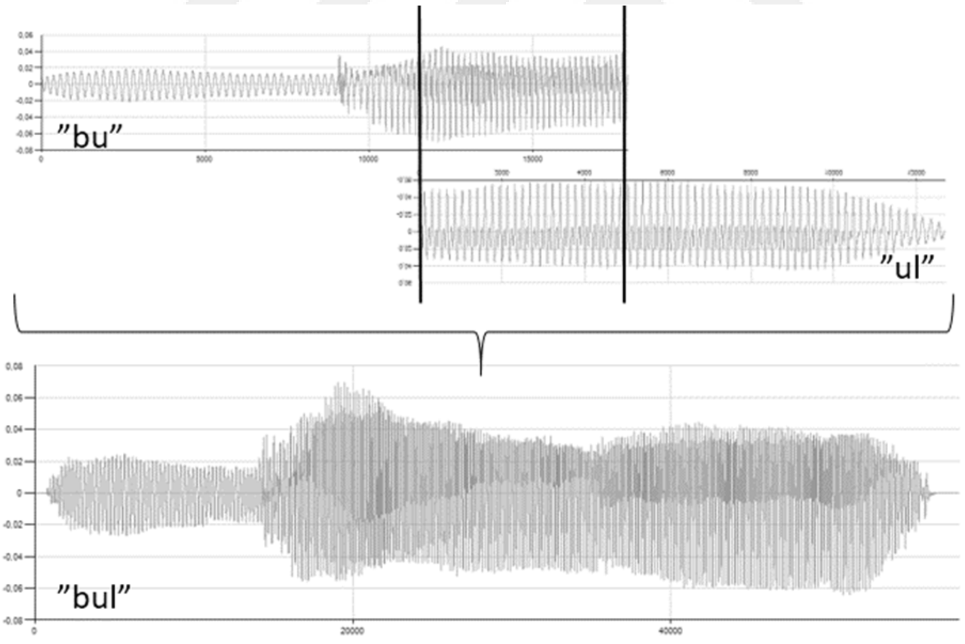


Figure 3.36 Overlapping ”bu” and “ul” syllables

Although the new triple syllable is understandable, there are some interruptions and tone differences in the syllable. Therefore, a number of improvements need to be made.

First, the frequencies of the two syllables that will be combined are tried to be equalized. The syllables “bu” and “ul” are equalized to the same frequency by taking a pitch value specified in the software as a reference. Then, “bu” and “ul” syllables are extended by using the sound extension algorithm developed within the scope of this thesis. The graphical view of the extended syllables is shown in Figure 3.37.

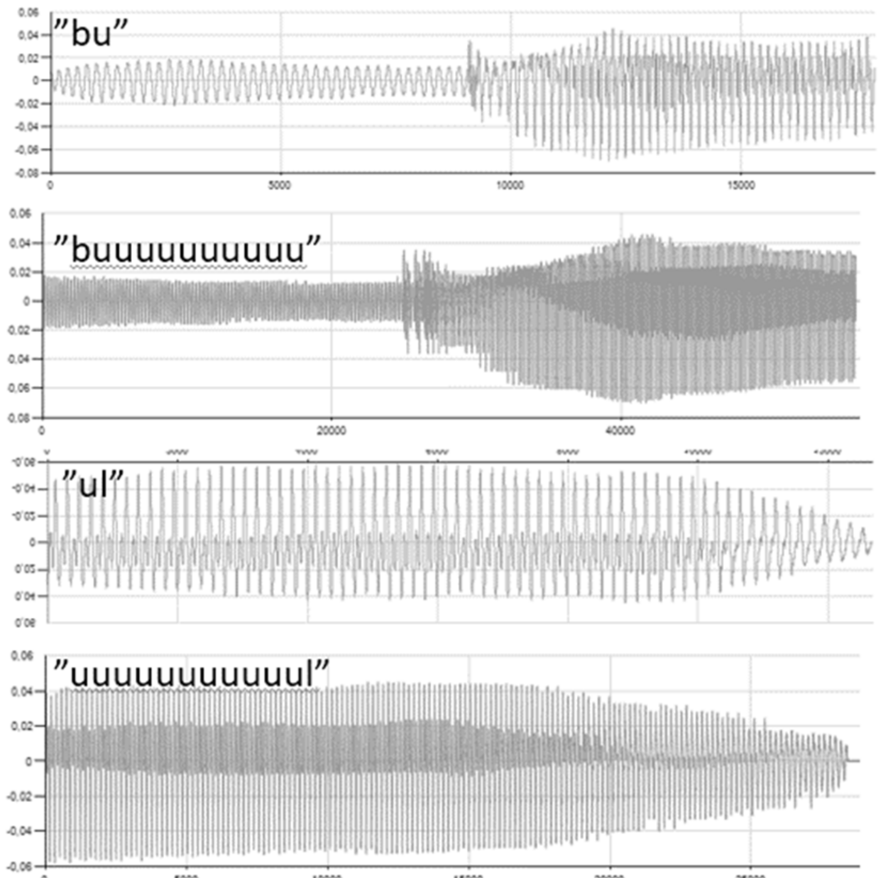


Figure 3.37 Extension of “bu” and “ul” syllables

These extended syllables are overlapped by using the method described above. The new extended sound data is shortened with the same algorithm. Thus, irregularities in the sound data that are caused by overlapping are also eliminated within the software.

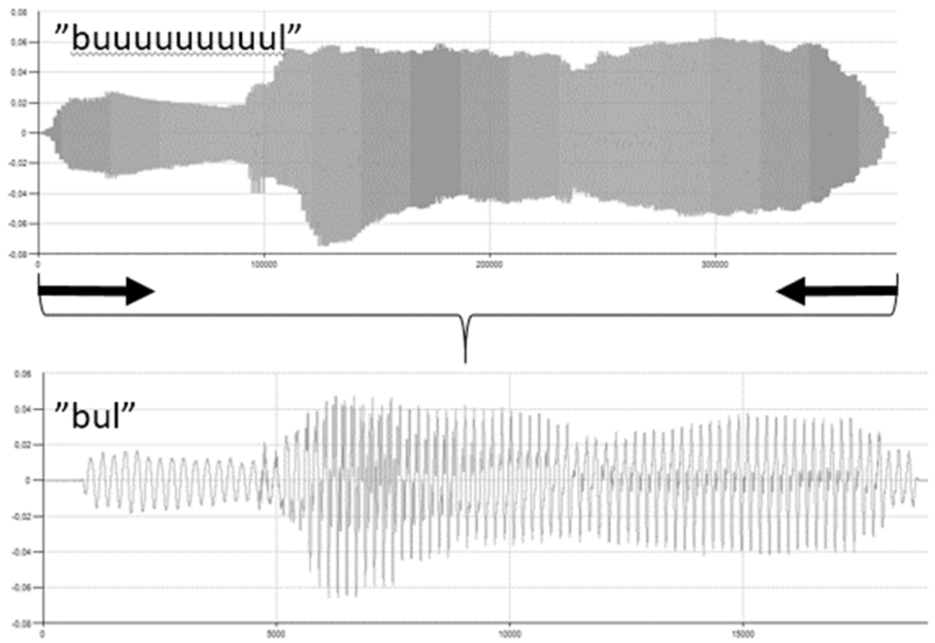


Figure 3.38 Shortening the new syllable formed from extended syllables by using the algorithm

Since unbalanced sound files (with different amplitude) may be present among the automatically generated files, an algorithm is applied on all files in order to avoid sound cracking at the joints. If desired, it is also possible to change the frequency value of the new syllable.

### 3.2.6. Determination of the pitch value of a sound syllable

When recording the sound syllables, it is possible that they have different pitches. Therefore, it is necessary to find a pitch value for syllables that will be combined. It will give information about the cause of the perceived sound transitions between the syllables. Nowadays, it is a popular topic to add a sentiment to the sound or change the sentiment in the sound in text to speech studies. Therefore, it is important to know the pitch values of the sound signals.

Association, in statistics, is the relationship between two random variables whereas correlation is a statistical association. Correlation gives an idea of how close two variables are from each other from having a linear relationship.

In digital signal processing, autocorrelation is the correlation of a signal with delayed copy of itself as a function of delay. Informally, it is the similarity between observations as a function of lag between them. Autocorrelation is like a search procedure in frequency detection. The signal is stepped through sample by sample and a correlation between reference window and the lagged window is performed. The correlation at "lag 0" will be the global maximum because the reference is being compared to the exact copy of the signal, at some point it will begin to increase again, then reach a local maximum again. The pitch is then estimated from the distance between "lag 0" and that first peak.

