Speaker Recognition in a handheld computer

CARLOS DOMÍNGUEZ SÁNCHEZ

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:285

Speaker Recognition in a handheld computer

Carlos Domínguez Sánchez

November 15, 2010

Supervisor and examiner: Gerald Q. Maguire Jr.

School of Information and Communication Technology (ICT)

Royal Institute of Technology (KTH)

Stockholm, Sweden

Abstract

Handheld computers are widely used, be it a mobile phone, personal digital assistant (PDA), or a media player. Although these devices are personal, often a small set of persons can use a given device, for example a group of friends or a family.

The most natural way to communicate for most humans is through speech. Therefore a natural way for these devices to know who is using them is for the device to listen to the user's speech, i.e., to recognize the speaker based upon their speech.

This project exploits the microphone built into most of these devices and asks whether it is possible to develop an effective speaker recognition system which can operate within the limited resources of these devices (as compared to a desktop PC). The goal of this speaker recognition is to distinguish between the small set of people that could share a handheld device and those outside of this small set. Therefore the criteria is that the device should work for any of the members of this small set and not work for anyone outside of this small set. Furthermore, within this small set the device should recognize which specific person within this small group is using it.

An application for a *Windows Mobile* PDA has been developed using C++. This application and its underlying theoretical concepts, as well as parts of the code and the results obtained (in terms of accuracy rate and performance) are presented in this thesis. The experiments conducted within this research indicate that it is feasible to recognize the user based upon their speech is within a small group and further more to identify which member of the group is the user. This has great potential for automatically configuring devices within a home or office environment for the specific user. Potentially all a user needs to do is speak within hearing range of the device to identify themselves to the device. The device in turn can configure itself for this user.

Sammanfattning

Handdatorer används mycket, det kan vara en mobiltelefon, handdator (PDA) eller en media spelare. Även om dessa enheter är personliga, kan en liten uppsättning med personer ofta använda en viss enhet, t.ex. en grupp av vänner eller en familj.

Det mest naturliga sättet att kommunicera för de flesta människor är att tala. Därför ett naturligt sätt för dessa enheten att veta vem som använder dem är för enheten att lyssna på användarens röst, till exempel att erkänna talaren baserat på deras röst.

Detta projekt utnyttjar mikrofonen inbyggd i de flesta av dessa enheter och frågar om det är möjligt att utveckla ett effektivt system högtalare erkännande som kan verka inom de begränsade resurserna av dessa enheter (jämfört med en stationär dator). Målet med denna högtalare erkännande är att skilja mellan den lilla set av människor som skulle kunna dela en handdator och de utanför detta lilla set. Därför kriterierna är att enheten bör arbeta för någon av medlemmarna i detta lilla set och inte fungerar för någon utanför detta lilla set. Övrigt inom denna lilla set, bör enheten erkänna som specifik person inom denna lilla grupp.

En ansökan om emph Windows Mobile PDA har utvecklats med C++. Denna ansökan och det underliggande teoretiska begreppet, liksom delar av koden och uppnådda resultat (i form av noggrannhet hastighet och prestanda) presenteras i denna avhandling. Experimenten som utförs inom denna forskning visar att det är möjligt att känna användaren baserat på deras röst inom en liten grupp och ytterligare mer att identifiera vilken medlem i gruppen är användaren. Detta har stor potential för att automatiskt konfigurera enheter inom en hemifrån eller från kontoret till den specifika användaren. Potentiellt behöver en användare tala inom hörhåll för att identifiera sig till enheten. Enheten kan konfigurera själv för denna användare.

Acknowledgements

First of all I would like to sincerely thank to my advisor, Professor Gerald Q. Maguire Jr., for overcoming all my doubts, sharing his huge knowledge, answering my questions as quickly as possible, and being always willing to help me.

I would like also to thank all my colleagues in the department, Sergio, Luis, Joaquin, Victor, David,... They have helped to create a really good atmosphere that has made my work easier and more fun.

My family must be mention here, especially my parents and my brother, because they have been encouraged me every day, even when they have not had their best moments.

There is another “family” in Stockholm: Gema, Victor, and Sandra. They have been happy when I was happy and they have been worried when I was worried.

And last but not least, my friends: especially those of you who are here (Victor, Patrica, Sergio, Manu, Mario, Álvaro, Fer, ...) the ones that are located in the rest of the world (Raquel, Hector, Alberto, Jaime, Jesús, Santos, Elena, Pedro, Cristina, ...) all of whom have encouraged me, making my life easier and making me laugh. Thank you to all!

Contents

List of Figures

[1]Details about Creative Commons Attribution License BY 2.0 can be found at: `http://creativecommons.org/licenses/by/2.0/` and Open Educational Resources can be found at `http://www.oercommons.org/`.

List of Tables

List of abbreviations

ADC	Analog Digital Converter
AT&T	American Telephone and Telegraph
BBN	Bolt, Beranek, and Newman
CCAL	Creative Commons Attribution License
CMS	Cepstral Mean Subtraction
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DTW	Dynamic Time Warping
FFT	Fast Fourier Transform
GMM	Gaussian Mixture Model
HMM	Hidden Markov Model
IDE	Integrated Development Environment
LAR	Log-Area Ratios
LBG	Linde, Buzo, and Grey
LP	Linear Prediction
MFC	Microsoft Foundation Class
MFCC	Mel Frequency Cepstrum Coefficients
MIT-LL	Massachusetts Institute of Technology Lincoln Laboratory
ORE	Open Educational Resources
PDA	Personal Digital Assistant
PIN	Personal Identification Number
RAM	Random Access Memory
RASTA	Representation Relative Spectra
ROM	Read Only Memory
SDK	Software Development Kit
WLAN	Wireless Local Area Network

Chapter 1

Introduction

1.1 Overview

Almost everyone owns a mobile device, be it a mobile phone, a personal digital assistant (PDA), or a media player. The main features of these devices is that they are small and you can use them wherever you want (With of course some limitations on their use in aircraft, hospitals, etc.). These devices are continually improving. These devices also have many components, including microphone, speakers, and a relatively powerful processor. Some of these device also offer internet connectivity (via a wireless interface), Bluetooth connectivity, integrated camera, accelerometer(s), fingerprint reader, etc.

With these devices we can measure a lot of information about the user's environment (i.e., the user's context) and we can use some of this to decide how to configure features of the device. For example, if we detect there is an available wireless local area network (WLAN), then we can attempt to connect to this WLAN, to have broadband wireless internet access. Similarly, we can look for other devices using a Bluetooth interface, for example to utilize a Bluetooth headset. Additionally, we can recognize if the device is lying face up or face down using the accelerometer(s) or we can authenticate a person based upon his or her his fingerprint or using a camera.

Because many of these devices are equipped with a built-in microphone we can exploit algorithms that can extract features of human speech in order to determine who is speaking (i.e., speaker recognition), what is being said (i.e., speech recognition), what is the emotional state of the speaker, etc. Because the microphone is connected to an analog to digital converter, we can sample the voice and perform a digital signal processing. One of the types of processing that we can do is to extract features, then compare these features with previously recorded and processed signals in order to recognize words, syllables, languages, or even speakers. Depending on the task, we need to extract different features from the speech and we have to apply different algorithms to the signal to accomplish the desired task.

In this thesis we study the feasibility of applying some of these algorithms on a mobile device, specifically on a PDA [1], in order to be able to recognize *who is speaking* to the device or who is speaking near the device.

1.2 Problem Statement

The goal of a speaker recognition algorithm is “to decide which voice model from a known set of voice models best characterizes a speaker” [23]. Speaker recognition systems can be used in two different ways: verification or identification. In the first a person who is claiming an identity speaks, then the system decides if he/she is an imposter or not. In identification systems we have a set of users and the system decides who is the most likely for a given speech utterance. There are two types of errors, first an impostor could be *incorrectly* accepted as a true claimant, and a true claimant could be *falsely* rejected as an impostor [23].

In this thesis project we have built an identification system that will execute on the user’s PDA, because this platform can and will be used by the user for for a number of applications. Some example applications that can be enabled by speaker a recognition are described in the next paragraphs.

Suppose you own a media player, with a speech recognition system, you could control it using your voice [29], but if you want to loan it to your son, he could not use it, because the speech recognition system was trained for your voice. Additionally your son probably wants to use a different playlist than you, hence not only does your son want to use the media player but he wants it to act as *his* media player. When using a speaker recognition system, we could use the first word to distinguish which of a small set of users is speaking, and then we can initialize the appropriate speech recognition system that has been trained with this specific speaker’s voice. Additionally, we can configure the media player with his or her playlist or favorite radio station. This speaker recognition is possible by training the PDA for only a few minutes per user.

Another possible application is to detect who is near the PDA, suppose you want to meet with a colleague (named Hans) when your device detects that Hans is speaking, it can advice you that Hans is nearby and remind you that you wanted to meet with him [24].

In addition to learning who is using the PDA or who is near it, we can detect *what* is near the device. For example, we can measure characteristics of the audio

[1]The PDA used for this project is the HP iPAQ 5500.

in order to know if I am alone in which case the device could use its speaker output rather than a headset.

Another goal of this thesis is to measure the limits of such an application. For example, we want to understand how many users the application can distinguish between, what is the accuracy of the recognition, and how does this accuracy depends on the environment. While we might not distinguish a specific individual from 1000 potential speakers, it is perhaps both interesting and sufficient to recognize members of your family, your friends, or colleagues/classmates/... in your department.

Chapter 2

Background

In this chapter we present how typical speaker recognition systems have been developed. Furthermore we present the parts that the system used to recognize who is speaking and summarize the results of some previous systems.

2.1 How does human speech work?

While speaking is a common way to communicate, it is a complex process and many parts or the body are involved as shown in Figure 2.1.

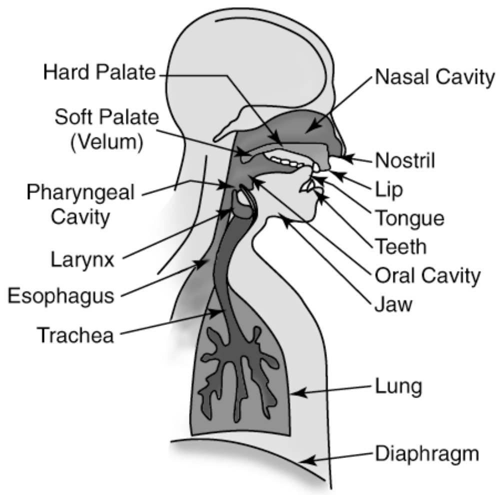

Figure 2.1. Speech production. Figure from Charles A. Bourman [4], appears here under the Creative Commons Attribution License BY 2.0 and it is an is an Open Educational Resource [a]

[a]Details about Creative Commons Attribution License BY 2.0 can be found at: `http://creativecommons.org/licenses/by/2.0/` and Open Educational Resources can be found at `http://www.oercommons.org/`.

To produce a vowel air comes from the lungs and flows along the vocal cords causing them to vibrate. The frequency of this vibration is called *the pitch of the sound*. This vibration propagates though the oral and nasal cavity, which acts as a filter. The output is a pressure wave through the lips that can be recorded by a microphone, and later analysed to extract features, such as the *pitch*. The features of this pressure wave vary between speakers because the vocal cords, oral cavity, and nasal cavity are different from person to person. Furthermore, the wave varies even if a speaker repeats the same utterance because it depends on their mood, changes in the voice, ..., thus it is impossible for a human being to reproduce exactly the same speech twice[1], this complicates the speaker recognition task.

2.2 Phases in a speaker recognition system

Every speaker recognition system has two phases: a training and a test phase. It is mandatory to perform the phases in this order, thus first we perform the training phase and only after finishing this phase can we perform the test phase. It is this final phase that will be used to recognize the speaker at some later point in time. We can further describe these two phases as:

Training phase In the training phase we teach the system to recognize a speaker. The longer this phase lasts, the more information we have about the speaker and higher the accuracy. The training set could be a recording from just one word or several minutes (or even hours) of speech. This phase must be completed before using the system and we have to record every user's voice (among those who will use the system). The result of this phase is the *Model* that we can see in Figure 2.2. There are as many *Models* as users in the system, hence every user must complete the training phase.

Test phase Once the system has created a model for the voices of the set of speakers, then we can start to use the system. To do this we record speech from an initially unknown user, then compare it with all the speakers who were enrolled during the training phase. The closest match is chosen as the output of the test phase.

We assume that there is e a finite number, N, of users in the system. However the set of users could be a *closed set* or an *open set*. If we have a *closed set* we have N users and anyone who is in this set can use the system. Otherwise, we have an *open set* where anyone can use the system, but only N users can be successfully

[1]There are persons who can imitate other speakers voices to some degree, see for example [6]

recognized. In order to work with an *open set* we need to include a threshold in the greatest likelihood decision: if the greatest likelihood is larger than the threshold, then the user is recognized as a member of the *open set*, but if the greatest likelihood is smaller than the threshold the user is unknown to the system[10].

