Magnus: Mouse Advanced GNU Speech

A computer mouse pointer controller through voice commands

Author : Alexandre Trilla Castelló

Advisor : Dr. Francesc Alías Pujol

Servei de Tecnologies per l'Aprenentatge i el Coneixement (TAC)
Departament d'Educació
Generalitat de Catalunya

Departament de Tecnologies Mèdia (DTM)
Enginyeria i Arquitectura La Salle
Universitat Ramon Llull

2008

Per a tu, Gemma

Abstract

This Master's Thesis deals with the development of an application to control the mouse pointer and keyboard arrows of a PC through Catalan voice commands, also documenting the technical aspects involved with the digital signal processing field as well as with the IT field.

The application is programmed in Java and distributed with four flavors, from the source code distribution aimed at developers to making use of the Java Web Start technology clearly aimed at end users. One of the interesting goals of the project is to provide a degree of oral accessibility for people with reduced mobility.

Abstract

Aquest Projecte Final de Carrera tracta el desenvolupament d'una aplicació per controlar el punter del ratolí i les fletxes del teclat d'un ordinador personal a través de comandes de veu en català, així com també la documentació dels aspectes tècnics relacionats amb el camp del processat digital del senyal i del camp de les TIC.

L'aplicació és programada en Java i distribuïda de quatre maneres, des de la distribució del codi font orientada a desenvolupadors fins a l'ús de la tecnologia de Java Web Start clarament orientada als usuaris finals. Un dels punts interessants del projecte és el poder lliurar un cert grau d'accessibilitat oral a persones amb mobilitat reduïda.

Abstract

Este Proyecto Final de Carrera trata el desarrollo de una aplicación para controlar el puntero del ratón y las flechas del teclado de un ordenador personal a través de comandos de voz en catalán, así como también la documentación de los aspectos técnicos relacionados con el campo del procesado digital del señal y del campo de las TIC.

La aplicación está programada en Java y distribuida de cuatro formas, des de la distribución del código fuente orientada a desarrolladores hasta el uso de la tecnología de Java Web Start claramente orientada a usuarios finales. Uno de los puntos interesantes del proyecto es el poder proporcionar un cierto grado de accesibilidad oral a personas con mobilidad reducida.

Summary

Chapter 1 has an approach to the speech recognition field, describing some of the most usual terms used.

Chapter 2 deals with the formal mathematical methods and models that desribe the creation and functioning of a Hidden Markov Models based speech recognition system.

Chapter 3 describes Sphinx-4, a HMM-based speech recognition engine written in Java that has been taken as base system for Magnus.

Chapter 4 shows a speech enhancement method that improves the intelligibility of a speech signal by raising the power of the higher parts of the frequential spectre.

Chapter 5 describes the architecture and main components of Magnus, as well as the various means of distribution.

Chapter 6 brings the theorical concepts shown in Chapter 4 to practice, according to the design specifications of Sphinx-4.

Chapter 7 details the instrumentation tools available to score the performance of the system.

Chapter 8 reviews the rate used to measure the quality of the system, explains the preparation process of the audio files workbench and shows the results of the recognitions.

Chapter 9 summarizes the various conclusions that derive from the development of this Master's Thesis and proposes some future lines for the improvement of the application.

Acknowledgements

Although I'm the one that ran the way doing this Master's Thesis, there's a lot of people that were there to show me the right direction, with whom I'm deeply grateful.

I would like to especially give my thanks to ...

... to Francesc Busquets, my boss at the Department of Education, for the good taste that remains after being under his leadership at the IT unit for education, for the essential lessons in Java, and most important of all, for the chance for letting me produce this thesis on my scholarship at the Department.

... to Dr. Francesc Alías, my thesis advisor at the university of La Salle, for the excellent guidance, willing attitude and motivation throughout all the project. As I once said to him, if my academic career goes beyond a Master's Degree I would definitely want him to be my advisor again.

... to my parents, Josep Lluís and Maria José, for giving me the opportunity to study Telecommunications Engineering at La Salle in Barcelona, and thus allowing me to get to this Academic Degree.