Autocorrelation is a very powerful tool in signal processing. It comes very handy in finding repeating patterns such as a periodic signal interfered by noise, fundamental frequency hidden in the harmonics etc.

The discrete autocorrelation function $\emptyset$ at lag $\tau$ for a discrete signal x(n) is defined by:

$$\emptyset(\tau) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)x(n-\tau)$$

For a pure tone, the autocorrelation exhibits peaks at lags corresponding to the period and its integral multiples. The peak in the autocorrelation of expression at the lag corresponding to the signal period will be higher than that at the lag values corresponding to multiples of the period. For a musical tone consisting of the fundamental frequency component and several harmonics, one of the peaks due to each of the higher harmonics occurs at the same lag position as that corresponding to the fundamental, in addition to several other integer multiples of the period (sub harmonics) of each harmonic.

Thus, a large peak corresponding to the sum contribution of all spectral components occurs at the period of the fundamental (and higher integral multiples of the period of the fundamental). This property of the autocorrelation makes it very suitable for the pitch tracking of monophonic musical signals. The autocorrelation pitch detection algorithms chooses as the pitch period the lag corresponding to the highest peak within a range of lags. Since the signals are commonly noisy, some extra processing is needed to increase the performance of the autocorrelation.

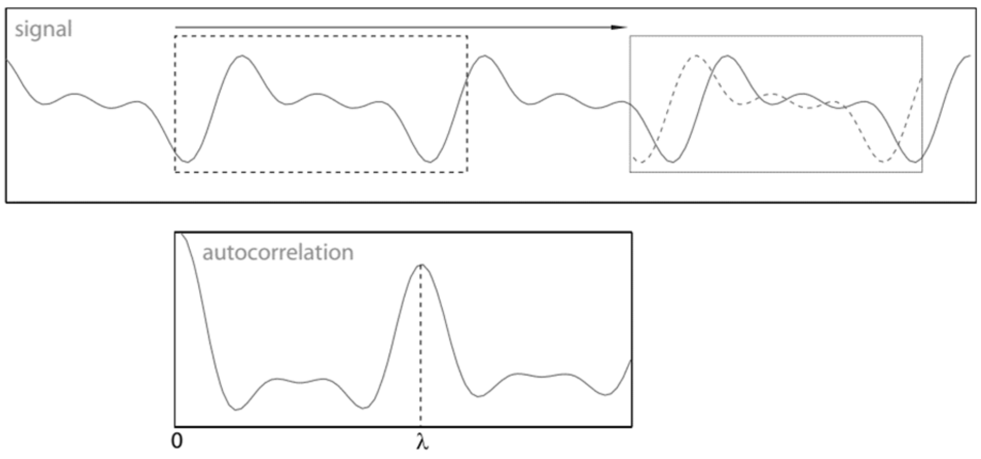


Figure 3.39 Autocorrelation. There is a local maximum when the signal is shifted λ steps

When the shift $\tau$ approaches an integer multiple of the fundamental wavelength the correlation will increase, reaching a local maximum when the signal is shifted exactly an integer number of fundamental wavelengths as can be seen in Figure 3.39.

The developed pitch detection algorithm is given at the appendix.

The resulting frequency can be expressed in music terminology with a note. The note, as a word, is of Latin origin. Note is used for recording purposes. A term for recording sounds. The universal note language and portrait that is in use today was found by Guido d'Arezzo, a priest and musician in Italy in the 11*th* century. The first syllable of each verse of a hymn gives the note its name. After the addition of *si* sound and replacing the *Do* with *UT* word, this sequence has become as: *do (C), re (D), mi (E), fa (F), sol (G), la (A), si (B)*. When the tuning fork was first invented in 1711, the frequency of the pitch La was 423.5 (A423.5). In 1936, the American National Standards Institute designated it as A440. In 1953, the International Standards Institute (ISO) adopted the same frequency (ISO, 1975).

In western music, a single octave interval is divided into 12 intermediate sounds. The frequency of a note is a $2^{\frac{1}{12}}$ multiple of the frequency of the previous note. In this case, if we go 12 steps forward or backward from any sound in an octave region, we reach the sound at twice higher or lower frequency of that sound.

For example, while the A4 note is 440 Hz, the frequency of the sound immediately following it is $440 * 2^{\frac{1}{12}} = 466Hz$. But if we do the calculation in this way, there will be cutting and rounding errors in the product of the numbers after the point. To avoid this type of error, we can use $f_n = f_0 * 2^{\frac{n}{12}}$.

Due to the above correlation, the ratio of the frequencies of two neighboring notes is always $2^{\frac{1}{12}}$.

Consider the two sinusoidal signals $y = sin(at)$ and $g = sin(bt)$ with angular frequencies a and b.

Let's look at the sum of $y + g$. This can be thought of as the simultaneous listening of two sounds at different frequencies.

$$sin(a + b) = sin(a)\,cos(b) + cos(a)\,sin(b)$$
$$sin(a - b) = sin(a)\,cos(b) - cos(a)\,sin(b)$$

$$sin(a+b) + \sin(a-b) = 2\,sin(a)\,cos(b)$$
$$u = a + b \; and \; v = a - b$$
$$a = \frac{u+v}{2} \; and \; b = \frac{u-v}{2}$$
$$sin(u) + sin(v) = 2\,sin(\frac{u+v}{2})\,cos(\frac{u-v}{2})$$

The last statement means that a sound with a frequency at half of the sum of the two note frequencies will undergo amplitude modulation with another sound at frequency half of the difference of these two frequencies. Table 3.7 shows the notes and their frequencies.

The following lines of code indicate which note the sound syllable refers to.

```
private void Find_Closest_Note(double freq, out double closestFreq, out string
                                                                    note_Name)
    {
      string[] Note_Names = { "A", "A#", "B/H", "C", "C#", "D", "D#", "E", "F",
                                                         "F#", "G", "G#" };
      double Tone_Step = Math.Pow(2, 1.0 / 12);
      const double AFreq = 440.0;
      const int Tone_Index_Offset_To_Positives = 120;
      int tone_Index = (int)Math.Round(Math.Log(freq / AFreq, Tone_Step));
      note_Name   =   Note_Names[(Tone_Index_Offset_To_Positives   +
                                      tone_Index) % Note_Names.Length];
      closestFreq = Math.Pow(Tone_Step, tone_Index) * AFreq;
    }
```

Table 3.7 Notes and their frequencies

| Perde | C | C# | D | D# | E | F | F# | G | G# | A | A# | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 16,35 | 17,32 | 18,35 | 19,45 | 20,6 | 21,83 | 23,12 | 24,5 | 25,96 | 27,5 | 29,14 | 30,87 |
| 1 | 32,7 | 34,65 | 36,71 | 38,89 | 41,2 | 43,65 | 46,25 | 49 | 51,91 | 55 | 58,27 | 61,74 |
| 2 | 65,41 | 69,3 | 73,42 | 77,78 | 82,41 | 87,31 | 92,5 | 98 | 103,8 | 110 | 116,5 | 123,5 |
| 3 | 130,8 | 138,6 | 146,8 | 155,6 | 164,8 | 174,6 | 185 | 196 | 207,7 | 220 | 233,1 | 246,9 |
| 4 | 261,6 | 277,2 | 293,7 | 311,1 | 329,6 | 349,2 | 370 | 392 | 415,3 | 440 | 466,2 | 493,9 |
| 5 | 523,3 | 554,4 | 587,3 | 622,3 | 659,3 | 698,5 | 740 | 784 | 830,6 | 880 | 932,3 | 987,8 |
| 6 | 1047 | 1109 | 1175 | 1245 | 1319 | 1397 | 1480 | 1568 | 1661 | 1760 | 1865 | 1976 |
| 7 | 2093 | 2217 | 2349 | 2489 | 2637 | 2794 | 2960 | 3136 | 3322 | 3520 | 3729 | 3951 |
| 8 | 4186 | 4435 | 4699 | 4978 | 5274 | 5588 | 5920 | 6272 | 6645 | 7040 | 7459 | 7902 |