2.3 Kinds of speaker recognition systems

There are some different kinds of speaker recognition system depending on the utterances that the users should speak to the system

Text-dependent In a text-dependent approach we have to say exactly the same thing during both the training phase and the test phase. This utterance could be a word, a few numbers, your name, or whatever you want; but must be the same in both phases. It is important to note that all speakers must say the same thing.

Text Independent A text independent approach is the opposite of the text-dependent approach. Here during the training phase and during the test phase the speaker can say anything. Typically in this kind of system the training phase is longer and the test phase may require more speech (i.e., a longer utterance) in order to recognize the speaker.

2.4 Recognition and Verification

There are two main goals of speaker recognition technologies: *speaker recognition* and *speaker verification.*

The goal of speaker recognition is to distinguish between speakers, based upon the data collected during the training phase we will identify one of these speakers as a specific speaker during the test phase. Typically such systems require a long training phase. The training and test phases can be text-dependent or text-independent.

As we can see in Figure 2.2 the process starts when an unknown user speaks to the system, then some features are extracted and later these features are compared with the models of the speakers that previously had been calculated during the training phase. Finally the speaker whose model has the greatest likelihood compared with the extracted features is recognized as the user who is speaking to the system.

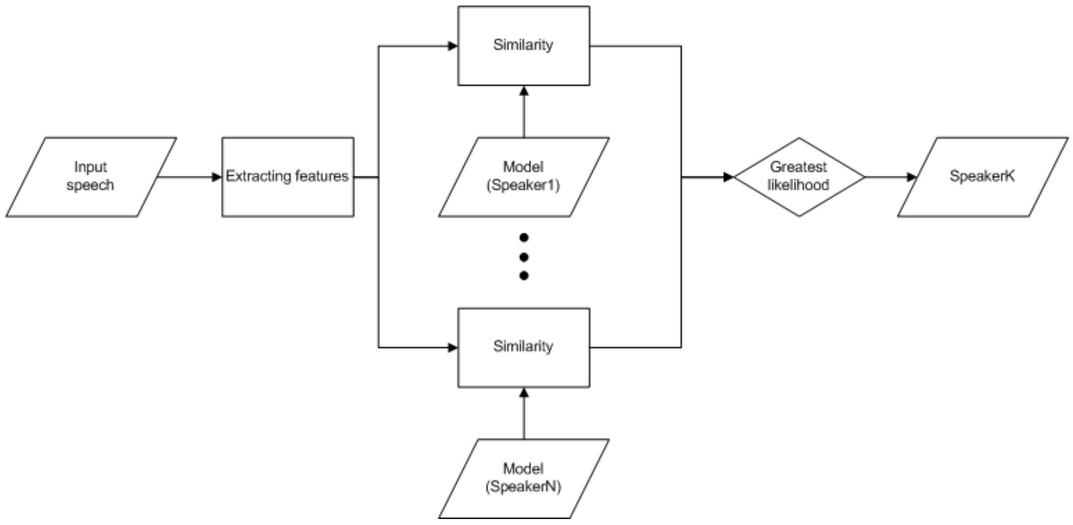

Figure 2.2. Flowchart of a speaker recognition system.

Speaker recognition is widely used to provide security as part of user verification. The claimant tells the system who he or she (i.e., the speaker indicates who they claim to be), for example by typing a user name, and then the user starts speaking. The system checks if this speaker matches with the supposed user. If they do not match the user is marked as an impostor. Typically security systems are text-dependent in order to reduce the false positive rate (i.e., to minimize the probability of rejecting valid users). Note that in practice the specific text that the user must say may be indicated by the system, for example the user is prompted to speak a generated sequence of words - this can be done to reduce the possibility of an attack using a recording of the user's voice.

An example of a verification application could be a user saying a PIN code or credit card number. This approach could be used to increase the security of an on-line purchase using the user's personal mobile device.

As we can see in Figure 2.3 the process is similar, the main difference is the *threshold.* In order to ensure that a claimant is who they claim to be, the likelihood must be greater than this threshold.

A detailed explanation about *extracting features* and the way to calculate the *similarity* is given in Chapter 3.

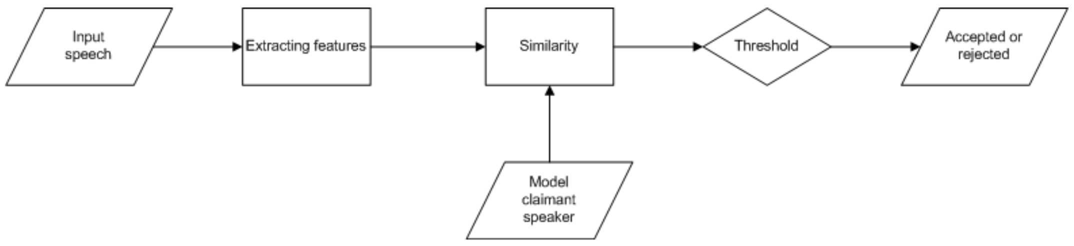

Figure 2.3. Flowchart of a speaker verification system.

2.5 Previous work

In the 1960s Bell Labs started studying the feasibility of developing an automatic speaker recognition system. Over many years, several text dependent dependent and independent systems were developed using different ways to extract the *features*, different ways to match the *models*, and varying the length of the *training* and *test* phases[7]. Table 2.1 lists some of these systems in chronological order.

Table 2.1. Advancenment in speaker recognition [2].

Organization	Features	Method	Input	Text	Population	Error
AT&T	Cep.	Pattern Match	Lab	D	10	2%@0.5s
STI	LP	L. T. Statistic	Labs	I	17	2%@39s
BBN	LAR	Nonpar.pdf	Phone	I	21	2%@2.5s
ITT	LP Cep.	DTW	Lab	I	11	21%@3s
MIT-LL	Mel-Cep.	HMM	Office	D	138	0.8%@10s

The *organization* column show the company who developed the system. The *features* and *Method* columns show the methods used to extract the features and to match the patterns. *Input* indicates the quality of the speech: Lab indicating laboratory recordings, Phone indicating telephone recordings, and Office indicating recordings in a typical office environment. The column labelled *text* indicates if the system is text dependent (abbreviated “D”) or independent (abbreviated “I”). The *population* column indicates the number of users in the system. The final column *error* shows the percentage of incorrectly recognized users together with an indication of the length of the training phase.Today several high level features are being studied, specifically idiolect, phone usage, pronunciation,...Some researches are using a combination of audio and visual features, i.e. studying the lip movement of the speaker as they speak.

Chapter 3

Steps performed in a speaker recognition system

This chapter presents the theoretical concepts used in our speaker recognition system. In the following sections and subsections the algorithms used are presented in chronological order of use, first the algorithms needed for the training phase and last the algorithms used in the test phase. Figure 3.1 show the flow chart of the training phase.

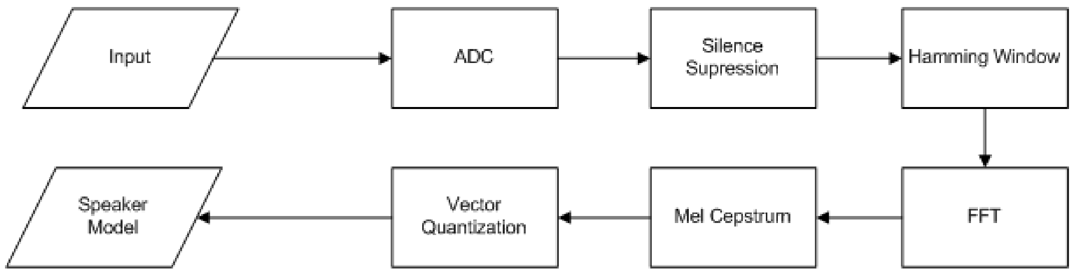

Figure 3.1. Flow chart of the training phase.

3.1 Extracting features

This section explains in detail the box *extracting features* shown in Figures 2.2 and 2.3. The first step is to record the speech and sample and quantize it. Once we have a digital signal, we can extract a lot of attributes from it, such as “clarity”, “roughness”, “magnitude”, “animation”, “pitch intonation”, “articulation rate”, and “dialect”. Some of these attributes are difficult to extract and some have little significance for speaker recognition. Others attributes are more measurable and depend on the nature of voice, thus they do not change very much over time. The later types of attributes are called low-level attributes. These are useful for speaker identification, hence they are more interesting for us. Three of these attributes are spectrum, pitch[1], and glottal flow[2].

3.1.1 Sampling the speech

The first step in every speaker recognition system is to extract a digital signal containing the information contained in the pressure wave that each person produces when speaking. A microphone is used to convert this pressure wave into an analog signal, and then this analog signal is converted into a digital signal using an analog to digital converter.

The analog signal resulting from human speech contains most of its energy between 4 Hz and 4 KHz, so we can use a low pass filter with a cut off frequency of 4 KHz in order to reduce aliasing [3] when digitizing the signal. In our test device (*HP iPAQ 5500*) this filter is include in its *Asahi Kasei AK4535* 16 bit stereo CODEC and its cut off frequency is set to $0.454 * fs$ [11]. With this filter we can sample the analog signal at 8 KHz (or a higher sampling rate). This filter bandwidth and sampling rate can be increased to obtain more information about the speech signal. In all cases, the sampling rate should be at least twice the highest frequency that we expect in our signal and in this case this is the highest frequency that the low pass filter allows. The Nyquist-Shannon sampling theorem [31] says that we can reproduce an analog signal if we sample at a rate that is at least twice the highest frequency contained in the signal.

Let $x(t)$ be our low pass filtered analog signal, we can obtain the digital signal by sampling:

$$x[n] = \sum_{m} x(t).\delta(t - m.T_s) \tag{3.1}$$

were T_s is the sampling period, the sampling rate f_s is $\frac{1}{T_s}$. We make f_s samples per second. The common rates for sampling voice are: 8 KHz, 12 KHz, or 22 KHz.

Once we have the samples we need to quantize them, this to assign a digital number to represent each sample. The more bits you use to quantify this number, the more precisely you quantize your signal. It is typical to use 8 or 16 bits to quantize speech samples. We use 16 bits because an audio device have a 15 or 16 bit analog to digital converter (often as part of an audio CODEC chip) for audio.

[1]The pitch is the main frequency of the sound, in this case the signal.

[2]The glottal flow can be studied for speaker and dialect identification as we can see here [32].

[3]Aliasing is an effect that produces distortion in a signal when it is sampled , as higher frequency signals will appear as lower frequency aliases[3].

This both gives us a lot of resolution and such values are easy to represent in most programming languages, for example with a `short int` in C.

The upper curve in Figure 3.2 shows the analog signal, while the lower curve shows the signal sampled and quantized as 16 bit number. This stream of 16 bit numbers is the digital signal that we will process.

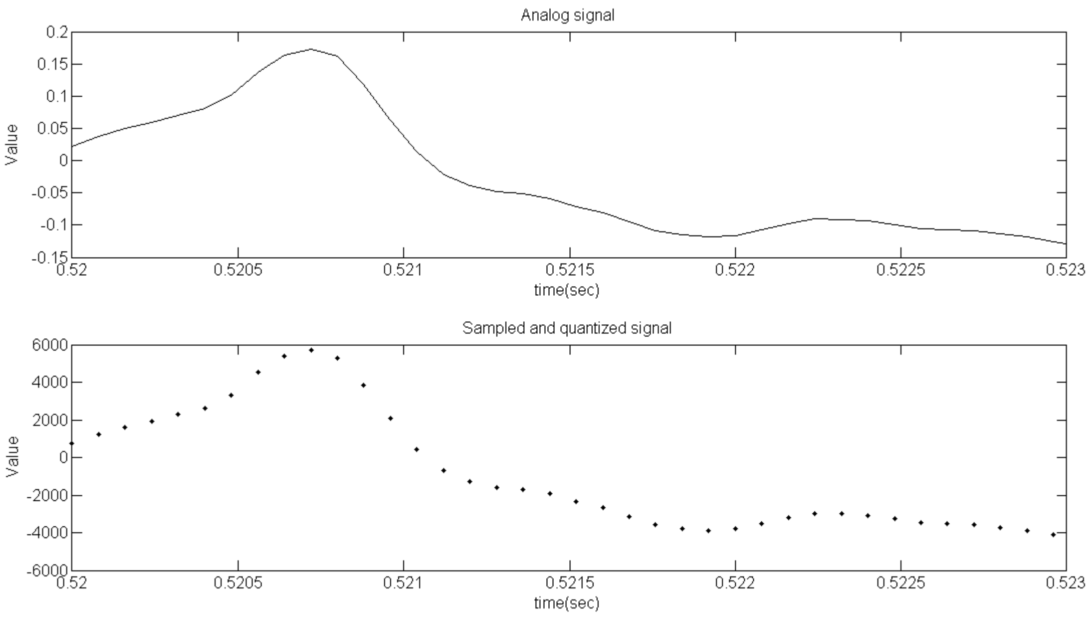

Figure 3.2. Analog speech signal vs. Sampled and Quantized speech signal.

3.1.2 Silence suppression

When processing speech we generally divide the stream of samples into frames, where each frame is a group of N samples. The first frame contains the first N samples. Thus a single frame can contain samples from M until $M + N$. Therefore we divided the signal such that there are overlapping $N - M$ samples [5]. This overlap enables us to process the frames independently.