... to my girlfriend, Gemma, and her father, Carlos, for the hours spent at the desk doing this thesis, for the patience, advice and faith in that what I was doing would be worth it.

... to Alfred Gaza, a former work colleague and mentor, for the present feeling of applying the technical knowledge to the development of a wealthier society.

... to all the people that have to any extent contributed to this project, from the interest that Joan de Gràcia, one of the main Linkat GNU/Linux developers and promoters, has always shown in the project to the last of the

people that have offered the beauty of their voices to the goodness of the speech recognition science. Indeed, thank you.

Alexandre Trilla
September 2008

Preface

As it can be seen presently, computers lead the latest revolution of mankind. From time to time new designs and technologies appear in order to make machines compute faster, more precise and cheaper. Personal computers inhabit almost every home in our world, we are very used to working with them, and many initiatives and projects are on the run to spread this habit to every person on Earth, whether they are aging people reluctant to the use of computers or people living in countries in development.

It can be stated, without a doubt, that computers have helped society in many ways, from easing the drag of doing repetitive work to the advances in science and security that enables us to have a pretty comfortable life. But all these brilliant progresses usually have a common feature: they are all programmed to actuate always in the same way. A microwave oven will always heat water the same way, as well as a washing machine will always do the washing-up in a similar manner. There can be excellent systems, that may contemplate a wide range of variables which respond perfectly to previous studies made by the designers and scientists, but their responses are still unknown for a supposedly different situation never seen before. What would happen if we tried to heat a calculator? An iron pair of scissors? Would it be dangerous? Could the machine avoid the danger? Possibly. And what if we tried to wash our laptop? We'd better not have a try unless we want to get rid of the computer in a very stupid way.

This is basically described as sequential programming. It's a nice and intuitive approach to the reasonable answer for the many possible situations. The microwave oven works excellent with a soup and the washing machine with the dishes and cutlery.

The more possible environments, the more lines in the program to deal with them all. But don't leave the door open to new experiments, they are all bound to fail terribly. The examples written before were just as useless and idiot as we could imagine. Nevertheless, the machines exemplified did their job very well, and so our lives easier. But when it comes to more complex goals, such as robot guidance, face identification, or in our case, (i.e. emulating human capabilities) speech recognition, we have to leave the idea of foreseeing all the possible situations so that we could provide the correct answer. We must enable the machine, let's say the computer in charge of the task, to have enough paths to select the most appropriate solution according to a set of rules that would have been learned, not programmed, previously. In other words, we'd have to give some sort of intelligence to the machine so that it would learn from our teaching. Just like the way we humans do.

For accomplishing this mission, a couple of very important changes must be done. Since we are not programmed in a sequential way (or we are not programmed at all, but we leave this question for philosophers), we will always do more or less the same task with slightly different changes, according to our mood, our haste, our attention or the need of doing something else in the meantime. So, one first important attribute that characterizes mankind is the ability of doing different chores in parallel. If one thread can hold a concrete weight, let's notice the improvement with many threads working at the same time. This aspect must be reflected on computers as well. Parallelism is one of the main issues that are being developed nowadays. Technology astonishes the world with the multi core designs of the state-of-the-art microprocessors, that double twice the power of their older brothers. It is then a very interesting point to bear in mind in order to make powerful applications.

On the other hand, every field in science must be thoroughly studied if useful results are wanted, and automatic speech recognition is no exception. So apart from the advantages that the parallel programming offer, it is the specific methods and processes of the field that must be implemented by this technique.