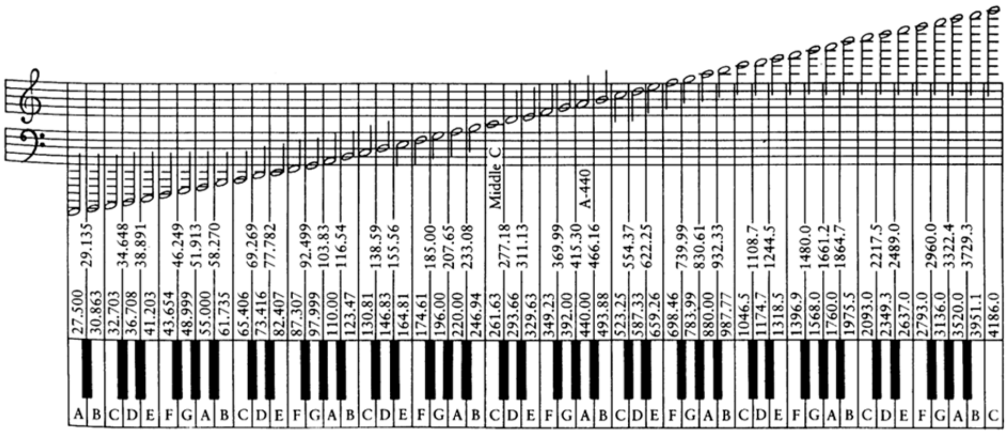


Figure 3.40 Notes and octave intervals on the piano (http://people.math.sc.edu/sharpley)

### 3.2.7. Pitch Shifting

The pitch is closely related to the frequency of a sound. Although the pitch is associated with music and musical notes, it also concerns human voice and other sounds. The pitch of a male voice can be increased to a female voice, or vice versa.

Pitch shifting is the name given to the process that changes or transposes the pitch of a sound. The most practical way to shift the pitch of a syllable is to first determine the pitch frequency of the syllable as described above. The ratio of this value to the pitch value to be shifted is the amount of shift. In this way, the pitch value can be changed by playing with the sound data at the specified shift amount.

The sound syllables combined with this method are brought to the same frequency value and so a harmony is formed in the merged words.

There is a pitch shift factor in the algorithm used. This parameter ranges from 0.5 (one octave down) to 2 (one octave up). If you set this value to 1, there will be no shifting.

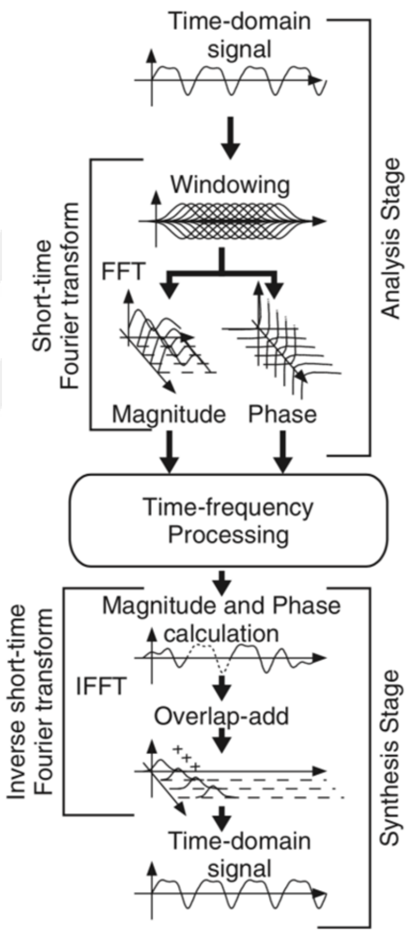

Time-domain signal
Windowing
Short-time Fourier transform
FFT
Magnitude
Phase
Analysis Stage
Time-frequency Processing
Magnitude and Phase calculation
Inverse short-time Fourier transform
IFFT
Overlap-add
Time-domain signal
Synthesis Stage

Figure 3.41 Pitch shifting algorithm (Zhang)

The pitch shifting algorithm is shown in Figure 3.41. The array containing the syllable sound signal is divided into frames and a window function is implemented. Then by applying FFT passed to the frequency domain. Here by making changes in frequency data, the data is analyzed by returning to the time domain again and again with IFFT.

The pitch shifting can be separated into an analysis part and a synthesis part, with an optional processing stage in between. The basis for the pitch shifting is the Short-Term Fourier Transform (STFT), in which a vector of sound samples is divided into multiple N-sized blocks referred to as frames. Frames are usually overlapped and windowed.

Fourier transform is originally based on calculating the unlimited signal, however only a small number of samples is processed each time in STFT. Windowing every short-term signal is to compensate this problem. We adopt four window functions: Hamming, Hanning, triangular and rectangular windows. Each frame is then decomposed into a set of N complex numbers that represent a set of sinusoids (known as bins). Decomposition is accomplished using the Fourier transform. w(n) is the window function, also shown in this formula:

$$X(m) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)\, e^{-i2\pi nm/N} w(n), \qquad m = 0,1,\ldots,N-1$$

Each bin represents a single sinusoid. The frequency of a particular bin can be obtained from its index *m* and the sampling rate *sr* :

$$f(m)_{bin} = \frac{m \times sr}{N}$$

The real and imaginary parts of a bin’s complex number can be manipulated to obtain the magnitude and phase of each sinusoid :

$$|X(m)| = \sqrt{X_R(m)^2 + X_I(m)^2}$$

$$\angle X(m) = \tan^{-1}\frac{X_I(m)}{X_R(m)}, \quad where \ \tan^{-1}(x) \in [-\pi, \pi]$$

Each sinusoid is completely described by these three pieces of information. The frequency of each bin is fixed, at equally-spaced intervals up to the *Nyquist* rate (half the sampling rate). Sinusoids that fall between bin frequencies have their energy distributed across the surrounding bins. In other words, a single bin frequency is usually only an approximation of the true frequency of the represented sinusoid. One of the main innovations of the phase vocoder is that the difference in phases of a particular bin across two successive frames can be used to derive an adjustment factor:

$$\emptyset(m) = [\angle X(m) - \angle X(m-1)]_{-\pi}^{\pi} \times (sr/2\pi N)$$

This can then be added to the original approximate frequency to obtain an improved frequency estimate:

$$f(m)_{instantaneous} = f(m)_{bin} + \phi(m)$$

Additionally, we use FFT-shift and zero-padding for STFT. FFT-shift is to exchange the first half samples and the rest half samples before FFT calculation. Zeropadding shown in Figure 3.42 is to get more precise bin frequencies. The FFT size becomes 8 larger than frame size and the added samples have 0's for their values.

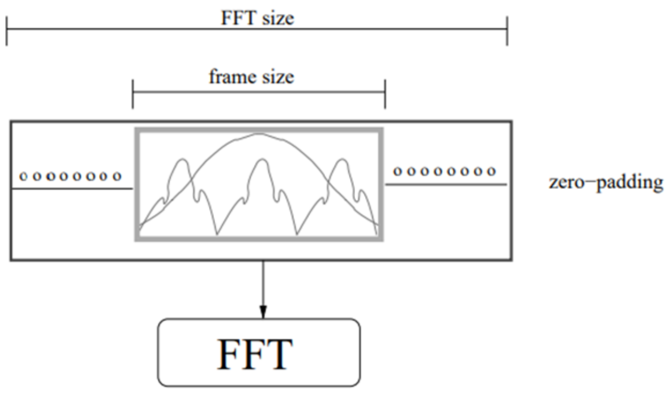


Figure 3.42 Fourier transform with zero-padding

The analysis algorithm in the pitch shifting is following steps;

1. Take frame(N) size samples from input sound. Store them into a frame buffer.
2. Choose a proper window and window the buffer
3. Zero-pad the buffer
4. FFT-shift the zero-padded buffer.
5. Decompose the buffer by Fourier transform
6. Calculate and store magnitude and phase (output from analysis that is used in resynthesis).
7. Go back to step 1 and start taking frame size sample located at hop size distant from the first sample of the previous loop.

The synthesis stage is basically the analysis stage in reverse: a list of sinusoids in complex number representations are converted resynthesized using an inverse Fourier transform (IFFT) into a sound signal. While resynthesis can be accomplished using other techniques, such as direct additive synthesis, the IFFT method is much faster, and with today’s computers can easily be done in real time.

If the output of the analysis stage is used directly for resynthesis, the result will be the original sound file with no perceptible difference from the original (and we would have nothing more than a very computationally-expensive way to copy sound information). However, one of the strengths of the phase vocoder is the possibility of introducing additional processing between the analysis and synthesis stages.