It is possible to measure the energy in each frame by applying formula 3.2, where N is the number of samples per frame. A typical value of N is 256.

$$E_{frame} = \sum_{N} x[n]^2 \tag{3.2}$$

When the stream of energies is analysed, if there are a number of consecutive frames during which energy is larger than a specified threshold, then the start of an utterance has been found. Conversely, if there are a number of consecutive frames in which the energy is lower than the same, then it is not necessary to compute more frames because the utterance has finished. This the process is repeated with the following frames in order to find new utterances.

An example of such a silence suppression algorithm is shown in Figure 3.3. This signal was recorded with the PDA, so the first peak does not correspond to the real signal, it corresponds to the first readings of the buffer, before any samples have been collected. So the first time we access the buffer we have to skip the first several frames, and then start applying the silence suppression algorithm. The number of consecutive frames needed to find the end and the start of an utterance was set to 5 and the threshold is set to 130. These specific values were found experimentally, but 5 frames at an 8 KHz sampling rate corresponds to 160 milliseconds[4] , while a threshold of 130 was found to be effective in detecting silence.

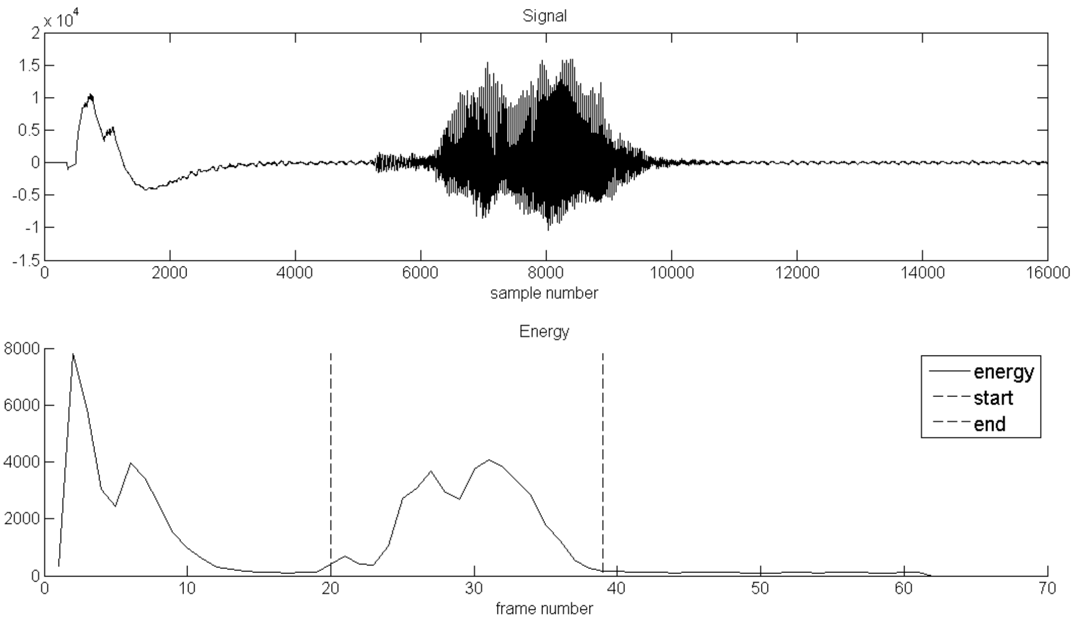

Figure 3.3. Execution of the silence suppresion algorithm.

In order to establish the threshold in practice, it is possible to sample during a couple of seconds of silence and then set the threshold to the average energy per

[4]As the the average phoneme duration is 100 milliseconds, and the phonemes must have energy we can cover detect them[1].

frame. Hence, the value of the threshold depends on the current audio environment.

3.1.3 Hamming Windowing

Every frame must be window in order to avoid discontinuities at the beginning and at the end of each frame. The most widely used window in signal processing is the *Hamming* window, shown in Figure 3.4 and described by Equation 3.3.

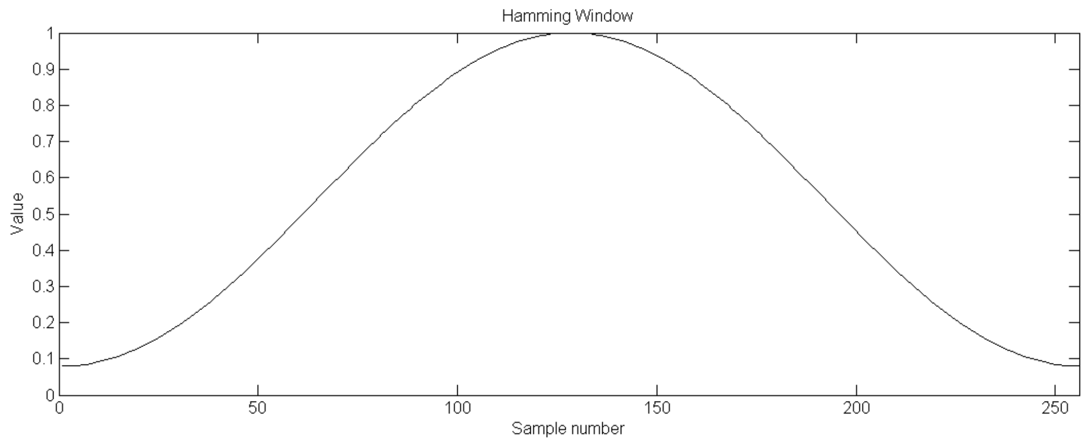

Figure 3.4. Hamming Window.

$$h(n) = 0.54 - 0.46.cos(\frac{2.pi.n}{N}) \tag{3.3}$$

For more information about windowing functions and how they improve the FFT see [13].

3.1.4 Fourier Transform

At this point in the process we have a number of meaningful frames (i.e., we have suppressed the silence). One of the most useful ways to study these frames is to compute their spectrum, i.e., a spectral measurement. We can easily do this by computing a Short Time Fourier Transform. As each frame contains $N = 256$ samples, and the sampling rate used was 8 or 12 KHz, then each frames contains between 20 and 30 milliseconds of speech.

In order to calculate the spectral measurements quickly a Fast Fourier Transform (FFT) is computed. With this algorithm it is feasible to get the same result as with a Discrete Fourier Transform (DFT) in a faster way. Formula 3.4 presents how to calculate the DFT.

$$X[k] = \sum_{n=0}^{N-1} x(n).e^{-i.2.\pi.k.\frac{n}{N}} \qquad k = 0, ..., N-1 \tag{3.4}$$

As the digital signal is real [5], the whole DFT is not needed. When $x[n]$ is real, then $X[k] = X^*[-k]$. Hence, half of the coefficients contain no information and are not required in the rest of the process. Actually, just the module of $X[k]$ is required with $k = 0...\frac{256}{2} + 1$.

After transforming each frame into the frequency domain we have a vector of values as encoded as a color (in this figure a shade of gray) in each of the columns in Figure 3.5. In this figure we transform a speech signal [6] with $N = 256$ and $M = 100$ and then take the logarithm of the resulting values before mapping them to a color (see the scale to the right of the figure) in order to highlight the differences between high and low values. Most of the energy is under 4 KHz, so a sampling rate of 8 - 12.5 KHz is quite suitable.

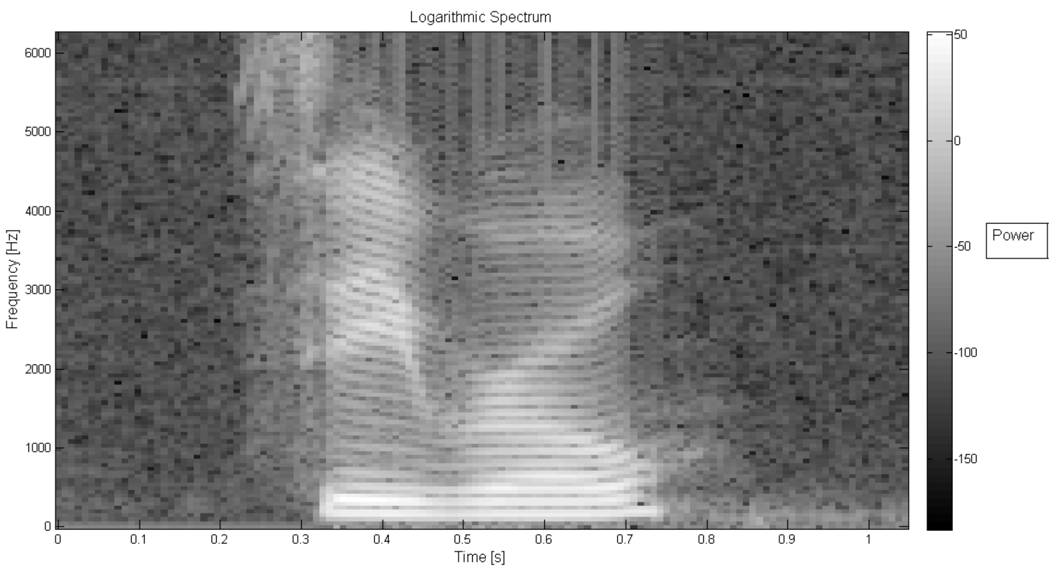

Figure 3.5. Logarithmic spectrum of a speech signal as a function of time.

[5] The signal is real because the imaginary part is 0.

[6] The signal is an English speaking man saying ”zero”, sampled at 12.5 KHz with 16 bits per sample.

3.1.5 Mel Frequency Cepstrum Coefficients

The most successful feature used for performing speaker recognition over the years is called Mel Frequency Cepstrum Coefficients. This algorithm consists of applying a set of filters to the spectrum of our signal in order to measure how much energy is in each frequency band (channel). The result is a parametric representation[15] of speech, while reducing the amount of information that needs to be compared between samples of speech from a given speaker and previously recorded speakers.

Mel-Cepstrum is based on how the human ear works, as it exploits the variation in amplitude sensitivity in different bands by applying different types of filters over different ranges of frequencies. More information about Mel-Cepstrum can be found in [15] and [23].

To calculate the Mel Frequency Cepstrum Coefficients (MFCC) we need to perform two steps, first we translate the frequencies into the mel-frequency scale, this is a logarithmic scale. We can apply this transformation with equation 3.5 .

$$mel(f) = 2595. \log_{10}(1 + \frac{f}{700}) \tag{3.5}$$

The second step returns us to the time domain. For this we have to choose the number of coefficients that we wish, a typical value is K=16 or 32. Then we will apply these K filters, spaced over the mel scale as shown in Figure 3.6

Both, the center frequency and the bandwidth of the filter vary in the mel-frequency scale. Due to the mel-frequency scale the first filters have a small bandwidth compared to the last filters and there are more filters in the low frequency part of the spectrum than in the high frequencies.

In Figure 3.6, the filters are shown in the frequency domain. Hence to filter the signal we multiply the signal in the frequency domain by the coefficients of the filter as shown in Formula 3.6, where $X[k]$ is the DFT of a frame, $Y_p[k]$ is the filter number p, K is the total number of filters, and N is the order of the DFT.

$$MFC_p = \sum_{k=0}^{\frac{N}{2}+1} X[k].Y_p[k] \qquad\qquad p = 1..K \tag{3.6}$$

Finally we need to apply a Discrete Cosine Transom (DCT) to the logarithm of the output of the filterbank which transforms the results to the *cepstrum* domain, thus we de-correlate the feature components[26]. Equation 3.7 shows the final result.

$$MFCC_p = DCT(MFC_p) = \sum_{k=1}^{K} \log(MFC_k). \cos(n.(k - \frac{1}{2}).\frac{\pi}{K}) \tag{3.7}$$

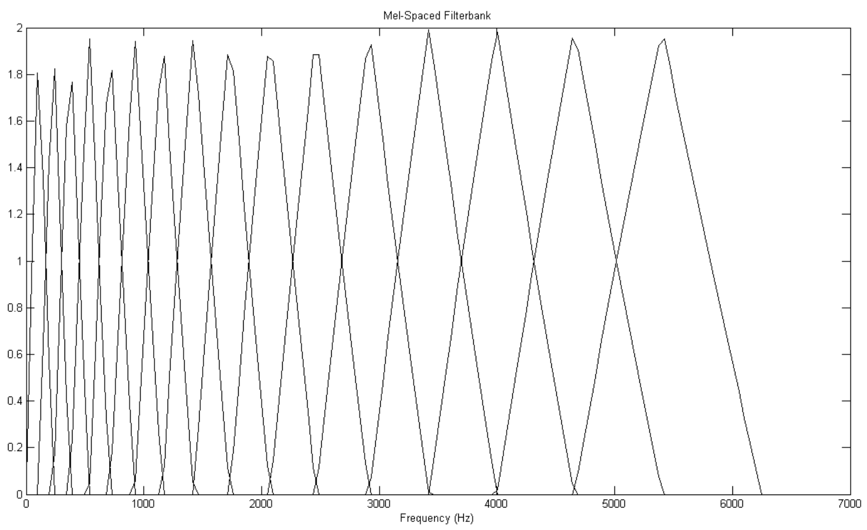

Figure 3.6. Mel-spaced Filterbank.