From now on, the Master's Thesis will expose the development of Magnus: Mouse Advanced GNU Speech, an application that will use an ASR (Auto-

matic Speech Recognition) engine to control the mouse pointer on the screen, so that it can emulate a hand. This project is intended to provide oral accessibility for people with reduced mobility. It has been programmed with Java, so that it doesn't require a concrete platform (operating system) to work, so it will do on Windows as well as on GNU/Linux. Moreover, in order to make it accessible to the largest possible community groups, it has been adapted to the Java Web Start technology, which implies no installation process, making it easier for the users to run. Java Web Start (JWS) will do all the work. If the application is run from the Internet for the first time, JWS will install it automatically, making such process transparent to the final user, and launch it afterwards. If it is then launched, either from the Internet or from the local user's computer, JWS will check that no newer release is available, or will download it automatically if so, and will launch it finally.

Magnus has been developed for the IT Projects Unit for the Educational Sector from the Department of Education of the Government of Catalonia, the Generalitat de Catalunya, with the collaboration of Enginyeria i Arquitectura La Salle (Universitat Ramon Llull), with the wish that it is most useful to society.

Motivation, objectives and hypothesis

The motivation that exists in the development of this project comes from the satisfaction of doing, as long as oneself feels able to do so, something good to help society. The author, being a Telecommunications and Electronics Engineer, wants to apply such and interesting and appealing technology, to the needs of disabled people, so that his Master's Thesis may help to ease the way to new technologies.

There is the extra motivation in developing the application for such an important institution, the Department of Education of the Government of Catalonia, thus establishing a link between the scientific environment of the author's university, La Salle, and the Department of Education, in a so widespread and important programming language, Java, and the joy of contributing to the free software community.

The main purpose of this thesis is to provide a clear and friendly, when possible, guide or manual, to create the proposed accessibility application, through the theory involved in Speech Recognition processes and the praxis development in a computerized environment.

And although it may be blurry to extract a hypothesis from such a practice project, the author has chosen to put it the following way:

"Starting from scratch, I am able to look for all the precise documentation on Automatic Speech Recognition, understand it and apply it to create a Java application to provide oral accessibility for people with reduced mobility, writing a complete manual with the needed and improved information to enable the project's continuity and making it all free software and free documentation on the Internet under a General Public compatible License and a Creative Commons licence respectively."

Contents

Preface **iv**
Motivation, objectives and hypothesis vi

Contents **viii**

List of Tables **xii**

List of Figures **xiii**

I Theory **1**

1 Approach to Speech Recognition **2**
1.1 Overview 3
1.2 Intention 3
1.3 Articulation and resonance 4
1.4 Hearing 4
1.5 Features Extraction 5
1.6 Microphone 6
1.7 Digitization 7
1.8 Spectrogram 7
1.9 Phonemes 8
1.10 Fluent speech 9
1.11 Understanding 10
1.12 Response 10
1.13 Techniques 11

2 HMM-based Speech Recognition 12
2.1 Maximum Likelihood . . . 12
2.2 Basis of Hidden Markov Models . . . 14
2.2.1 Overview . . . 15
2.3 Problems associated with HMM . . . 17
2.4 Solutions to the Three Problems . . . 18
2.4.1 Forward-Backward algorithm . . . 18
2.4.2 Viterbi algorithm . . . 18
2.4.3 Baum-Welch algorithm . . . 20
2.5 Modeling the Distributions of Sequences of Features Vectors . 20
2.6 Key Decoding Issues . . . 26
2.6.1 Active Lists and Beamwidths . . . 27
2.6.2 Language Weight . . . 27
2.6.3 Word Insertion Penalty . . . 29
2.6.4 Performance Evaluation . . . 29
2.6.5 Livemode Decoding . . . 30

3 Sphinx-4: A Java ASR engine 31
3.1 Introduction . . . 31
3.2 Framework – High Level Architecture . . . 34
3.3 FrontEnd . . . 36
3.3.1 microphone . . . 37
3.3.2 speechClassifier . . . 38
3.3.3 speechMarker . . . 39
3.3.4 nonSpeechDataFilter . . . 39
3.3.5 preemphasizer . . . 41
3.3.6 windower . . . 42
3.3.7 fft (Fast Fourier Transform) . . . 44
3.3.8 melFilterBank . . . 44
3.3.9 dct . . . 46
3.3.10 liveCMN . . . 46
3.3.11 featureExtraction . . . 47
3.4 Linguist . . . 48
3.4.1 LanguageModel . . . 49
3.4.2 Dictionary . . . 49
3.4.3 AcousticModel . . . 50
3.4.4 SearchGraph . . . 52
3.5 Decoder . . . 53