Processing and resynthesis algorithm for pitch-shifting is:

1. Unwrap phases which values become between π and -π (Equation 2.5).
2. Multiply the phases by pitch-shift ratio (processing).
3. Calculate complex numbers from both a) the original magnitudes from analysis and b) modified phases from step 1 (resynthesis from this step and below).
4. Compose the complex numbers by Inverse FFT, and store modified sound samples in a buffer.
5. FFT-shift the buffer.
6. Window the buffer.
7. Interpolate the sound samples in the buffer.
8. Place the buffer in a new empty sound data at every analysis hop size interval.
9. Go back to step 1.

Because phase is wrapped after 2π (360 degree), phase unwrapping process is necessary to get linearly continuous phase.

### 3.2.8. Equalizing Energy

The amplitude of the speech signals varies with time. Short Term Energy is higher in areas where there is speech than in areas where there is no speech. This gives us important clues about the speech zones. The formula of Short Term Energy is as follows:

$$E(n) = \sum_{i=1}^{N} X_n^2\,(i)$$

Where; N denotes the length of the sound frame, *X(i)* the original speech signal and *E(n)* the energy of the sound frame. As can be seen, the energy of a sound frame can be calculated by taking the sum of the squares of each sound sample.

In order to change the energy of the speech piece, all samples are multiplied by the square root of the energy ratio

$$E_1 = \frac{1}{K}\sum_{n=1}^{K} d_1^2\,(end - n)$$

$$E_2 = \frac{1}{K}\sum_{n=1}^{K} d_2^2\,(n)$$

$$\propto = \sqrt{\frac{E_1}{E_2}} \qquad s_2 = \propto .\, d_2$$

Where; $E_1$ is the energy of the last *K* samples of the first diphone, and $E_2$ is the energy of the first K samples of the following diphone. α (by taking the square root of the energy ratio) is multiplied by the second diphone and a new diphone with equal energy is obtained.

# 4. RESULTS AND DISCUSSION

In this study, a text to speech system based on additive synthesis technique, which uses syllable as the length of the sound unit, is realized. The following features are intended when implementing a text synthesizer: the quality of the generated speech output should be high, the memory requirement should be low, the complexity of the application should not be too high, and the computational speed should be high. In the designed system, rules related to merging the recorded syllables have been established. These rules are formed depending on the forms of sounds at the junction of syllables (breath, breathless, explosive, etc.) or production places of the sounds. These rules are then added to the merging software. A syllable database is created and this word synthesizer software is tested on it. The system generates syllables from the text information it receives at the input. Then it begins to combine these syllables. At this stage, the rules determined according to the types of sounds are applied to the junction points of syllables and naturalness (similar to waveforms in real sound files) is tried to be created.

In order to evaluate the success of the text to speech system presented in this thesis, 5-sentence intelligibility tests are conducted. Ten volunteers are included in these tests whose mother tongue was Turkish. The tests were conducted for each listener at separate times. The audience is asked to assess each sentence they listened for their intelligibility according to the scores given in Table 4.1.

Table 4.1 Scoring system

| **Score** | **Intelligibility Level** |
|---|---|
| 1 | Good |
| 2 | Medium |
| 3 | Bad |

The audience was asked to evaluate the intelligibility according to how easily they could perceive the words in the sentences they listened to. The intelligibility scores of the audience are averaged separately for each sentence. Then, the overall intelligibility scores and success percentages of the system are obtained by taking the averages of the intelligibility scores obtained separately for all sentences. The test results of the Turkish text to speech system developed in the scope of this thesis is shown in Table 4.2.

Table 4.2 Intelligibility assessment of the system

| **Sentences** | **Intelligibility Percentage** |
|---|---|
| Sentence 1 | 75 |
| Sentence 2 | 65 |
| Sentence 3 | 65 |
| Sentence 4 | 70 |
| Sentence 5 | 80 |
| Sentence 6 | 65 |
| ***Average*** | 70 |

According to the results of the intelligibility assessment tests, it can be said that the text to speech system developed in the scope of this thesis has an average of %70 success. The system is developed by focusing on creating intelligible and acoustically natural syllables. Because of this, the system is weak in terms of intonation of sentences according to the semantic content, and vocalization of words with appropriate emphasis and appropriate rhythms. However, due to the successful implementation of syllable merging, the generated speech is understandable.

# 5. CONCLUSIONS AND SUGGESTIONS

Although there are several studies that have been conducted on Text to Speech Synthesis (TTS) for different languages, there have not been many studies on additive languages such as Turkish. Since Turkish is an additive language many words and phrases can be produced from a root word, and so different sound changes occur. This situation causes difficulties in the speech synthesis process from Turkish text. For example, tens of words can be derived from a word. Therefore, the number of words in the language can be increased easily. For example; *A*, *Ada*, *Adana*, *Adanalı*, *Adanalıyık*, *Adanalılaşmak* and so on.

The quality of a speech synthesizer is assessed by the similarity and intelligibility of speech to human voice. Speech synthesis systems are widely used today and their application areas are increasing day by day. Artificial intelligence studies, virtual assistants in various occupational groups, assistance systems for visually impaired people, multimedia devices, navigation applications, consumer electronics products, telecommunications systems, and so on can be given as an example to main text to speech usage areas.

In most of the studies, because the existing sound files were digitally processed, a robotic sound could be presented to the audience instead of a real human voice. The new TTS systems, which were designed for foreign languages and prepared with great investments by commercial firms, started to be more successful in naturalness and started to add Turkish to their applications in 2009, too.

In the existing TTS systems developed for academic or commercial purposes, differences in pronunciation, reading habits in numeric statements and voice changes in punctuations and abbreviations used in Turkish are not taken into consideration. In other words, the Turkish text taken as an input is not analyzed semantically, but is converted to speech as it is written. In this study, the method of synthesizing triple syllables from binary syllables is tried. It is observed that it is easier to combine the sounds with this method and there is not much need for digital processing. While developing the application on the subject, instead of vocalizing all the words in Turkish, a sample text was studied and it was aimed to develop solutions to the problems experienced in Turkish.

As a result, the standardization method prepared will be enriched with each passing day by means of solution methods that will be produced against the exceptions to be determined and so more accurate conversions will be obtained. Studies on emphasis, intonation, homogenity and sentiment will contribute greatly to the completion of the text to speech synthesis system with the closest intended real human voice.

## REFERENCES

*A hybrid model for text-to-speech synthesis.* Fabio Violaro, Olivier Böeffard. 1998. 5, s.l. : IEEE Transactions on Speech and Audio Processing, 1998, Vol. 6.

Ann K. Syrdal, Raymond W. Bennett, Steven L. Greenspan. 1994. *Applied Speech Technology.* s.l. : CRC Press, 1994. 0849394562, 9780849394560.

Arık, Güray. 2011. Görme engelliler için bilgisayar kullanımının etkinleştirilmesi, erişilebilirlik ve bir türkçe hece tabanlı konuşma sentezleme sisteminin geliştirilmesi . *Yüksek Lisans Tezi.* s.l. : Gazi Üniversitesi , 2011.

Bicil, Yücel. 2010. *Türkçe metinden konuşma sentezleme, Yüksek Lisans Tezi.* s.l. : Sakarya Üniversitesi, 2010.

Coşkun, Prof. Dr. Mustafa Volkan. 2008. *Türkçenin Ses Bilgisi.* İstanbul : Bilge Kültür Sanat, 2008. ISBN:978-605-9241-04-5.

David M. Howard, Damian T. Murphy. 2007. *Voice Science, Acoustics, and Recording.* s.l. : Plural Publishing, 2007. 1597568260, 9781597568265.

Demir, Cansel. 2009. *Konuşma tanıma sentezleme sistemlerinin okul öncesi dönem yabancı dil eğitiminde kullanılması , Yüksek Lisans Tezi.* s.l. : Gazi Üniversitesi , 2009.

Erdemir, Cavit. 2010. *Türkçe metin seslendirme için doğal konuşma sentezleme, Yüksek Lisans Tezi.* İstanbul : İstanbul Üniversitesi, 2010.

Ertürk, Sarp. 2005. *Sayısal İşaret İşleme.* İstanbul : Birsen , 2005. 309-6.

Fant, Gunnar. 1970. *Acoustic Theory of Speech Production.* s.l. : Walter de Gruyter, 1970. 9027916004, 9789027916006.

Furui, Sadaoki. 2018. *Digital Speech Processing: Synthesis, and Recognition.* s.l. : CRC Press, 2018. 1351990926, 9781351990929.

Gazi, Orhan. 2011. *Sinyaller ve Sistemler.* Ankara : Seçkin, 2011. 978-975-02-1633-6.