After this process, a vector with p components (the number of filters in the filterbank) is obtained per frame, this vector is called *features vector* or *cepstral vector*. The amount of information has been reduced, from $N = 256$ samples to $p = 16$ or 32 components.Consider an utterance of 2 second duration, roughly 100 frames, so 3200 components would need to be stored as the *speaker model*. To further reduce the amount of data we can utilize algorithms that can reduce the number of vectors, while maintaining most of the information. These algorithms are presented in the following section.

3.2 Recognition algorithm

After extracting the features (in our case the MFCC) we need to compare them in order to estimate which of the speakers is the most likely speaker. In this section the box *greatest likehood* from Figure 2.2 on page 8 is explained in detail.

A number of approaches can be used to measure the distance between the measured features and the features captured during the training phase, such as Gussian Mixture Model (GMM) and Hidden Markov Model (HMM). In this project I have chosen to use a Vector Quantization approach because it is widely used in *text dependent* speaker recognition systems.

As stated in Section 3.1, the amount of data resulting from several seconds of samples is quite a large number of bits, even after it has been reduced. Comparing 3200 components per speaker would take a long time. In order to reduce this information a compression algorithm can be used. As will be described below vector quantization provides both this compress and prepares them for computing the distance between the current speaker and the speaker models from the training phase.

3.2.1 Vector quantization

There are many algorithms that try to compress information by calculating centroids. This is: vectors in a vector space are clustered into centroids and the label of the nearest centroid can be used to represent the vector. Ideally there are few but well separated centroids (the more centroids we have, the lower the distortion). A *model* is the set of centroids $\{c_1, c_2, ..., c_L\}$. An entry in this codebook is the *model* of each speaker, see Section 2.4

Let assume that there are M vectors, $\{x_1, x_2, ..., x_M\}$ (in a speaker recognition system there are as many vector as frames in the utterance) and each vector has k components (as same as the number of Mel Frequency Cepstrum Coefficients).

Supposing that the codebook contains $L = 1$ vectors, then centroid c_1 can easily be calculated as the average of all the vectors, see Formula 3.8.The distortion measure is given by Formula 3.9.

$$c_1 = \frac{1}{M} \sum_{m=1}^{M} x_m \tag{3.8}$$

$$D_{average} = \frac{1}{M.k} \sum_{m=1}^{M} ||x_m - c_1||^2 \tag{3.9}$$

The distortion when there are only one centroid is large, hence the *model* will not be a good representation of the speaker's voice. To decrease this distortion we needed to introduce more centroids. The algorithm used called the *Linde, Buzo and Grey (LBG)* design [8].

Once the first centroid is calculated it is feasible to split it, two multiplying by $(1 + \epsilon)$ and $(1 - \epsilon)$ where ϵ is a small value. The results are two centroids, c_1 and c_2. Centroid c_1 is updated to be the average of the vectors which are closest to centroid c_1, and the same process is performed for c_2. The update operation must be repeated until the variation of the centroids is as small as needed[7].

[7] In practice between 15 and 20 iterations are sufficient to determine the real centroid.

After calculating c_1 and c_2, we can repeat the *splitting* operation to obtain four centroids, and so on until we obtain L centroids (L must be a power of 2).

A summary of the vector quantization algorithm is presented in Figure 3.7, in the first subplot we can see a representation of the vectors [8]. It is important to notice that every vector is composed of k components and in the plot only two of these components are shown, hence the centroid we can see in this plot is not exactly the centre of the of the points in the signal. In the second sublplot the same signal is presented with one centroid, this is $L = 1$. Similarly, the third subplot presents the signal with two centroids ($L = 2$) and the last subplot is a comparison between the signal and centroids from two different speakers. In the figure the 5th and the 6th components of the features vectors are represented because corresponds to the peak of energy of the speech spectrum.

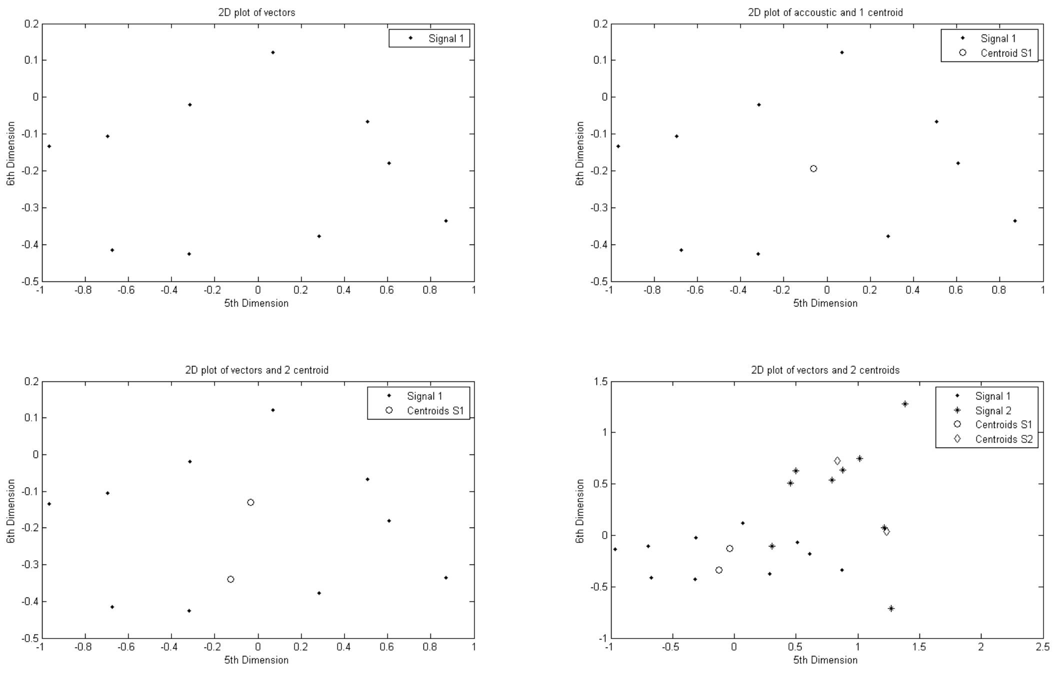

Figure 3.7. Representation of vectors and centroids.

Notice that the amount of information has been reduced from more than 3000 components to $k * L$ elements.

At this point we can calculate the *model* of each speaker, as explained in

[8]In a speaker recognition system these vectors are the Mel Frequency Cepstrum Coefficients of each frame.

Subsection 2.2. The training phase is finished when the *model* of the speaker is obtained. Now the question is: how can we distinguish models? The answer is explained in Subsection 3.2.2.

3.2.2 Euclidean Distance

In the *test phase* we need to measure the distance between the model from an unknown speaker's voice and the models previously calculated in the *training phase*.

Suppose we have two models from speakers A and B. Then a feasible approach to measure distance between A and B is to measure the *euclidean distance* between each feature vector from speaker A and its closest feature vector from speaker B, then *normalize* the distance by dividing by the number of vectors in the A model. Equation 3.10 shows how to compute this distance, where C_B^* is the closest vector to C_A^i belonging to $model_B$.

$$\begin{aligned} model_A &= (C_A^1, C_A^2, ..., C_A^N) \\ model_B &= (C_B^1, C_B^2, ..., C_B^N) \\ D_{A->B} &= \frac{1}{N}\sum_1^N ||C_A^i - C_B^*|| \end{aligned} \tag{3.10}$$

If the codebook length is 1 ($N = 1$), then each model is a vector containing the average of the mel coefficients from each frame, and the distance between models is the euclidean distance between the vectors. Thus we are now able to recognize the unknown speaker as one of the speakers who were enrolled during the training phase.

Chapter 4

Robust speaker Recoginition

If we apply algorithms explained in Chapter 3 we can obtain a good accuracy rate in *matched conditions*. Matched conditions exist when the speaker is located at the same relative position to the microphone during both the *training* and the *tes*t phases. But, what happens if we perform the *training* phase with the speaker close to the microphone and then we perform the *test* phase with the microphone in a different location? The answer to this question is that the accuracy rate decreases really fast. The further the speaker is from the microphone, the lowest the accuracy rate. Furthermore, if the speaker is not speaking directly into the microphone the accuracy rate will be even worse. To solve this problem we can apply Skosan and Mashao's "Modified Segmental Histogram Equalization for robust speaker verification" [27].

4.1 Standart techniques used to improve the robustness

In this section several techniques that have been used to improve the robustness in speaker recognition are presented.

Cepstral Mean Substraction (CMS) In this method the mean of the cepstral vectors is subtracted. It works like a high-pass filter. It was indicated that using CMS on clear speech decrease the accuracy[9], hence it is not useful in our case.

(RASTA) This method high-pass filter the the cepstral coefficients. It was indicated that this method was suitable for speech recognition, but when applied to speaker recognition removed significant portions of speaker information[9].

Feature Warping The aim of feature warping is to construct a more robust representation of each cepstral feature distribution. This method conforms the individual cepstral feature to follow a specific target distribution [9]. It was reported that this method increases the accuracy in mismatched conditions, hence we have included it in the project. A detailed explanation of this method is presented in Section 4.2.

4.2 Feature Warping

In this subsection an algorithm to increase the robustness is presented. The basic idea of the algorithm is to transform the features to get a desired probability distribution of the transformed features.

Assume that there are M feature vectors [1], $\{x_1, x_2, ..., x_M\}$. We study the variation of each component from the feature vector in time from the set of feature vectors from the utterance. Hence, we have a set of M values and we can compute the probability density function of these values.

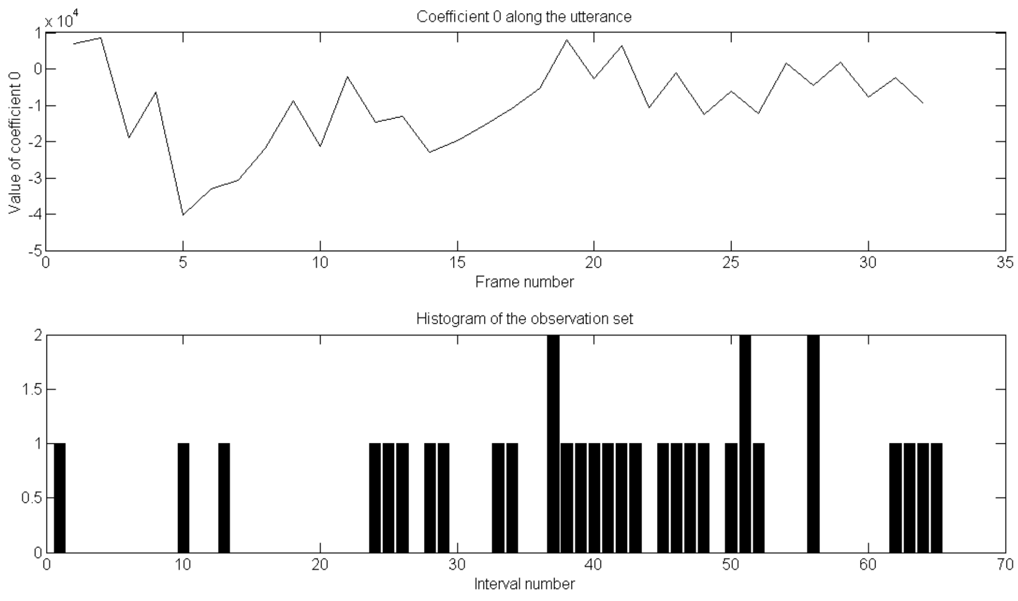

Figure 4.1. Variation of a component (above) and a histogram of these observations (bellow)

In the upper part of Figure 4.1 we can see the variation of the first component of the feature vector along a utterance that contains 35 frames. In the low subplot we have present an histogram of these observations. In order to calculate the histogram of observations we split the whole interval into 64 subintervals, then we have analysed how many values of the first component fall into each interval.