4 Speech Enhancement **56**
4.1 Introduction 56
4.2 Speech intelligibility enhancement system 58
4.3 Real-Time Implementation 61

II Practice **62**

5 Architecture, main components and software distributions **63**
5.1 Architecture 63
5.2 Main components 64
5.2.1 User interface 64
5.2.2 Sphinx-4 interaction 66
5.2.3 Sphinx-4 configuration 67
5.3 Software distributions 70
5.3.1 Source code distribution 70
5.3.2 Development distribution 72
5.3.3 Binary distribution 73
5.3.4 Java Web Start distribution 73

6 Speech Enhancement Modules **75**
6.1 Overview 75
6.2 Tunable high-pass shelving filter 76
6.2.1 Level detection 77
6.2.2 APF parameter estimation 77
6.2.3 High-pass shelving filter 81
6.3 Excess boost reducer 82
6.4 De-esser 83
6.4.1 Side-chain 83
6.4.2 Low-pass shelving filter 85
6.5 Speech enhancement results 86

7 Regression Tests **89**
7.1 Setting up a regression test 89
7.1.1 Input audio files 90
7.1.2 Batch file 90
7.1.3 Acoustic model and dictionary 91
7.1.4 Configuration file 92

7.1.5 Grammar file 93
7.1.6 Batch-mode recognizer 94
7.2 Instrumentation tools 95
7.2.1 Logger 95
7.2.2 Accuracy tracker 96
7.2.3 Speed tracker 98
7.2.4 Memory tracker 99

8 Speech Recognition Results 100
8.1 Word Error Rate 100
8.2 Audio files workbench 101
8.3 Regression tests results using the Wall Street Journal acoustic models 104
8.3.1 Acoustic insulated environment 105
8.3.2 White noise polluted environment 106
8.3.3 Blue noise polluted environment 108
8.3.4 Violet noise polluted environment 109
8.3.5 Pink noise polluted environment 111
8.3.6 Red noise polluted environment 112
8.3.7 Grey noise polluted environment 114
8.3.8 Babble noise polluted environment 115
8.3.9 Babble noise mixed with pink noise polluted environment 116
8.4 Regression tests results using the Resource Management acoustic models 119
8.4.1 Acoustic insulated environment 120
8.4.2 Pink noise polluted environment 121
8.5 Regression tests results using the WSJ acoustic models and an internal pink noise generator 122
8.5.1 Voss algorithm 122
8.5.2 Pink noise generator implementation 125
8.5.3 Impact of the internal addition of pink noise 126

9 Conclusions and Future Work 129

Bibliography 133

Thematic Index 137

List of Tables

3.1 Values for the constants that characterize the filter bank . . . 46

4.1 Frequential limits of the vocal formants 57

8.1 Audio files workbench 102
8.2 Recognition results obtained in an acoustic insulated environment 105
8.3 Recognition results obtained in a white noise polluted environment 107
8.4 Recognition results obtained in a blue noise polluted environment 109
8.5 Recognition results obtained in a violet noise polluted environment 110
8.6 Recognition results obtained in a pink noise polluted environment 112
8.7 Recognition results obtained in a red noise polluted environment 113
8.8 Recognition results obtained in a grey noise polluted environment 115
8.9 Recognition results obtained in a babble noise polluted environment 116
8.10 Recognition results obtained in a babble noise mixed with pink noise polluted environment 118
8.11 Recognition results obtained in an acoustic insulated environment 120
8.12 Recognition results obtained in a pink noise polluted environment 121
8.13 Pattern of evaluation of the random number generators 123
8.14 Results obtained with the internal pink noise generator (PNG) 126