Görmez, Zeliha. 2009. Implementation of a text-to-speech system with machine learning. [book auth.] Zeliha GÖRMEZ. *M.Sc. Thesis.* İstanbul : Fatih University, 2009.

Güldalı, Kenan. 2009. Türkçe metin seslendirme. [book auth.] Kenan GÜLDALI. *Yüksek Lisans Tezi.* İstanbul : İstanbul Teknik Üniversitesi, 2009.

HyperPhysics. *Sound Waves in Air.* [Online] [Cited: 08 18, 2019.] http://hyperphysics.phy-astr.gsu.edu/hbase/Sound/tralon.html.

ISO. 1975. Acoustics -- Standard tuning frequency (Standard musical pitch). *International Organization for Standardization.* [Online] 1 1975. [Cited: Temmuz 1, 2019.] https://www.iso.org/standard/3601.html.

Kahrs, Mark and Brandenburg, Karlheinz. 2002. *Applicatıons of digital signal processing to audio and acoustics.* New York : Kluwer Academic, 2002. 0-7923-8130-0.

Kutlugün, Mehmet Ali. 2017. Gözetimli makine öğrenmesi yoluyla türe göre metinden ses sentezleme. [book auth.] Mehmet Ali KUTLUGÜN. *M.Sc. Thesis.* İstanbul : İstanbul Sabahattin Zaim Üniversitesi, 2017.

Learning, Lumen. University Physics Volume 1. [Online] Lumen Learning.[Cited: 08 17, 2019.] https://courses.lumenlearning.com/suny-osuniversityphysics/chapter/17-5-sources-of-musical-sound/.

Lee, Edward Ashford and Varaiya, Pravin. 2011. *Structure and Interpretation of Signals and Systems.* Boston : Addison-Wesley, 2011. 0-201-74551-8.

Li Deng, Douglas O'Shaughnessy. 2003. *Speech Processing: A Dynamic and Optimization-Oriented Approach.* s.l. : CRC Press, 2003. 0824740408, 9780824740405.

Martin, J.A.M. 2012. *Voice, Speech, and Language in the Child: Development and Disorder.* s.l. : Springer Science & Business Media, 2012. 3709170427, 9783709170427.

Morris, L. Robert. January 1989. Special Feature a Pc-Based Digital Speech Spectrograph. *IEEE Micro.* January 1989, Vols. 68 - 85 .

Özen, Şifa Serdar. 2002. *Türkçe metinden konuşma sentezleme, Yüksek Lisans Tezi.* s.l. : Hacettepe Üniversitesi , 2002.

Özsoy, A. Sumru. 2004. *Türkçe'nin Yapısı -1 Sesbilim.* İstanbul : Boğaziçi Üniversitesi, 2004. ISBN:975-518-227-6.

Öztürk, Özlem. 2005. *Modeling phoneme durations and fundamental frequency contours in Turkish speech , PhD.* Ankara : Middle East Technical University, 2005.

Parker, Barry. 2015. *Güçlü Titreşimler Müziğin Fiziği.* Ankara : Tübitak, 2015. ISBN 978-975-403-986-3.

Parker, Michael. 2010. *Digital Signal Processing 101.* United Kingdom : Newnes, 2010. 978-1856179218.

Perez-Meana, Hector. 2007. *Advances in Audio and Speech Signal Processing: Technologies and Applications .* London : Idea Group Publishing, 2007. ISBN 978-1-59904-134-6.

Rabiner, Lawrence R. and Schafer, Ronald W. 2007. *Introduction to Digital Speech Processing.* Hanover : now Publishers, 2007. 978-1-60198-070-0.

Richard Lunchsinger, Godfrey E. Arnold. 1967. *Voice-Speech-Language Clinical Communicology, Its Physicology and Patology,.* s.l. : Wadsworth Publishing Company California, 1967.

Sak, Haşim. 2004. A corpus-based concatenative speech synthesis system for Turkish. *M.Sc. Thesis.* İstanbul : Boğaziçi University, 2004.

Sanjaume, Jordi Bonada. 2002. Audio Time-Scale Modification. *Ph. D. Program.* Barcelona : Universitat Pompeu Fabra, 2002.

sapp.org. WAVE PCM soundfile format. [Online] [Cited: 08 16, 2019.] http://soundfile.sapp.org/doc/WaveFormat/.

Schroeder, Manfred R. 2004. *Computer Speech: Recognition, Compression, Synthesis.* s.l. : Springer, 2004. 978-3540212676.

Sharpley, Professor Robert. MATH 750. *MATH 750.* [Online] [Cited: 08 14, 2018.] http://people.math.sc.edu/sharpley/math750/MathMusic.pdf.

Tatham, Mark and Morton, Katherine. 2005. *Developments ın speech synthesıs .* England : John Wiley & Sons, Ltd., 2005. ISBN 0-470-85538-X.

Taylor, Paul. 2009. *Text-to-Speech Synthesis.* Cambridge : Cambridge University Press, 2009. 978-0-521-89927-7.

Taylor, Paul, Caley, Richard and Zen, Heiga. 2014. The Festival speech synthesis system: system documentation. *http://www.cstr.ed.ac.uk/projects/festival/.* [Online] 2014. [Cited: Temmuz 1, 2019.] http://www.cstr.ed.ac.uk/projects/festival/.

TDK, Türk Dil Kurumu. 2000. *Türkçe İmla Kılavuzu.* 2000.

Uslu, İbrahim Baran. 2012. Konuşma işleme ve Türkçenin dilbilimsel özelliklerini kullanarak metinden doğal konuşma sentezleme. [book auth.] İbrahim Baran USLU. *Doktora Tezi.* Ankara : Ankara Üniversitesi, 2012.

Ünaldı, İlker. 2007. Taşınabilir cihazlar için Türkçe metinden konuşma sentezleme sistemi. *Yüksek Lisans Tezi.* Ankara : Hacettepe Üniversitesi, 2007.

Vural, Esra. 2003. A Prosodic Turkish text-to-speech synthesizer. *M.Sc. Thesis.* s.l. : Sabancı University, 2003.

Wikipedia. Wikipedia. *https://en.wikipedia.org/wiki/WAV.* [Online] [Cited: Haziran 1, 2019.] https://en.wikipedia.org/wiki/WAV.

Zeren, Ayhan. 1995. *Müzik Fiziği.* İstanbul : Pan Yayıncılık, 1995. ISBN 975-7652-46-6.

Zhang, Jingjie. Echoing Harmonics. *Echoing Harmonics.* [Online] [Cited: 08 10, 2019.] https://ccrma.stanford.edu/~jingjiez/portfolio/echoing-harmonics/index.html.

## BIOGRAPHY

He was born in Kavak village of Sivas, Şarkışla district. He spent his childhood in a mountain village in Mazgirt district of Tunceli (Dersim). Since his father was a teacher, he completed his primary, secondary and high school education in different cities. He graduated from Trakya University Computer Engineering Department. He has been working as an Instructor at Çukurova University Department of Informatics since 2003.