[1]These feature vectors are the result of Subsection 3.1.5

With this histogram, it is easy to calculate the cumulative distribution function by sorting the values and computing the probability of an observation being smaller than each value. See Formula 4.1, where X is the set of values of each component.

$$x \mapsto F(x) = P(X \leq x) \tag{4.1}$$

The goal now is transform this cumulative distribution to the desired distribution, in this case we want a normal distribution. Hence we have to transform each value x to a value y which has the same cumulative probability in a normal distribution. See Figure 4.2 and Formula 4.2 to clarify this transformation.

$$\int_{-\infty}^{x} C_x(x) = \int_{-\infty}^{y} C_{ref}(y) \tag{4.2}$$

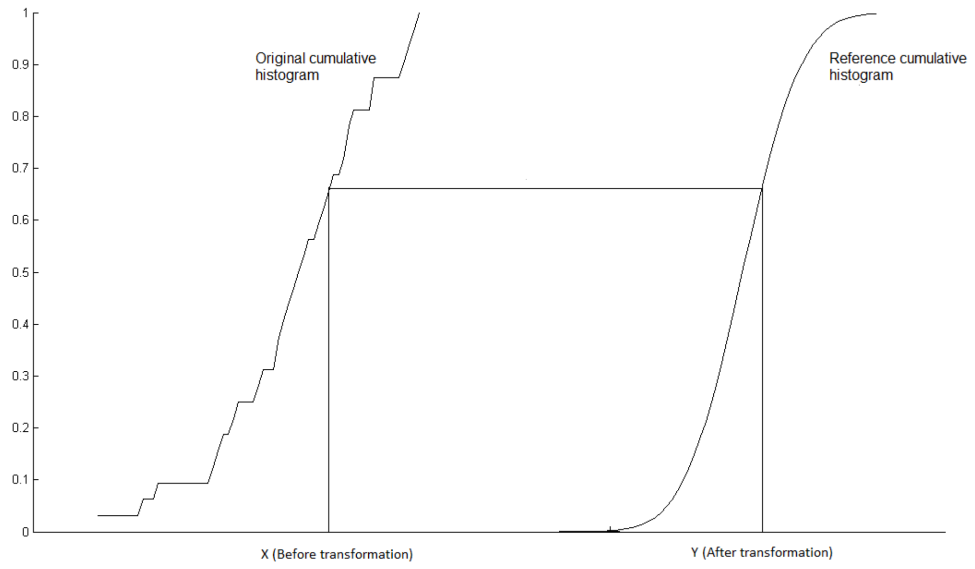

Figure 4.2. The cumulative distribution matching performed by HEQ [27].

In Figure 4.3 it is possible to see that the variation of the feature vector becomes sharper, hence it is easier to distinguish between the feature vectors from different speakers. A very similar technique is used in photography [12].

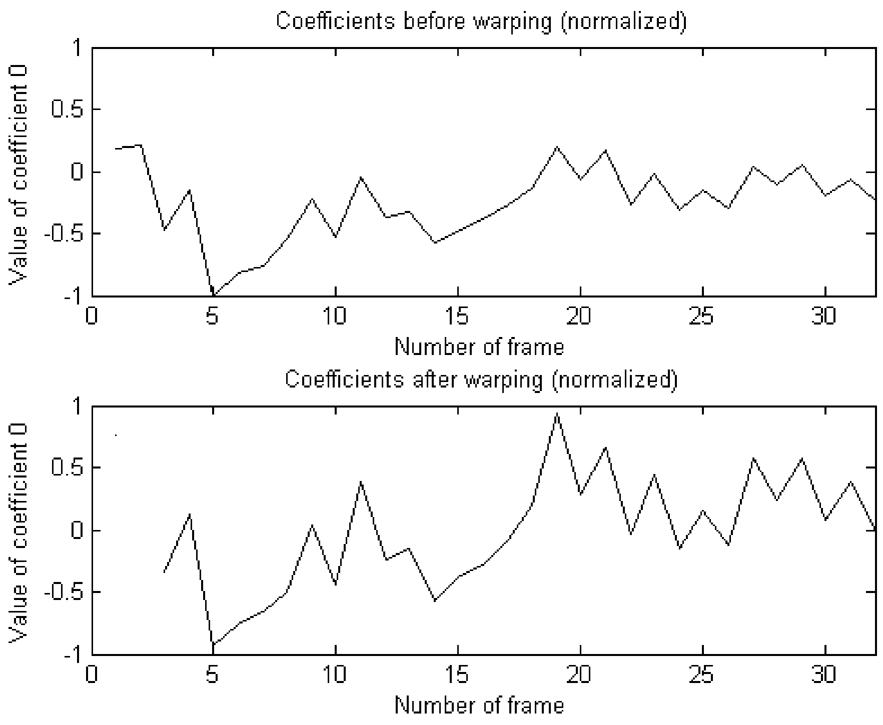

Figure 4.3. Variation of the first component of a set of feature vectors before and after performing feature warping.

Chapter 5

A speaker recognition system in C++

In Chapters 3 and 4 all the theoretical concepts have been explained in detail. This chapter presents and explains some of the main parts of the code of an application that has been developed during this master thesis project. C++ was chosen bor development, because compiled C++ execute faster than Java or Python implementations of the same algorithms. To develop this application the *Microsoft Visual Studio 2008* Integrated Development Environment (IDE) has been used [17].

As the the application runs on a PDA , specifically the *HP iPAQ h5500*, which runs *Microsoft's Windows Mobile 2003* operative system, the Software Development Kit (SDK) for Windows Mobile 2003-based Pocket PCs has been useful [19].

5.1 Reading audio samples from the microphone

First of all we need to sample the voice based upon the output of a microphone. As the number of bits per sample is 16, we store each sample in a `short int`. The *Microsoft Foundation Class* (MFC) [16] provides easy access to the microphone through the class `HWAVEIN`.

The structure `PCMWAVEFORMAT` stores the parameters required to record sounds with a microphone. In the code below the encoding will be pulse code modulation (PCM), the sampling is mono (a single channel rather than stereo), the sampling rate is set to 11025 Hz, and each sample has 16 bits:

```

HWAVEIN Input;
PCMWAVEFORMAT Format;

//Record format
Format.wf.wFormatTag = WAVE_FORMAT_PCM;
Format.wf.nChannels = 1; //One channel
Format.wf.nSamplesPerSec = SamplesPerSecond; //11025Hz
Format.wBitsPerSample=16; //16 bits per sample
Format.wf.nAvgBytesPerSec=Format.wf.nChannels*Format.wf.
   nSamplesPerSec
        *Format.wBitsPerSample/8; //Bytes
Format.wf.nBlockAlign=Format.wf.nChannels*Format.
   wBitsPerSample/8; //Length of the sample
```

Next we open the device for recording, specifying a handler that processes the messages produced during the recording (i.e. when the device driver's buffer is full). The result of this operation is a `MMRESULTS` type value which contains a status code indicating success or the type of error. For more information about the possibles failures in the recording process see [18].

```

MMRESULT mRes=0;
mRes=waveInOpen(&Input,0,(LPCWAVEFORMATEX)&Format,
                (DWORD)this->m_hWnd,0,CALLBACK_WINDOW);
```

Once the device is open we need to allocate memory for a buffer to contain the samples from the device and to prepare the buffer for waveform input. This can be done with the following code, where `LGbuf` contains the size of the buffer in bytes.

```

HGLOBAL IdCab,IdBuf;
LPWAVEHDR Head;
LPSTR Buffer;

IdCab=GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,sizeof(WAVEHDR));
Head=(LPWAVEHDR)GlobalLock(IdCab);

IdBuf=GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,LGbuf);
Buffer=(LPSTR)GlobalLock(IdBuf);

Head -> lpData = Buffer;
Head -> dwBufferLength = LGbuf;

mRes = waveInPrepareHeader(Input,Head,sizeof(WAVEHDR));
```

Finally, we must pass this buffer to the device and start recording.

```

mRes = waveInAddBuffer(Input,Head,sizeof(WAVEHDR));
mRes=waveInStart(Input);
```

When the buffer is full an `MM_WIM_DATA` message is automatically generated, hence we have to handle this message. First we capture the message defining the *message map* as follows:

```

BEGIN_MESSAGE_MAP(CTestPhaseDlg, CDialog)
#if defined(_DEVICE_RESOLUTION_AWARE) && !defined(
    WIN32_PLATFORM_WFSP)
        ON_WM_SIZE()
#endif
        //}}AFX_MSG_MAP
        ON_MESSAGE(MM_WIM_DATA,OnMM_WIM_DATA)//When the buffer
            is full
END_MESSAGE_MAP()
```

Then every time a `MM_WIN_DATA` message is received the method `onMM_WIM_DATA` is executed. In this later method we close the input and start processing the signal.

```

LRESULT CTestPhaseDlg::OnMM_WIM_DATA(UINT wParam,LONG lParam)
{
        waveInUnprepareHeader(Input,Head,sizeof(WAVEHDR));
        waveInClose(Input); //We close the input
        return 0;
}
```

As each sample contains 16 bits, we can store it in a `short int`, we can access the buffer as short integers with the following lines of code:

```

short int *Buffer16;
Buffer16 = (short int *)Buffer;//Each sample is a short int
```

Now all the samples are accesible from the `Buffer16` as `short int`. At this point the sampling (as described in Subsection 3.1.1) is finished and we start processing the digital speech signal.

In some speaker recognition applications it is necessary to record samples a long time (i.e., longer than a single buffer can contain). In this situation we use a *circular buffer*. In order to develop this circular buffer we must define several buffers, and each time one buffer is full we store the samples in the next buffer while we process the buffer that has just been filled [28] [20].

5.2 Fixed point

After sampling the speech we need to suppress silence (Subsection 3.1.2). Hence, we need *real numbers*, that can be stored as a `float` in C++. The problem is that most handheld devices do not have a *floating-point unit* and it takes a long time to perform operations on floating point numbers [1].

To solve this problem we have used *fixed point*. This approach uses a fixed number of bits to represent real numbers. Some of the bits represent the decimal part and other bits represent the integer part [21].

The approach uses a *scaling factor* $R = 2^{16}$ and the result is stored in a `long`. The scaling factor could be any value, it can even change during the computation. For example, suppose the real number is 2.45. In fixed point this number could be represented as $160563 = round(2.45 * R)$. It is most efficient to use a power of 2 as the scaling factor because then multiplication is just a *shift* operation. Note, that this is only an approximation and real numbers smaller than $\frac{1}{R}$ can not be represented.

In order to perform operations on *fixed point* a C++ class has been developed and all the required operations have been redefined. The following lines of code show part of the header of this class named `Fixed`. The complete definition of this class is given in Appendix A.

```

#ifndef FIXED_H
#define FIXED_H
class Fixed
{
private:

        long m_nVal;

public:
        Fixed& operator=(float floatVal);
        Fixed& operator=(Fixed fixedVal);
        Fixed operator*(Fixed b);
        Fixed operator-(Fixed b);
        Fixed operator+(Fixed b);

};
#endif
```

[1]A first version of the system was developed using floats and it took more than 15 seconds to process a two second utterance

As we can see in the code, the class contains a `long` attribute called `m_nVal`. Such a variable can store 32 bits per number. Furthermore, we can see the most important operations, in the full code there are several more operations in order to perform operations on different types of numbers such as `int` or `short int`.

The implementation in fixed point of the most important operations in signal processing $(+, -, *)$, are explained in detail below.

```

#include "stdafx.h"
#include "Fixed.h"

#define RESOLUTION          65536L
#define RESOLUTION_BITS         16

Fixed Fixed::operator+ (Fixed b)
{
        Fixed a;
        a.m_nVal = m_nVal+b.m_nVal;
        return a;
}
Fixed Fixed::operator- (Fixed b)
{
        Fixed a;
        a.m_nVal = m_nVal-b.m_nVal;
        return a;
}
Fixed Fixed::operator*(Fixed b)
{
        Fixed a;
        a.m_nVal =((( long)m_nVal*b.m_nVal)>>RESOLUTION_BITS);
        return a;
}
```

Hence, we can perform as many operations as required in fixed point, keeping in mind that this is an approximation and error is induced in each operation.

Finally it is important to ensure that the result of the operations is not larger than the maximum value that we can represent with our current fixed point representation, in our case this 2^{16}. If the value is larger we need to eliminate the least relevant bits (this is the same as changing the *scaling factor*). In our system we have studied the worst case, this is: *what is the maximum number of bits needed in each part of the process?*. The results are shown in Table 5.1.

As there are never more than 32 bits needed in any part of the computation we have avoided *overflow*[2]. It is important to consider the size of the result of every

[2]A situation where the data value exceeds that which can be stored.

Table 5.1. Major number of bits in each state.

State	Bits for integer part	Bits for decimal part	Total
Input	0	16	16
Hamming Window	0	16	16
Power spectrum	15	16	31
Mel coefficients (log)	13	16	29
DCT	11	16	27

operation because otherwise *overflow* might occur and the result will make no sense.

5.3 Look up tables

The next step in the our speaker recognition system 3.1 consists of Hamming windowing. It is not efficient to calculate the Formula 3.3 for every frame, because the result for a given input value is always the same and because is difficult to compute it in fixed point. Instead we precompute the result for the 256 points and store these results in a look up table.

The following lines of MATLAB [14] code produces the lookup table that will be included in the C++ code. Notice that the constructor `Fixed(true, value)` returns a `Fixed` value whose attribute `m_nVal` is *value*. Following this we show the first few lines of resulting `HammingTable[]` that will be inserted into the C++ code.

```

RESOLUTION = 65536;
hw = hamming(256);
for i=1:156
    msg =  sprintf('Fixed(true, %.0f),', round(hw(i)*
       RESOLUTION));
    disp(msg);
end
```

```
const Fixed HammingTable[n] = {
        Fixed(true, 5243),
        Fixed(true, 5252),
        Fixed(true, 5279),
        Fixed(true, 5325),
        Fixed(true, 5389),
        ...};
```

Now *windowing* is simply implemented as a loop multiplying the value stored in `frame[i]` by `HammingTable[i]`. Furthermore, using look up tables is much more efficient than performing arithmetic computations in the PDA.