List of Figures

1.1 MFCC features extraction process 6

2.1 Trellis diagram of the Viterbi algorithm 19
2.2 HMM example 21
2.3 HMM concatenation example 24

3.1 Sphinx-4 framework 34
3.2 Sphinx-4 FrontEnd 36
3.3 A data stream with only one speech region 40
3.4 A data stream with only one speech region after filtering . . . 40
3.5 A data stream with two speech regions 40
3.6 A data stream with two speech regions after filtering, when mergeSpeechSegments is set to true 41
3.7 A data stream with two speech regions after filtering, when mergeSpeechSegments is set to false 41
3.8 Relationship between original data, window size, window shift, and the windows returned 43
3.9 The Hamming window function with its corresponding spectral diagram 43
3.10 A Mel-filter bank 45
3.11 Layout of the returned features 47
3.12 Delta and double delta vector computation 48
3.13 Example SearchGraph 52

4.1 The proposed speech intelligibility enhancement system 59

5.1 Magnus architecture and main components. 64

6.1 Tunable high-pass shelving filter diagram. 77

6.2 Step response of the desired controller. 79
6.3 Map between the cut-off frequency and the α parameter. . . . 81
6.4 De-esser structure. 83
6.5 Hard-knee compressor function. 84
6.6 Spectral plot of an enhanced speech utterance with a sibilant. 87
6.7 Temporal plot of a sibilant speech utterance. The most sibilant parts of the chunk have been highlighted. 88

8.1 Frequential analysis of white noise 107
8.2 Frequential analysis of blue noise 109
8.3 Frequential analysis of violet noise 110
8.4 Frequential analysis of pink noise 111
8.5 Frequential analysis of red noise 113
8.6 Frequential analysis of grey noise 114
8.7 Frequential analysis of babble noise 116
8.8 Frequential analysis of babble noise mixed with pink noise . . 117
8.9 Frequential analysis of the pink noise generated by the Voss algorithm 125
8.10 Results deviations with respect to the abscence of the pink noise generator 127

Part I

Theory

Chapter 1

Approach to Speech Recognition

In this chapter, a very basic and intuitive view of speech recognition systems is given, along with a set of technical vocabulary widely spead among speech developers. Its intention is that the reader gets used to the descriptive speech terminology.

A mathematical perspective of speech recognition is also given in order to get used with the rigor by which this science can be very well described.

In this chapter, the basic analysis of an Automatic Speech Recognition (ASR) system is described. From a very general point of view, the several elements that build an ASR engine are discussed, making references continuously to the engine used in this project, named Sphinx-4, thoroughly analyzed in the following chapters.

If more information is needed, please refer to [Colton, 2003], which is a very basic but complete enough tutorial to get the gist of ASR systems terminology, with some specific parts that are left out or summarized in this paper. For deeper mathematical description of ASR processes, the reader is invited to read [Rabiner, 1989] and [Rabiner and Juang, 1991].

1.1 Overview

Automatic Speech Recognition is a computerized process that receives as input a speech signal, whether it may come from a recording previously taken or a live signal from a microphone, and through deep digital signal processing, it produces as its output a transcription of the spoken speech.

Despite this process is yet an unsolved problem, nobody in the world has yet demonstrated the ability to do it as well as humans. ASR stands an attractive study subject for many researchers and developers who want to include such an outstanding technology in their applications. This fact implies that although many ways have been, and are presently being, studied and exploited, such systems dont have a 100% of reliability, so it must be taken into consideration that any program that includes speech, voice or sound recognition is subject to error. The goal of scientists is to provide a method that aims at the ideal 100%, but it is a difficult task that still needs some years of deep investigation.

Up to date, some learning algorithms are used to emulate human intelligence, which is so far the one that best accomplishes the task of speech recognition as stated before, and any artificial intelligence proposal is welcome to have a try on the game.

1.2 Intention

Speech starts with the intention to communicate. The intended sounds produced by a speaker are meant to carry a meaning. But there are many sounds on the other hand that do not mean anything in particular themselves: a sigh, a sneeze, a hum... though in a human conversation may have a lot of significance. It is the first ones which are of interest to ASR, obviously.