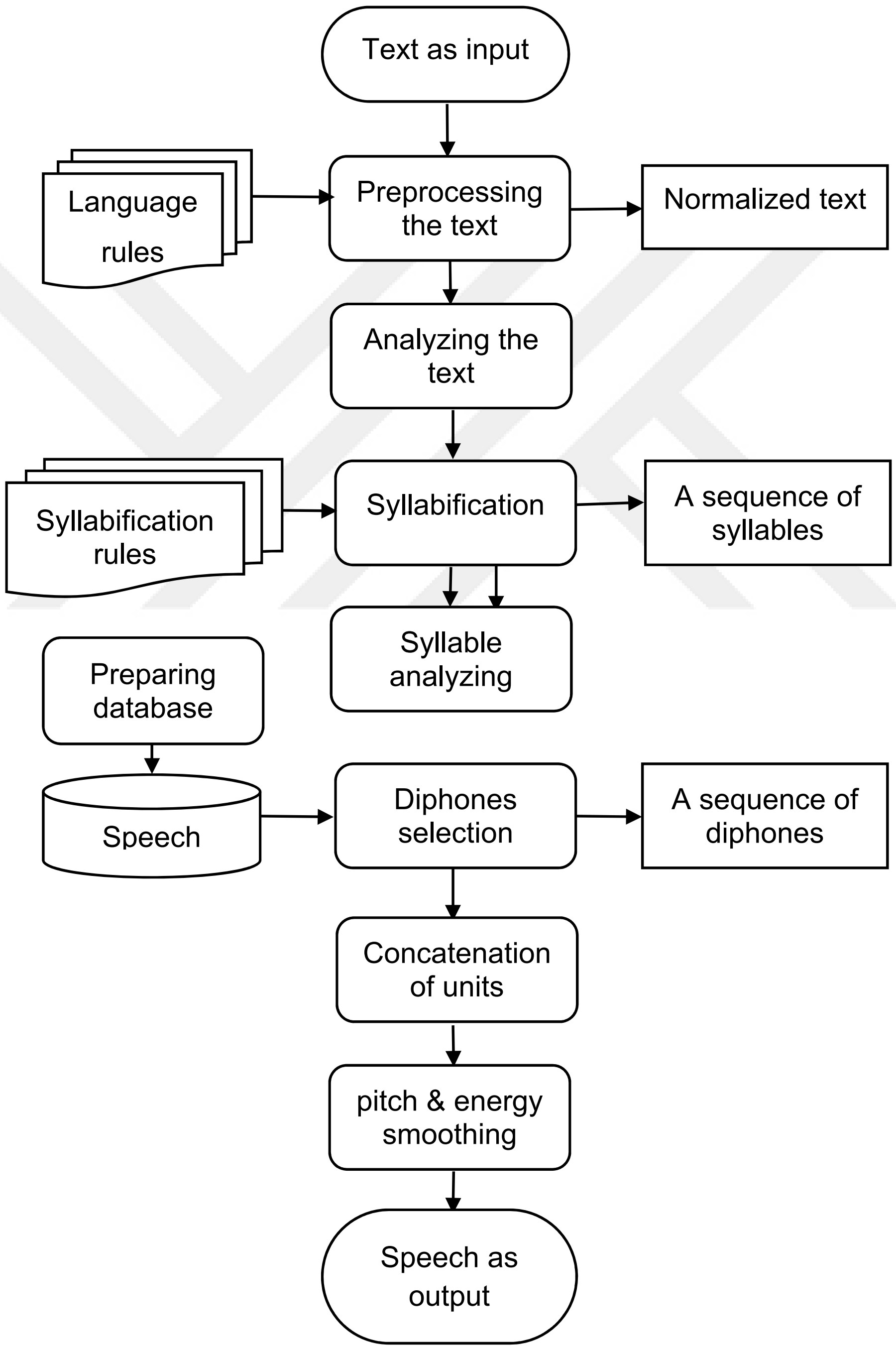
He received his master's degree from Çukurova University, Institute of Science, Department of Agricultural Machinery and Technologies Engineering. He continues his doctoral studies in the same department.

He is married and has 4 children. He is interested in IT, new technologies, music and basketball. In addition to his academic studies, he is a member of Turkey Informatics Association and TMMO Chamber of Computer Engineers Adana representative. He is involved in the establishment of an IT cooperative initiative that will be the first in our country.

# APPENDICES

## APPENDIX 1

TTS System Architecture



## APPENDIX 2

```
public class TimeStretchClass
{
    public double[] Stretch(double[] src, double ratio, int frameLength, int
                                                      overlapLength, int searchLength)
    {
        double[] dst = new double[(int)(ratio * src.Length)];
        Array.Copy(src, dst, frameLength);
        int curDstEndIndex = frameLength;
        while (true)
        {
            int srcStartIndex = (int)((double)curDstEndIndex / dst.Length *
                                                                    src.Length);
            int connectStartIndex = FindConnectStartIndex(dst, curDstEndIndex, src,
                                 srcStartIndex, overlapLength, searchLength);
            for (int t = 0; t < overlapLength; t++)
            {
                if (connectStartIndex + t == dst.Length || srcStartIndex + t ==
                                                      src.Length)
                    return dst;
                double a = (double)t / overlapLength;
                dst[connectStartIndex + t] = a * src[srcStartIndex + t] + (1 - a) *
                                                   dst[connectStartIndex + t];
            }
            for (int t = overlapLength; t < frameLength; t++)
            {
                if (connectStartIndex + t == dst.Length || srcStartIndex + t ==
                                                                  src.Length)
                    return dst;
                dst[connectStartIndex + t] = src[srcStartIndex + t];
            }
            curDstEndIndex = connectStartIndex + frameLength;
        }
    }
    private static double CalcDifference(double[] dst, int dstStartIndex, double[]
                                        src, int srcStartIndex, int overlapLength)
    {
        double sum = 0;
        for (int t = 0; t < overlapLength; t++)
        {
            double d = dst[dstStartIndex + t] - src[srcStartIndex + t];
            sum += d * d;
        }
        return sum;
    }
    private static int FindConnectStartIndex(double[] dst, int dstEndIndex,
                            double[] src, int srcStartIndex, int overlapLength, int
                            searchLength)
    {
        int dstSearchStartIndex = dstEndIndex - overlapLength - searchLength;
        double minDiff = double.MaxValue;
        int minDiffIndex = new int();
        for (int t = 0; t < searchLength; t++)
        {
            double diff = CalcDifference(dst, dstSearchStartIndex + t, src,
                                                   srcStartIndex, overlapLength);
            if (diff < minDiff)
```

```
                {
                    minDiff = diff;
                    minDiffIndex = dstSearchStartIndex + t;
                }
            }
            return minDiffIndex;
        }
    }
```