Another look up table is used to perform the Fast Fourier Transform. The FFT algorithm will be explained in detail in the following sections. . To facilitate this computation we need to precompute a look up table containing $sin(\frac{-2.\pi}{n})$, where n is an integer smaller than 512. To clarify the need fir the *sine* look up table see Section 5.4.

We will use a final look up table containing the coefficients of the filters in the filterbank (described in Subsection 3.6). Hence, in order to obtain the mel coefficients we simply multiply each power spectrum frame value with each filter stored in the look up table.

5.4 Fast Fourier and Discrete Cosine Transforms in C++

As explained in Subsection 3.1.4 it is important to calculate the Fourier Transform efficiently. The algorithms from the*Numerical recipes in C* book [22] has been adapted to work on *fixed point* (using the functions described in Subsection 5.2).

Listing 5.1. Fast Fourier Transform algorithm in fixed point (adapted from [22]).

```
/*Calculate the Fourier transform of a set of n-real valued data points. Replaces the
 *data (which is stored in array data[0..n-1]) by the positive frequency half of its complex
 *Fourier Transform. Data[0] -> Real Part of F_0, Data[1] -> Real part of F_N/2 Real part -> Data[even], Imaginary part -> Data[odd] */
 void FastFourierTransform::realft(Fixed data[], unsigned long n, int isign) {
         unsigned long i,i1,i2,i3,i4,np3;
         Fixed c1=0.5f,c2,h1r,h1i,h2r,h2i, wr,wi,wpr,wpi,wtemp,theta;

         if (isign == 1) {
                 c2 = -0.5f;
                 f.four1(data,n>>1,1);
         } else
                 c2=0.5f;
         wtemp=Fixed(-1.0f)*Fixed(isign)*sinetable[n<<1];//sin(0.5*theta);
         wpr = Fixed(-2.0f)*wtemp*wtemp;
         wpi=Fixed(-1.0f)*Fixed(isign)*sinetable[n];//sin(theta);
         wr=Fixed(1.0f)+wpr;
         wi=wpi; np3=n+3;
         for (i=2;i<=(n>>2);i++) {
                 i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
                 h1r=c1*(data[i1-1]+data[i3-1]);
                 h1i=c1*(data[i2-1]-data[i4-1]);
                 h2r = Fixed(-1.0f)*c2*(data[i2-1]+data[i4-1]);
                 h2i=c2*(data[i1-1]-data[i3-1]);
                 data[i1-1]=h1r+wr*h2r-wi*h2i;
                 data[i2-1]=h1i+wr*h2i+wi*h2r;
                 data[i3-1]=h1r-wr*h2r+wi*h2i;
                 data[i4-1] = Fixed(-1.0f)*h1i+wr*h2i+wi*h2r;
                 wr=(wtemp=wr)*wpr-wi*wpi+wr;
                 wi=wi*wpr+wtemp*wpi+wi;
         }
         if (isign == 1) {
                 data[0] = (h1r=data[0])+data[1];
                 data[1] = h1r-data[1];
         } else {
                 data[0]=c1*((h1r=data[0])+data[1]);
                 data[1]=c1*(h1r-data[1]);
                 f.four1(data,n>>1,-1);
         }
}
```

The line 19 shows that we need to calculate $sin(\frac{\theta}{2})$, where $\theta = \frac{2.\pi}{n}$ and $n \in [2, 512]$. Hence, we can avoid this operations with the `sinetable[n]` look up table as explained in Subsection 5.3:

$$\texttt{sinetable[n]} = sin(\frac{-2.\pi}{n}) \tag{5.1}$$

This look up table is used in other parts of the complete code in the same way. The next listing is the adaptation to fixed point of the Discrete Cosine Transform algorithm. Note that this method also uses the method explained in Listing 5.1.

Listing 5.2. Discrete Cosine Transform algorithm in fixed point (adapted from [22].

```
void FastFourierTransform::cosft32(Fixed y[], int isign){
        int n = NUMBEROFFILTERS;         int i;
        FastFourierTransform f;
        void realft(Fixed data[], unsigned long n, int isign);
        Fixed sum,sum1,y1,y2,ytemp,theta,wi=0.0f,wi1,wpi,wpr,
            wr=1.0f,wr1,wtemp;
        //theta=0.5*PI/n;
        wr1=Fixed(true, cosPI64);//cos(theta);
        wi1=Fixed(-1.0f)*sinetable[n<<2];//sin(theta);
        wpr = Fixed(-2.0f)*wi1*wi1;
        wpi=Fixed(-1.0f)*sinetable[n<<1];//sin(2.0*theta);
        if (isign == 1) {
                for (i=1;i<=n/2;i++) {
                        y1=Fixed(0.5f)*(y[i-1]+y[n-i]);
                        y2=wi1*(y[i-1]-y[n-i]);
                        y[i-1]=y1+y2;
                        y[n-i]=y1-y2;
                        wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
                        wi1=wi1*wpr+wtemp*wpi+wi1;
                }
                f.realft(y,n,1);
                for (i=3;i<=n;i+=2) {
                        wr=(wtemp=wr)*wpr-wi*wpi+wr;
                        wi=wi*wpr+wtemp*wpi+wi;
                        y1=y[i-1]*wr-y[i]*wi;
                        y2=y[i]*wr+y[i-1]*wi;
                        y[i-1]=y1;
                        y[i]=y2;
                }
                sum=Fixed(0.5f)*y[2];
                for (i=n;i>=2;i-=2) {
                        sum1=sum;
                        sum = sum + y[i-1];
                        y[i-1]=sum1;
                }
        } else if (isign == -1) {
                ytemp=y[n];
                for (i=n;i>=4;i-=2) y[i]=y[i-2]-y[i];
                y[1]=Fixed(2.0f)*ytemp;
                for (i=3;i<=n;i+=2) {
                        wr=(wtemp=wr)*wpr-wi*wpi+wr;
                        wi=wi*wpr+wtemp*wpi+wi;
                        y1=y[i-1]*wr+y[i]*wi;
                        y2=y[i]*wr-y[i-1]*wi;
                        y[i-1]=y1;
                        y[i]=y2;
                }
                f.realft(y,n,-1);
```

```
                    for (i=1;i<=n/2;i++) {
                            y1=y[i-1]+y[n-i];
                            y2=(Fixed(0.5f).divide(wi1))*(y[i-1]-y
                                [n-i]);
                            y[i-1]=Fixed(0.5f)*(y1+y2);
                            y[n-i]=Fixed(0.5f)*(y1-y2);
                            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
                            wi1=wi1*wpr+wtemp*wpi+wi1;
                    }
            }
            for(int i=0; i<n; i++){
                    if(i==0)
                            y[i] = y[i]*Fixed(true, 11585)*Fixed(
                                true,92681);
                    else
                            y[i] = y[i].divide(Fixed(true,
                                2621440));//
            }
}
```

5.5 Calculating Mel Filterbank

In section 3.1.5 the theoretical concept of Mel Frequency Cepstrum Coefficients (MFCC) was explained. The question is: *How can we calculate the filters to obtain the MFCC?*. As the result is stored in a look up table, we can precompute the filters using MATLAB, because it is faster and easier than computing the coefficients at runtime on the PDA.

A function has been developed in order to obtain the filters simply by specifying the sampling rate, the number of filters required, and the number of points in the DFT. The code of this function appears below. The algorithm splits the frequency sampled in the mel scale in `nFilters` slices, then transforms each slice into the frequency scale yielding an index in the DFT domain. Finally each filter starts in the previous centre and finishes in the next centre in a triangle shape.

Listing 5.3. Calculating the filterbank.

```
function filters = melfilterbank(nFilters, N, sf)
%nFilters -> number of filters
%N -> number of points in the FFT
%sf -> sampling rate
%max frequency on the mel scale
melmax = freq2mel(sf/2);
%increment on the mel scale
melinc = melmax./(nFilters+1);
%index of the centers on the frequency scale
indexcenter = round(mel2freq((1:nFilters).*melinc)./(sf/N));
%index of the starts and stops on the frequency scale
indexstart = [1, indexcenter(1:nFilters-1)];
indexstop = [indexcenter(2:nFilters), N/2];
filters = zeros(nFilters, N/2);
for c = 1:nFilters
    %left side
    increment = 1.0/(indexcenter(c) - indexstart(c));
    for i = indexstart(c):indexcenter(c)
        filters(c,i) = (i - indexstart(c))*increment;
    end
    %right side
    decrement = 1.0/(indexstop(c) - indexcenter(c));
    for i = indexcenter(c):indexstop(c)
       filters(c,i) = 1.0 - ((i - indexcenter(c))*decrement);
    end
end
function b = mel2freq (m)
% compute frequency from mel value
b = 700*((10.^(m ./2595)) -1);
function m = freq2mel (f)
% compute mel value from frequency f
m = 2595 * log10(1 + f./700);
```

Finally, as explained in Section 5.3 we include the values of the filters as a look up table and multiply each FFT result by each filter obtaining the MFCC. The following lines of code shows how to do it if the look up table `FILTERBANK[]` contains all the filters placed consecutively and `spectrum` contains the FFT of the frame.

```
for (int i=0; i<NUMBEROFFILTERS; i++){
        for(int j=0; j<HALFN; j++){
                MelFrequencyCepstrumCoefficients[i] =
                    MelFrequencyCepstrumCoefficients[i] +
                    espectrum[j]*FILTERBANK[i*HALFN+j];
        }
}
```

5.6 Vector Quantization algorithm

The vector quantization algorithm used is the Linde, Buzo, and Gray (LBG) approach. Assuming that the MCFCs are stored in an array accessed by double pointer, each row contains the MCFC from a frame, and there are as many rows as frames in the utterance. A method has been developed in this project to return as many centroids as needed [3] in an array accessed by double pointer. This is useful because we can easily vary the number of centroids and study the difference in the accuracy rate. The algorithm follows the explanation given in the next listing.

[3] The number of centroids must be a power of 2

```
/*
* length -> Number of frames in the utterance
* N MUST BE A POWER OF 2
*/
Fixed ** VectorQuantization::getCentroids(Fixed ** mfcc, int length, int n)
{
        Fixed *** tempgroups = new Fixed**[n];//Stores the vectors belonging to a centroid
        for(int i=0; i<n; i++){
                tempgroups[i] = new Fixed*[length];
                for(int j=0; j<length; j++){
                        tempgroups[i][j] = new Fixed[NUMBEROFFILTERS];
                }
        }
        //count of the vectors in each centroid
        int * tempcount = new int[n];
        //pointer to the centroids
        Fixed ** centroids = new Fixed*[n];
        for(int i=0; i<n; i++){
                centroids[i] = new Fixed[NUMBEROFFILTERS];
        }
        //first centroid is the average;
        for(int j=0; j<length; j++){
                for(int i=0; i<NUMBEROFFILTERS; i++){
                        centroids[0][i] = centroids[0][i] + mfcc[j][i].divide(length);
                }
        }
        if(n == 1){
                return centroids;
        }
        //start iterating
        int centroidsCalculated = 1;

        while(true){
        for(int j=0; j<NUMBEROFFILTERS; j++){
                for(int k=centroidsCalculated; k>0; k--){
                        //split each vector already calculated
                        centroids[2*k-1][j] = centroids[k][j]*me;
                        centroids[2*k-2][j] = centroids[k][j]*pe;
                }
        }
        for(int l=0; l<ITERATIONS; l++){
                for(int i=0; i<2*centroidsCalculated;i++)
```

```
                                tempcount[i]=0;
                        for(int i=0; i<length; i++){
                                Fixed mindistance = MAXFIXED;
                                Fixed * tempdist = new Fixed[2*centroidsCalculated];
                                int min = 0;
                                for(int j=0; j<2*centroidsCalculated; j++){
                                        tempdist[j] = eu.getDistance(mfcc[i], centroids[j]);
                                        if(tempdist[j]<mindistance){
                                                mindistance= tempdist[j];
                                                min=j;
                                        }
                                }
                                tempgroups[min][tempcount[min]] = mfcc[i];
                                tempcount[min]++;
                        }
                        //update the temp centroid
                        for(int k=0; k<2*centroidsCalculated; k++){
                                for(int j=0; j<tempcount[k]; j++){
                                        for(int i=0; i<NUMBEROFFILTERS; i++){
                                                centroids[k][i] = centroids[k][i] + tempgroups[k][j][i].divide(tempcount[k]);
                                        }
                                }
                        }
                }
                centroidsCalculated = centroidsCalculated*2;
                if(centroidsCalculated == n)
                        return centroids;
                }
return centroids;
}
```

It is important to notice that there is a variable called `ITERATIONS`, the bigger this value is, the better the centroids are. The number `ITERATIONS` represents *how many times a centroid is updated to reach its final value.*