**APPENDIX 3**

```
public class Pitch_Shifter
  {
    private static int MAXFRAMELENGTH = 44100;
    private static double[] g_In_FIFO = new double[MAXFRAMELENGTH];
    private static double[] g_Out_FIFO = new double[MAXFRAMELENGTH];
    private static double[] g_FFT_worksp = new double[2 *
                                                   MAXFRAMELENGTH];
    private static double[] g_Last_Phase = new double[MAXFRAMELENGTH /
                                                                  2 + 1];
    private static double[] g_Sum_Phase = new double[MAXFRAMELENGTH /
                                                                    2 + 1];
    private static double[] g_Output_Accum = new double[2 *
                                                         MAXFRAMELENGTH];
    private static double[] g_Ana_Freq = new double[MAXFRAMELENGTH];
    private static double[] g_Ana_Magn = new double[MAXFRAMELENGTH];
    private static double[] g_Syn_Freq = new double[MAXFRAMELENGTH];
    private static double[] g_Syn_Magn = new double[MAXFRAMELENGTH];
    private static long g_Rover;
    #region Public Static  Methods
    public static double[] Pitch_Shift(double pitch_Shift, long
             num_Samps_To_Process, double sample_Rate, double[] in_data)
    {
      return Pitch_Shift(pitch_Shift, num_Samps_To_Process, (long)2048,
                     (long)10, sample_Rate, in_data);
    }
    public static double[] Pitch_Shift(double pitch_Shift, long
                  num_Samps_To_Process, long fft_FrameSize, long osamp,
                  double sample_Rate, double[] indata)
```

```
{
  double _magn, _phase, _tmp, _window, _real, _imag;
  double freq_Per_Bin, expct;
  long i, k, qpd, index, inFifoLatency, stepSize, fft_FrameSize2;
  double[] outdata = indata;
  fft_FrameSize2 = fft_FrameSize / 2;
  stepSize = fft_FrameSize / osamp;
  freq_Per_Bin = sample_Rate / (double)fft_FrameSize;
  expct = 2.0 * Math.PI * (double)stepSize / (double)fft_FrameSize;
  inFifoLatency = fft_FrameSize - stepSize;
  if (g_Rover == 0) g_Rover = inFifoLatency;
  for (i = 0; i < num_Samps_To_Process; i++)
  {
    g_In_FIFO[g_Rover] = indata[i];
    outdata[i] = g_Out_FIFO[g_Rover - inFifoLatency];
    g_Rover++;
    if (g_Rover >= fft_FrameSize)
    {
      g_Rover = inFifoLatency;
      for (k = 0; k < fft_FrameSize; k++)
      {
        _window = -.5 * Math.Cos(2.0 * Math.PI * (double)k /
              (double)fft_FrameSize) + .5;
        g_FFT_worksp[2 * k] = (double)(g_In_FIFO[k] * _window);
        g_FFT_worksp[2 * k + 1] = 0.0F;
      }
      /* **************** ANALYSIS ***************** */
      Short_Time_Fourier_Transform(g_FFT_worksp, fft_FrameSize, -1);
      for (k = 0; k <= fft_FrameSize2; k++)
```

```
{
   _real = g_FFT_worksp[2 * k];
   _imag = g_FFT_worksp[2 * k + 1];
   _magn = 2.0 * Math.Sqrt(_real * _real + _imag * _imag);
   _phase = Math.Atan2(_imag, _real);
   _tmp = _phase - g_Last_Phase[k];
   g_Last_Phase[k] = (float)_phase;
   _tmp -= (double)k * expct;
   qpd = (long)(_tmp / Math.PI);
   if (qpd >= 0) qpd += qpd & 1;
   else qpd -= qpd & 1;
   _tmp -= Math.PI * (double)qpd;
   _tmp = osamp * _tmp / (2.0 * Math.PI);
   _tmp = (double)k * freq_Per_Bin + _tmp * freq_Per_Bin;
   g_Ana_Magn[k] = (double)_magn;
   g_Ana_Freq[k] = (double)_tmp;
}
/* **************** PROCESSING ****************** */
for (int zero = 0; zero < fft_FrameSize; zero++)
{
   g_Syn_Magn[zero] = 0;
   g_Syn_Freq[zero] = 0;
}
for (k = 0; k <= fft_FrameSize2; k++)
{
   index = (long)(k * pitch_Shift);
   if (index <= fft_FrameSize2)
   {
        g_Syn_Magn[index] += g_Ana_Magn[k];
        g_Syn_Freq[index] = g_Ana_Freq[k] * pitch_Shift;
    }
}
/* **************** SYNTHESIS ***************** */
for (k = 0; k <= fft_FrameSize2; k++)
{
    _magn = g_Syn_Magn[k];
    _tmp = g_Syn_Freq[k];
    _tmp -= (double)k * freq_Per_Bin;
    _tmp /= freq_Per_Bin;
    _tmp = 2.0 * Math.PI * _tmp / osamp;
    _tmp += (double)k * expct;
    g_Sum_Phase[k] += (float)_tmp;
    _phase = g_Sum_Phase[k];
    g_FFT_worksp[2 * k] = (float)(_magn * Math.Cos(_phase));
    g_FFT_worksp[2 * k + 1] = (float)(_magn * Math.Sin(_phase));
}

for (k = fft_FrameSize + 2; k < 2 * fft_FrameSize; k++)
                            g_FFT_worksp[k] = 0.0F;
Short_Time_Fourier_Transform(g_FFT_worksp, fft_FrameSize, 1);
for (k = 0; k < fft_FrameSize; k++)
{
    _window = -.5 * Math.Cos(2.0 * Math.PI * (double)k /
                          (double)fft_FrameSize) + .5;
    g_Output_Accum[k] += (double)(2.0 * _window *
                        g_FFT_worksp[2 * k] / (fft_FrameSize2 * osamp));
}
```

```
            for (k = 0; k < stepSize; k++) g_Out_FIFO[k] = g_Output_Accum[k];
            for (k = 0; k < fft_FrameSize; k++)
            {
                g_Output_Accum[k] = g_Output_Accum[k + stepSize];
            }
            for (k = 0; k < inFifoLatency; k++)
                g_In_FIFO[k] = g_In_FIFO[k + stepSize];
        }
    }
    return outdata;
}
#endregion
#region Private Static Methods
public static void Short_Time_Fourier_Transform(double[] fft_Buffer, long
                    fft_FrameSize, long sign)
{
    double wr, wi, arg, temp;
    double tr, ti, ur, ui;
    long i, bitm, j, le, le2, k;

    for (i = 2; i < 2 * fft_FrameSize - 2; i += 2)
    {
        for (bitm = 2, j = 0; bitm < 2 * fft_FrameSize; bitm <<= 1)
        {
            if ((i & bitm) != 0) j++;
            j <<= 1;
        }
        if (i < j)
        {
```

```
        temp = fft_Buffer[i];
        fft_Buffer[i] = fft_Buffer[j];
        fft_Buffer[j] = temp;
        temp = fft_Buffer[i + 1];
        fft_Buffer[i + 1] = fft_Buffer[j + 1];
        fft_Buffer[j + 1] = temp;
    }
}
long max = (long)(Math.Log(fft_FrameSize) / Math.Log(2.0) + .5);
for (k = 0, le = 2; k < max; k++)
{
    le <<= 1;
    le2 = le >> 1;
    ur = 1.0F;
    ui = 0.0F;
    arg = (double)Math.PI / (le2 >> 1);
    wr = (double)Math.Cos(arg);
    wi = (double)(sign * Math.Sin(arg));
    for (j = 0; j < le2; j += 2)
    {
        for (i = j; i < 2 * fft_FrameSize; i += le)
        {
            tr = fft_Buffer[i + le2] * ur - fft_Buffer[i + le2 + 1] * ui;
                        ti = fft_Buffer[i + le2] * ui + fft_Buffer[i + le2 + 1] *
           ur;
            fft_Buffer[i + le2] = fft_Buffer[i] - tr;
            fft_Buffer[i + le2 + 1] = fft_Buffer[i + 1] - ti;
            fft_Buffer[i] += tr;
            fft_Buffer[i + 1] += ti;
            }
            tr = ur * wr - ui * wi;
            ui = ur * wi + ui * wr;
            ur = tr;
        }
    }
  }
  #endregion
}
```