5.7 Feature warping algorithm

This algorithm follows the explanation given in Section 4.2 and utilizes a look up table containing the *inverse of the cumulative normal distribution function.* This look up table is called `NORMINV` in the code below. This method also uses a sorting algorithm. An implementation of the quicksort algorithm is used because it is much more efficient than other sorting algorithms (such as bubble sort), but the code does not appear because it is not specific in this thesis. For more information about the *quicksort algorithm* see [25]. The rest of the algorithm appears in the next listing.

```
void FeatureWarping::warpFeatures(Fixed ** melCoefficients, int length){
        int M = 64; //Number of intervals in the histogram of observation
        for(int k=0; k<NUMBEROFFILTERS; k++){
                Fixed * arr = new Fixed[length];
                for(int i=0; i<length; i++){
                        arr[i] = melCoefficients[i][k];
                }
                int * histogram = new int[M];
                for(int i=0; i<M; i++)
                        histogram[i]=0;
                quick(arr, 0, length-1);//sort the numbers
                Fixed min = arr[length-1];
                Fixed intervalSize = (arr[0]-arr[length-1]);
                long intervalSizel = intervalSize.m_nVal/M;
                intervalSize = Fixed(true, intervalSizel);
                //Histogram of observation with M intervals
                for(int i=0; i<length; i++){
                       arr[i] = arr[i]-arr[length-1];
                       int index = arr[i]/intervalSize;
                       histogram[index]++;
                }
                //Cumulative histogram of original variable
                for(int i=1; i<M; i++){
                        histogram[i]= histogram[i]+histogram[i-1];
                }
                Fixed *histogramF = new Fixed[M];
                int *normInvIndex = new int[M];
                for(int i=0; i<M; i++){
                        histogramF[i]= Fixed((float)histogram[i]/length);
                        normInvIndex[i] =  histogramF[i].m_nVal/256;
                }
                for(int i=0; i<length; i++){
                    melCoefficients[i][k] = melCoefficients[i][k]-min;
                     int index = arr[i]/intervalSize;
                    melCoefficients[i][k] = NormInv[normInvIndex[index]];
                }
        }
}
```

5.8 Distance between models

Finally we can obtain the feature vectors from each speaker's voice, each of these is a speaker model. To match an unknown voice against these models we need to measure the distance between models. Suppose we have two arrays each accessed by double pointers, these arrays contain the feature vectors from two different speakers A and B. We calculate the *euclidean distance* between each feature vector from speaker A with its closest feature vector from speaker B and then add up the results. The value of this sum is the distance between A and B. The listing below shows how to compute this distance, where the method `getDistance` returns the euclidean distance between two vectors.

Listing 5.4. Distance between models

```

/*
 * Calculates the Euclidean distance between two groups of
    feature vectors with lengths lengthA and lengthB
 */

Fixed EuclideanDistance::getTotalDistance(Fixed ** mfccA, int
   lengthA, Fixed ** mfccB, int lengthB)
{
        Fixed totalDistance =0;
        Fixed minimumDistance = MAXFIXED;
        Fixed distance = 0;
        for (int i=0; i< lengthA; i++){
         minimumDistance = MAXFIXED;
        for (int j=0; j< lengthB; j++){
           distance = getDistance(mfccA[i], mfccB[j]);
            if(distance < minimumDistance){
                minimumDistance = distance;
              }
        }
                totalDistance = totalDistance +
                   minimumDistance.divide(lengthA);
        }
        return totalDistance;
}
```

It is important to *normalize*, this is to divide the result by `lengthA`, if we do not normalize a short utterance will have a small distance even if it is compared to a different speaker. In the *training phase* a model of each speaker is stored, after that, during the *test phase* we compute the model of the unknown speaker and calculate the distance to all the speakers in the system as explained earlier. Finally, the claimant is recognized as the speaker whose model has the smallest distance from the unknown model.

Chapter 6

Results

The final application has been tested in a *HP iPAQ 5500*. The features of this device are shown in Table 6.1.

Table 6.1. Features of the HP iPAQ 5500.

Feature	
Processor	Intel XScale 400 MHz
RAM (Random Access Memory)	128 MB SDRAM
ROM (Read Only Memory)	48 MB ROM
Microphone	Integrated microphone up to 44.1 KHz sampling rate with 16 bits per sample
Connections	Integrated wireless LAN (802.11b)
Operative System	Windows Mobile 2003

There are two main questions to answer in this chapter, first of all: *How many users is it feasible to distinguish?* and finally: *How long does it take to distinguish a user?*. The answer to these questions are presented in the following sections.

In order to perform the tests a simple database of speakers from the *Image Formation & Processing Group at the University of Illinois* has been used. The database contains 8 speakers, all of them uttered the same word "zero"[1] once in a training session and once in a test session. The sessions were at least 6 months apart to take in account variations of the voice with time. As the database only contains two sessions, the first session is used in the training phase and the second session is used in the first attempt to evaluate the test phase.

[1]"Zero" is a good word for speaker recognition because the *z* sound is voiced, and contains a lot of energy. Furthermore the word contains the two most used vowels *e* and *o*.

6.1 Accuracy measurements

The accuracy has been measured in two different conditions: first when the training and test phase occur with the mouth in the same position with respect to the microphone. The result is that the first situation is an analysis of accuracy under *matched conditions*. In the second analysis of accuracy the training and test phase do not have the same relation of the mouth with respect to the microphone, this evaluates the accuracy under *mismatched conditions*. Figure 6.1 show where the microphone was located during test phase. The training phase always used voice samples taken in *Spot1*, hence *Spot2*, *Spot3* and *Spot4* represent mismatched conditions while *Spot1* in the test phase represents matched conditions. In all case the microphone used was a *Cosonic CD-610MV*. Figure 6.2 shows how the microphone was oriented.

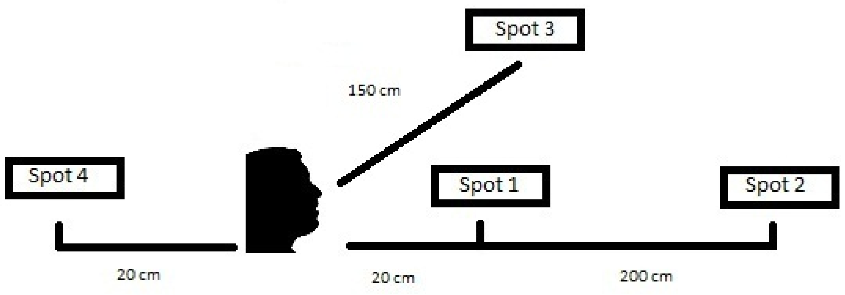

Figure 6.1. Different spots where microphone was placed during the tests.

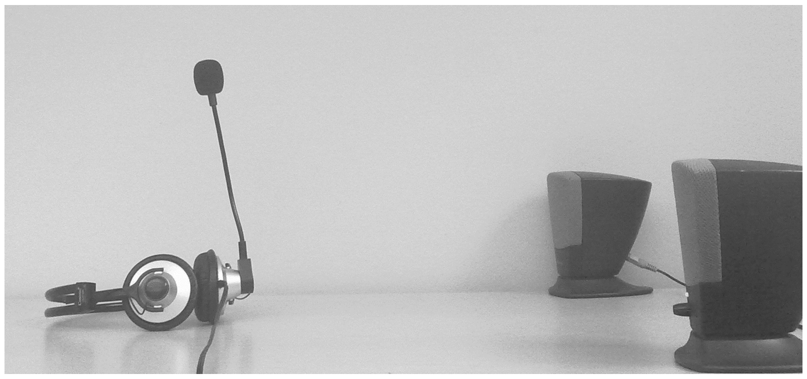

Figure 6.2. Picture of a test phase under matched conditions.

Table 6.2 shows the results for each of the accuracy tests that have been performed. Column *Codebook length* show the number of vectors in the speaker model, the next column show how many filters were there in the filterbank, and finally the columns for *SpotX* show how many users from the database (of 8 speakers) could be properly recognized in each spot. The training and test phases each took about 1 second. This first table does not use feature warping.

Table 6.2. Accuracy rate in different spots and conditions without feature warping.

Codebook length	N. of filters	Spot1	Spot2	Spot3	Spot4
1	16	5	2	2	3
4	16	5	2	2	3
16	16	5	2	2	3
1	32	7	4	3	2
4	32	6	3	2	2
16	32	6	3	3	3

The same tests were repeated using feature warping, but the results are not good (See Table 6.3). In matched conditions we can distinguish fewer users than without using it. Furthermore, in missmatched conditions the result of using feature warping are not better than without using feature warping.

Table 6.3. Accuracy rate in different spots and conditions with feature warping.

Codebook length	N. of filters	Spot1	Spot2	Spot3	Spot4
1	32	3	-	1	2
4	32	2	-	1	2
16	32	2	1	2	2

The tests were all made in a lab environment, where there is computer noise and other people maybe speaking (softly). The utterances from the database have been played out using common speakers in order to systematically include the noise of the laboratory when the PDA is recording.

Although all the test shown in this thesis have been performed with a sampling rate of 11025 Hz. Additional test were made with various sampling rate ranging from 8000 Hz to 12500 Hz, however, there is no noticeable difference in results.

6.2 Performance measurements

The second question, *How long does it take to distinguish a user?* is answered in this section. Table 6.4 show how long it takes to process a utterance, from when the utterance is completely uttered until we obtain the speaker model under different conditions. The sampling rate is set to 11025 Hz.

Table 6.4. Processing time without feature warping (msec).

CODEBOOKLENGTH	Number of filters 16	Number of filters 32
1	747	908
2	765	990
4	828	1187
8	1630	1724
16	1763	1932

The time shown in Table 6.4 includes finding the utterance within a period of two seconds, obtaining the codebook with the specified length, and writing the results into a file. Performing *feature warping* requires about 20 milliseconds more time in the worst case. Finally we need to measure the distance between models, this takes 12 milliseconds per user enrolled in the system. Thus with 8 users, computing the minimum distance with respect to all of the models takes less than 100 milliseconds.

6.3 Required size in RAM

The application needs to store the model from each speaker. These stored models are needed the next time an unknown user's utterance is to be analysed. Each model needs $codebookLength * numberOfFilters * 32$. For example, for 8 users, a codebook of length 1, and 32 filters the total model storage is 8192 bits or 1024 bytes.

During the test phase the models were loaded in the Random Access Memory (RAM) in order to be able to access them quickly. Furthermore, the various look up tables must be also loaded in the RAM. Table 6.5 shows the size of each look up table. It takes about 200 nanoseconds to read a value from a look up table.

The whole application needs 1.476 MB of storage[2] (including a simple graphic interface). When the program is running needs 0.86 MB in RAM.

[2]This storage corresponds to the ".exe" file of the application.

Table 6.5. Size of the look up tables.

Look up table	Size in bits
Hamming Window	8192
Sine	16384
Mel filters	$numberOfFilters * 129 * 32$
Normal inverse	8192

Chapter 7

Conclusions and future work

In this chapter the conclusions are presented along with some suggestions about how to continue this work, specifically identifying *the next obvious things to be done.*

7.1 Conclusions

A speaker recognition system has been developed. It works properly in a handheld device when the number of users is limited (6 or 7 users) under matched conditions, highlighting that these matched conditions are the *worse* because there are 20 centimetres between the mouth and the microphone. Users using a headset can reach better accuracies.

The system needs only one word for the training phase, and the same word for the test phase. This is very good because the users can be recognised by the system quickly, and they do not need to spend a lot of time training the system.

The fixed point and look up table approaches work very well in handheld devices and reduced the processing time from more than 10 seconds to less than one second while having a similar accuracy rate.

I suggest using a 32 filter filterbank because increases the accuracy and does not require much more time (about 20 % more time) than when using 16 filters. It is important to note that the required storage space for each speaker is exactly double, hence some application might not be afford this amount of storage space.

I suggest using a codebook length of 1. In speaker recognition where the training and test phase is only one word, one vector of information is sufficient as seen in Table 6.2. Using a larger codebook length increases the processing time and the required model storage space, but does not increase the accuracy. Given the current processing time and memory requirements, this algorithm would be useful in a speaker recognition system even with a longer training or test phase.

The feature warping algorithm was not useful when the training and test phase each take less than one second. But Qin Jin [10] has reported increased accuracy in mismatched conditions by about 40 percent when the training phase takes about 10 seconds.

To summarize: a set of algorithms have been developed in C++ that are easily adaptable to another operating system given suitable computing power versus the required processing time and given sufficient space in memory. These algorithms can distinguish in about one second between a small set of users that trained the system based upon one second of audio captured by a handheld devices. These algorithms could also be used in systems where the training or test phase utilized longer utterances. Note that some of the processing could be overlapped with the acquisition of voice samples, hence reducing the apparent time of the computation. Thus this project met its goal of being able to distinguish a speaker from among family members or a small number of colleagues. If we remember Table 2.1 we can notice that a few years ago AT&T developed a speaker recognition system similar than ours (training phase = 0.5s, population = 10) using a large computer, and now we obtain similar results with a simple handheld computer.

7.2 Future work

The first future work to continue with this thesis project is to build some useful applications that exploit this speaker recognition system. The speaker recognition system alone is not useful, but it could be utilized by a number of applications. For example, an application could exploit the speaker recognition system to manage meetings, if the device recognises that a specific person is near the device, and the owner of the device is waiting to meet him/her, then the device could notify both persons that the meeting could start immediately. Another example, is automatically configuring the device with the users's calendar, favorite wall paper, etc. when this user's voice is heard. There are as many possible applications as you can imagine.

The system have been tested with 8 users, but it would be interesting to test the system with more users and different databases.

As these devices are more and more connected to *online* servers, another alternative solution to analyse is the feasibility of executing the speaker recognition system on a network attached server. Inmaculada Rangel Vacas in her thesis [30] explains how to establish a connection between the PDA and a laptop sending the captured audio data to the remote laptop. In this approach the speaker recognition system would execute on the server, by sending an audio stream to the server. Then when an utterance is detected, recognition is triggered at the server and the result sent back to the PDA. This approach could be faster, but requires connectivity and potentially more communication traffic from the PDA to the server - which will consume batter power and network bandwidth.

Finally the system could be improved in terms of robustness and noise. A lot of researchers have demonstrated that it is feasible to increase the accuracy in bad conditions, i.e. when there is a lot of noise. This research should be applied to this system to allow it to work in worse conditions.

Bibliography

[1] Mark D. Skowronski, “Windows Lecture”, from the course EEL 6586: Automatic Speech Processing, Computational Neuro-Engineering Lab, University of Florida February 10, 2003. `http://www.cnel.ufl.edu/~markskow/papers/windows.ppt`.

[2] Anil K. Jain,Ruud Bolle, and Sharath Pankanti. *Biometrics: Personal identification in networked society*, chapter 8, page 169. Kluwer Academic Publishers, Norwell, MA, USA, 4 edition, 1988.

[3] Bonnie C. Baker. Anti-aliasing, analog filters for data acquisition systems. Technical report, Microchip Technology Inc., August 1999. `http://ww1.microchip.com/downloads/en/AppNotes/00699b.pdf`.

[4] Charles A. Bouman. Lab 9a - speech processing (part 1). Technical report, Connexions, Rice University, Texas, September 2009. `http://cnx.org/content/m18086/1.3/`.

[5] Minh N. Do. DSP mini-project: An automatic speaker recognition system. Technical report, Department of Electrical and Computer Engineering University of Illinois at Urbana-Champaign, 2003. `http://www.ifp.illinois.edu/~minhdo/teaching/speaker_recognition/speaker_recognition.html`.

[6] Mireia Farrús, Michael Wagner, Daniel Erro, and Javier Hernando. Automatic speaker recognition as a measurement of voice imitation and conversion. *International Journal of Speech Language and the Law*, 17(1), 2010.

[7] Sadaoki Furui. 40 years of progress in automatic speaker recognition. In Massimo Tistarelli and Mark Nixon, editors, *Advances in Biometrics*, volume 5558 of *Lecture Notes in Computer Science*, pages 1050–1059. Springer Berlin/ Heidelberg, 2009.

[8] H. B. Kekre, and Tanuja K. Sarode. Speech data compression using vector quantization. World Academy of Science, Engineering and Technology, 2008. `http://www.akademik.unsri.ac.id/download/journal/files/waset/v2-4-37-5.pdf`.

[9] Jason Pelecanos, and Sridha Sridharan. Feature warping for robust speaker verification, 2001.

[10] Qin Jin. Robust speaker recognition. P.h.dissertation, Language Technologies Institute, School of Computer Science, Carnegie Mellon University. `http://www.lti.cs.cmu.edu/Research/Thesis/QinJin.pdf`, January 2007.

[11] Asahi Kasei. Ak4535 16bit codec with mic/hp/spk-amp. `http://stomach.v2.nl/docs/Hardware/PDA/iPAQ/h5xxx/AK4535-MS0135-E-01-03-2002.pdf`, March 2002.

[12] Y. Kim. Quantized bi-histogram equalization. In *ICASSP '97: Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97) -Volume 4*, page 2797, Washington, DC, USA, 1997. IEEE Computer Society.

[13] R. Lyons. Windowing functions improve FFT results. In *Test and Measurement World, Parts I and II*, TRW, Sunnyvale, CA, september 1998. `http://www.tmworld.com/article/322450-Windowing_Functions_Improve_FFT_Results_Part_I.phpf`.

[14] The MathWorks, Inc. *Matlab*, March 2010. `http://www.mathworks.com/`.

[15] Md. Rashidul Hasan, Mustafa Jamil, Md. Golam Rabbani, and Md. Saifur Rahman. Speaker identification using mel frequency cepstral coefficients. In *3rd International Conference on Electrical & Computer Engineering ICECE 2004*, pages 565–568, Dhaka, Bangladesh, December 2004. Electrical and Electronic Engineering, Bangladesh University of Engineering and Technology. `http://www.buet.ac.bd/eee/icece2004/P141.pdf`.

[16] Microsoft Corporation. *Microsoft Foundation Classes*, September 2008. `http://www.microsoft.com/`.

[17] Microsoft Corporation. *Visual Studio*, 2008. `http://msdn.microsoft.com/es-es/vstudio/default.aspx`.

[18] Microsoft Corporation. *Microsoft Developer Network*, October 2009. `http://msdn.microsoft.com`.

[19] Microsoft Corporation. *SDK for Windows Mobile 2003-based Pocket PCs*, September 2009. `http://www.microsoft.com/`.

[20] Ali Nesh Nash. Voice over ip in a resource constrained environment. Master's thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, Stockholm, Sweden, March 2006. COS/CCS 2006-06, `http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/060317-Ali_Nesh-Nash-report-with-cover.pd`.

[21] Erick L. Oberstar. Fixed-point representation & fractional math. Technical report, Oberstar Consulting, Madison, WI, 2007. `http://darcy.rsgc.on.ca/ACES/ICE4M/FixedPoint/FixedPointRepresentationFractionalMath.pdf`.

[22] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing.* Cambridge University Press, New York, NY, USA, 1992.

[23] Thomas F. Quatieri. *Discrete-Time Speech Signal Processing: Principles and Practice*, chapter 14. Prentice Hall Signal Processing Series. Prentice Hall, 2001.

[24] Alexander Riedel. Collaborative scheduling using context-awareness. Master's thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, Stockholm, Sweden, March 2010. TRITA-ICT-EX-2010:35, `http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/100325-Alexander_Riedel-with-cover.pdf`.

[25] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[26] Yang Shao, S. Srinivasan, and DeLiang Wang. Incorporating auditory feature uncertainties in robust speaker identification. volume 4, pages IV–277 –IV–280, apr. 2007.

[27] Marshalleno Skosan and Daniel Mashao. Modified segmental histogram equalization for robust speaker verification. *Pattern Recognition Letters*, 27(5):479 – 486, 2006.

[28] Mindfire Solutions. Starting with VoIP. Technical report, Mindfire Solutions, India, March 2002. `www.mindfiresolutions.com/mindfire/Technical_Voice_over_IP.pdf`.

[29] Johan Sverin. Speech interface for a mobile audio application. Master's thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, Stockholm, Sweden, July 2005. IMIT/LCN 2005-17, `http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/050702-Johan_Sverin-with-cover.pdf`.

[30] Inmaculada Rangel Vacas. Context aware and adaptive mobile audio. Master's thesis, Royal Institute of Technology, School of Microelectronics and Information Technology, Stockholm, Sweden, March 2005. IMIT/LCN 2005-06, `http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/050418-Inmaculada-Rangel-Vacas.pdf`.

[31] C.T. Wildman. Math 168a - sampling and Nyquist's theorem. Technical report, Department of Mathematics of the University of California, San Diego, 2009.

[32] L.R. Yanguas and T.F. Quatieri. Implications of glottal source for speaker and dialect identification. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2:813–816, 1999. `http://doi.ieeecomputersociety.org/10.1109/ICASSP.1999.759795`.

Appendix A

Fixed class

A.1 Fixed.h

Listing A.1. Fixed.h

```

#ifndef FIXED_H
#define FIXED_H
class Fixed
{
private:
        long  m_nVal;

public:

        Fixed(void);
        Fixed(const Fixed& fixedVal);
        Fixed(const Fixed* fixedVal);
        Fixed(bool bInternal, long long nVal);
        Fixed(long nVal);
        Fixed(__int64 nVal);
        Fixed(int nVal);
        Fixed(short nVal);
        Fixed(float nVal);
        Fixed& operator=(float floatVal);
        Fixed& operator=(Fixed fixedVal);
        Fixed& operator=(int intVal);
        Fixed& operator=(long long longVal);
        Fixed& operator=(unsigned int intVal);
        Fixed operator*(Fixed b);
        int operator<(Fixed b);
        Fixed operator*(unsigned short int b);
        Fixed& Fixed::operator=(unsigned short shortVal);
        Fixed operator/(int b);
        Fixed divide(Fixed b);
        Fixed operator-(Fixed b);
        Fixed operator+(Fixed b);
        Fixed Fixed::abs();
        Fixed Fixed::log();
        BOOL Fixed::operator> (Fixed a);
        BOOL Fixed::operator> (Fixed a);

};
#endif
```

A.2 Fixed.cpp

Listing A.2. Fixed.cpp

```

#include "stdafx.h"
#include "Fixed.h"

#define RESOLUTION                      65536L
#define             RESOLUTION_BITS 16
#define FLOAT_RESOLUTIONf       0.0000005f
#define             LOGSTEPS 80


Fixed::Fixed(void)
{
        m_nVal = 0;
}

Fixed::Fixed(const Fixed& fixedVal)
{
        m_nVal = fixedVal.m_nVal;
}

Fixed::Fixed(const Fixed* fixedVal)
{
        m_nVal = fixedVal->m_nVal;
}

Fixed::Fixed(bool bInternal, long long nVal)
{
        m_nVal = nVal;
}

Fixed::Fixed(long nVal)
{
        m_nVal = (long long)nVal<<RESOLUTION_BITS;
}
Fixed::Fixed(long long int nVal)
{
        m_nVal = nVal<<RESOLUTION_BITS;
}
Fixed::Fixed(int nVal)
{
        m_nVal = (long long)nVal<<RESOLUTION_BITS;
}

Fixed::Fixed(short int nVal)
{
        m_nVal = nVal<<RESOLUTION_BITS;
}
Fixed::Fixed(float floatVal)
{
        floatVal += FLOAT_RESOLUTIONf;
        m_nVal = (long long)::floorf(floatVal*RESOLUTION);
}

Fixed& Fixed::operator=(float floatVal)
{
        floatVal += FLOAT_RESOLUTIONf;
        m_nVal = (long long int)::floorf(floatVal*RESOLUTION);
        return *this;
}

Fixed& Fixed::operator=(Fixed fixedVal)
{
        m_nVal = fixedVal.m_nVal;
        return *this;
}
Fixed& Fixed::operator=(int intVal)
{
        m_nVal = intVal<<RESOLUTION_BITS;
        return *this;
}
Fixed& Fixed::operator=(long long longVal)
{
        m_nVal = longVal<<RESOLUTION_BITS;
        return *this;
}

Fixed& Fixed::operator=(unsigned int intVal)
{
        m_nVal = intVal<<RESOLUTION_BITS;
        return *this;
}

Fixed Fixed::operator*(Fixed b)
{
        Fixed a;
        a.m_nVal =(((long long)m_nVal*b.m_nVal)>>
           RESOLUTION_BITS);
        return a;
}

Fixed Fixed::operator*(unsigned short  b)
{
        Fixed a;
        a.m_nVal = m_nVal*b;
        return a;
}

Fixed& Fixed::operator=(unsigned short shortVal)
{
        m_nVal = shortVal<<RESOLUTION_BITS;
        return *this;
}

Fixed Fixed::operator/(int b)
{
        Fixed _b = b;
        return divide(_b);
}
Fixed Fixed::divide(Fixed b)
{
        Fixed a;
        a.m_nVal = (m_nVal*RESOLUTION)/b.m_nVal);
        return a;
}

Fixed Fixed::operator- (Fixed b)
{
        Fixed a;
        a.m_nVal = m_nVal-b.m_nVal;
        return a;
}

Fixed Fixed::operator+ (Fixed b)
{
        Fixed a;
        a.m_nVal = m_nVal+b.m_nVal;
        return a;
}
Fixed Fixed::abs()
{
        Fixed a;
        if(m_nVal<0)
                a.m_nVal = -1*m_nVal;

        return a;
}


Fixed Fixed::log()
{
        // According to Taylor Series
        Fixed p_Base = Fixed(true, m_nVal);
        Fixed y = (p_Base-1).divide(p_Base+1);
        Fixed y2 = y*y;
        Fixed res = new Fixed(true,65536);
        for(int i=2*LOGSTEPS+1; i>0; i=i-2){
                res= res*y2 + 1/i;
        }
        return Fixed(2)*y*res;
}

int Fixed::operator< (Fixed b)
{
        if(m_nVal<b.m_nVal){
                return 1;
        }else{
                return 0;
        }
}

BOOL Fixed::operator> (Fixed a)
{
        if(m_nVal>a.m_nVal){
                return TRUE;
        }else{
                return FALSE;
        }
}

BOOL Fixed::operator< (Fixed a)
{
        if(m_nVal<a.m_nVal){
                return TRUE;
        }else{
                return FALSE;
        }
}
```

TRITA-ICT-EX-2010:285