## APPENDIX 4

```
public partial class Wav
  {
     public class Header
     {
        /*The header bytes of the wav file */
        /***RIFF******************/
        public byte[] chunkID;
        public uint fileSize;
        public byte[] riffType;
        /***fmt******************/
        public byte[] fmtID;
        public uint fmtSize;
        public ushort fmtCode;
        public ushort channels;
        public uint sampleRate;
        public uint fmtAvgBPS;
        public ushort fmtBlockAlign;
        public ushort bitDepth;
        /***data******************/
        public byte[] dataID;
        public uint dataSize;
        public int dataSampleCount;
        public uint wavMinLength;       //Length of audio in minutes
        public double wavSecLength;     //Length of audio in seconds
        public double wavInSec;
        public string fileName;
        public uint fileLength;
     }
```

```
public double[] mag;
public double[] selection;
public double[] data_double;
//public Complex[] df;

//The wav samples
private byte[] data;
public Wav()
{
   head = new Header();
   handle = new Handler();
}
public Wav(byte[] newData)
{
   head = new Header();
   head.chunkID = System.Text.Encoding.ASCII.GetBytes("RIFF");
   head.fileSize = 36 + (uint)newData.Length;
   head.riffType = System.Text.Encoding.ASCII.GetBytes("WAVE");
   head.fmtID = System.Text.Encoding.ASCII.GetBytes("fmt ");
   head.fmtSize = 16;//16 for PCM.
   head.fmtCode = 1; //PCM = 1
   head.channels = 1;// Mono = 1, Stereo = 2,
   head.sampleRate = 44100;// Common values are 44100 (CD), 48000
         (DAT). Sample Rate = Number of Samples per second, or Hertz. ;
   head.fmtAvgBPS = 88200;// ;//== SampleRate * NumChannels *
                  BitsPerSample/8 veya SampleRate * fmtBlockAlign
   head.fmtBlockAlign = 2;//== NumChannels * BitsPerSample / 8
   head.bitDepth = 16; // BitsPerSample    8 bits = 8, 16 bits = 16
```

```
    head.dataID = System.Text.Encoding.ASCII.GetBytes("data");
    head.dataSize = (uint)newData.Length; // == NumSamples * NumChannels
                                                   * BitsPerSample/8
    head.dataSampleCount = (int)head.dataSize / 2; //NumSamples =
                        NumBytes / (NumChannels * BitsPerSample / 8)
    head.wavMinLength = ((uint)head.dataSize / (uint)head.fmtAvgBPS) / 60;
    head.wavSecLength = ((double)head.dataSize / (double)head.fmtAvgBPS) -
                    (double)head.wavMinLength * 60;
    head.wavInSec = (double)head.dataSize / (double)head.fmtAvgBPS;
    data = newData;
    data_double = dataToDouble();
}
public double[] dataToDouble()
{
    handle = new Handler();
    double[] result = new double[data.Length / 2];
    for (int i = 0, pos = 0; pos < data.Length - 2; i++, pos++)
    {
        result[i] = handle.byteToDouble(data[pos], data[++pos]); //make method
                                                          static
    }
    return result;
}
public float[] dataToFloat()
{
    int input_Samples = data.Length / 2;
    float[] output = new float[input_Samples];
    int outputIndex = 0;
    for (int n = 0; n < input_Samples; n++)
    {
        short sample = BitConverter.ToInt16(data, n * 2);
        output[outputIndex++] = sample / 32768f;
    }
    return output;
}

//Returns the wav files samples in bytes.
public byte[] getData()
{
    return data;
}

public void setData (byte[] newData)
{
    data = newData;
    data_double = dataToDouble();
}
public double[] copy(int start, int end)
{
    double[] temp = new double[(end - start)];
    byte[] byteTemp = new byte[(2* end) - (2* start)];

    for (int i = 0; i < end - start; i++)
    {
        temp[i] = data_double[start + i];
    }
    for (int i = 0; i < (2 * end) - (2 * start); i++)
    {
        byteTemp[i] = data[(2 * start) + i];
    }
    Clipboard.SetAudio(byteTemp);
    return temp;
}
public void paste(int index)
{
    //If there is no data to paste
    if (Clipboard.GetAudioStream() == null)
        return;
    BinaryReader read = new BinaryReader(Clipboard.GetAudioStream());
    byte[] byteTemp =
                    read.ReadBytes((int)Clipboard.GetAudioStream().Length);
    List<byte> bTemp = data.ToList();
    bTemp.InsertRange(index * 2, byteTemp);
    updateData(bTemp.ToArray());
}
public double[] cut(int start, int end)
{
    double[] temp = copy(start, end);
    List<byte> lData = data.ToList();
    lData.RemoveRange(2 * start, (2 * end - 2 * start));
    updateData(lData.ToArray());
    //data_double = dataToDouble();
    return temp;
}
public void delete(int start, int end)
```

```
{
   double[] cut = new double[data_double.Length - (end - start)];
   List<byte> lData = data.ToList();
   try {
      lData.RemoveRange(2 * start, (2 * end - 2 * start));
      updateData(lData.ToArray());
   }
   catch { }
}

public void updateData(byte[] newData)
{
   head.fileSize = 36 + (uint)newData.Length;
   head.dataSize = (uint)newData.Length;
   data = newData;
   data_double=dataToDouble();
}

public byte[] toArray()
{
   List<byte> arr = new List<byte>();
   arr.AddRange(head.chunkID);
   arr.AddRange(BitConverter.GetBytes(head.fileSize));
   arr.AddRange(head.riffType);
   arr.AddRange(head.fmtID);
   arr.AddRange(BitConverter.GetBytes(head.fmtSize));
   arr.AddRange(BitConverter.GetBytes(head.fmtCode));
   arr.AddRange(BitConverter.GetBytes(head.channels));
   arr.AddRange(BitConverter.GetBytes(head.sampleRate));
```

```
    arr.AddRange(BitConverter.GetBytes(head.fmtAvgBPS));
    arr.AddRange(BitConverter.GetBytes(head.fmtBlockAlign));
    arr.AddRange(BitConverter.GetBytes(head.bitDepth));
    arr.AddRange(head.dataID);
    arr.AddRange(BitConverter.GetBytes(head.dataSize));
    arr.AddRange(data);
    return arr.ToArray();
}

public byte[] FloatArrayToByteArray(float[] floatArray1)
{
    int volume = 1;
    short[] destBuffer = new short[floatArray1.Count()];
    int destOffset = 0;
    for (int sample = 0; sample < floatArray1.Count(); sample++)
    {
        // adjust volume
        float sample32 = floatArray1[sample] * volume;
        // clip
        if (sample32 > 1.0f)
            sample32 = 1.0f;
        if (sample32 < -1.0f)
            sample32 = -1.0f;
        destBuffer[destOffset++] = (short)(sample32 * 32767);
    }
    return destBuffer.Select(x => Convert.ToInt16(x))
                  .SelectMany(x => BitConverter.GetBytes(x))
                  .ToArray();
}
```

```
public byte[] DoubleArrayToByteArray(double[] doubleArray)
{
   int nSamples = doubleArray.Length;
   byte[] dataB = new byte[nSamples * 2];
   int s = 0;
   for (int j = 0; j < dataB.Length; j = j + 2)
   {
      handle.doubleToBytes(doubleArray[s++], out dataB[j], out dataB[j + 1]);
   }
   return dataB;
}

public double[] ByteArrayToDoubleArray(byte[] inputArray)
{
   Handler handle = new Handler();
   double[] result = new double[inputArray.Length / 2];
   for (int i = 0, pos = 0; pos < inputArray.Length - 2; i++, pos++)
   {
      result[i] = handle.byteToDouble(inputArray[pos], inputArray[++pos]);
                                           //make method static
   }

   return result;
}
public float[] ByteArrayToFloat(byte[] input_Array)
{
   int input_Samples = input_Array.Length / 2; // 16 bit input, so 2 bytes per
                                                               sample
```

```
        float[] output = new float[input_Samples];
        int outputIndex = 0;
        for (int i=0, n = 0; n < input_Samples;i++, n++)
        {
            short sample = BitConverter.ToInt16(inputArray, n * 2);
            output[outputIndex++] = sample / 32768f;
        }
        return output;
    }

}
```

**APPENDIX 5**

```
public static class PitchDetection2
  {
     public enum Pitch_Detect_Algorithm
     {
        Autocorrelation, Amdf
     }

     public static double DetectPitch(Wav w)
     {
        return detect_Pitch_Calculation(w, 50.0, 500.0, 1, 1,
               Pitch_Detect_Algorithm.Autocorrelation)[0];
     }
     private static double[] detect_Pitch_Calculation(Wav w, double min_Hz,
                    double max_Hz, int n_Candidates, int n_Resolution,
                    Pitch_Detect_Algorithm algorithm)
     {
        // note that higher frequency means lower period
        int n_Low_Period_In_Samples = hz_To_Period_In_Samples(max_Hz,(int)
                w.head.sampleRate);
        int n_Hi_Period_In_Samples = hz_To_Period_In_Samples(min_Hz,
                    (int)w.head.sampleRate);
        if (n_Hi_Period_In_Samples <= n_Low_Period_In_Samples) throw new
                        Exception("Bad range for pitch detection.");
        if (w.head.channels != 1) throw new Exception("Only mono supported.");
        double[] _samples = w.data_double;
        if (_samples.Length < n_Hi_Period_In_Samples) throw new
                    Exception("Not enough _samples.");
```

```
double[] results = new double[n_Hi_Period_In_Samples -
 n_Low_Period_In_Samples];

if (algorithm == Pitch_Detect_Algorithm.Amdf)
{
   for (int period = n_Low_Period_In_Samples; period <
       n_Hi_Period_In_Samples; period += n_Resolution)
   {
      double sum = 0;
      for (int i = 0; i < _samples.Length - period; i++)
         sum += Math.Abs(_samples[i] - _samples[i + period]);
      double mean = sum / (double)_samples.Length;
      mean *= -1;
      results[period - n_Low_Period_In_Samples] = mean;
   }
}
else if (algorithm == Pitch_Detect_Algorithm.Autocorrelation)
{
   for (int period = n_Low_Period_In_Samples; period <
       n_Hi_Period_In_Samples; period += n_Resolution)
   {
      double sum = 0;
      for (int i = 0; i < _samples.Length - period; i++)
         sum += _samples[i] * _samples[i + period];
      double mean = sum / (double)_samples.Length;
      results[period - n_Low_Period_In_Samples] = mean;
   }
}
```

```
      int[] best_Indices = find_Best_Candidates(n_Candidates, ref results);
     // convert back to Hz
     double[] res = new double[n_Candidates];
     for (int i = 0; i < n_Candidates; i++)
        res[i] = periodInSamplesToHz(best_Indices[i] +
               n_Low_Period_In_Samples,(int) w.head.sampleRate);
     return res;
  }

  private static int[] find_Best_Candidates(int n, ref double[] inputs)
  {
     if (inputs.Length < n) throw new Exception("Length of inputs is not long
enough.");
     int[] res = new int[n]; // will hold indices with the highest amounts.
     for (int c = 0; c < n; c++)
     {
        // find the highest.
        double f_Best_Value = double.MinValue;
        int n_Best_Index = -1;
        for (int i = 0; i < inputs.Length; i++)
           if (inputs[i] > f_Best_Value) {
              n_Best_Index = i;
              f_Best_Value = inputs[i];
           }
        // record this highest value
        res[c] = n_Best_Index;
        // now blank out that index.
        inputs[n_Best_Index] = double.MinValue;
```

```
          }
          return res;
        }

        private static int hz_To_Period_In_Samples(double hz, int sample_Rate)
        {
          return (int)(1 / (hz / (double)sample_Rate));
        }
        private static double periodInSamplesToHz(int period, int sample_Rate)
        {
          return 1 / (period / (double)sample_Rate);
        }
    }
}
```