Sometimes, the production of a similar sound could confuse an accurate analysis, but human listeners have developed a means of understanding the gist of the vocal utterance by the use of common sense to resolve the ambiguity. This is also considered communication.

In the machine world, no intelligence is available, so it makes no sense to expect a computer to understand a confusing utterance with common sense, despite of the redundancy. But some models, called grammars, are of use to provide a solution to this problem. There is more to be written about this subject, but this is left to be dealt in the following chapters.

1.3 Articulation and resonance

Articulation is the act or process of dividing a unit into separate articles. In the case that concerns speech recognition, the unit is a sound and the articles are phonemes. We humans chain different phonemes to articulate speech utterances by the action of the mouth, nose and throat.

There are many aspects involved in articulation. As the air flows through the vocal cords, a tone or pitch is created. When this tone is present, the speech is said to be *voiced*. When it is absent, the speech is *whispered*. All vowels are voiced, but this premise doesn't apply to all consonants. There are some that are voiced, some that are not and some that are both voiced and unvoiced. This is a very important aspect to take into consideration because whispered consonants, yet humans can generally understand them, the ASR techniques cannot detect as many differences as voiced sounds. So the focus is on the voiced speech when it comes to recognition. Again, there is more to be said about this topic, because for example one of the goals to improve the speech intelligibility (and so the accuracy of the speech recognition system) is by emphasizing the presence of consonant sounds. More about this in the Speech Enhancement section.

Also, when articulating, the positioning of tongue, jaw and lips creates resonant chambers of various sizes within the head that contribute additionally to the speech tonality.

1.4 Hearing

Another important consideration is to be made when referring to communication: the hearing, because the computer running an ASR software must emulate human hearing in some way (there's an insistence on emulating a

human behavior on a machine, but again, we humans have developed, during our evolution, a unique and singular way of communicating that has not yet been imitated). This is achieved by the transformation of spoken sound signals into specific coefficients that represent the way humans perceive speech.

And yet, this last point concludes into a discrete and compressed representation of the signals that will make it possible to compute them for the following processes. Because humans perceive acoustic signals between 30Hz and 20KHz with a varying frequency response, this leads to the named Mel Scale, after the Melodic Scale. When filtering and parameterizing the incoming sound signals through the Mel frequency response, some coefficients can be obtained in order to perform the recognition in a similar way we humans do. This matter is exposed in the following chapters.

1.5 Features Extraction

Speech recognition systems do not interpret the speech signal directly, but rather a set of features vectors derived from the speech signal. In this thesis, as in most current speech recognition systems, the speech signal is parameterized into a sequence of vectors called Mel-Frequency Cepstral Coefficients (MFCC) [Davis and Mermelstein, 1980], or simply *cepstral coefficients*.

Cepstral coefficients attempt to approximate the spectral processing of the auditory system in a computationally efficient manner. They emulate the human auditory system through the use of the *mel scale*, which maps the frequency response of the auditory system. The incoming speech signal is divided into a sequence of short overlapping segments, called *frames*. Each frame is processed as follows. The frame is windowed and then transformed to the frequency domain using a Short-Time Fourier Transform (STFT) . More about this transform is available in [Nawab and Quatieri, 1987]. The squared magnitude of the STFT is computed and then multiplied by a series of overlapping triangular weighting functions called *mel filters*. These triangular filters are equally distributed along the mel frequency scale with a 50% overlap between consecutive triangles. These filters are spaced in frequency approximately linearly at low frequencies and logarithmically at higher frequencies. The *mel spectrum* of the frame is computed as a vector whose components represent the energy in each of the mel filters. To approximate

human auditory processing more closely, the natural logarithm of each of the elements in the mel spectral vector is then computed, producing the *log mel spectrum* of the frame. Finally, this vector is converted to mel-frequency cepstra via a Discrete-Cosine Transform (DCT) and then a truncation. The features extraction process is shown in Figure 1.1.

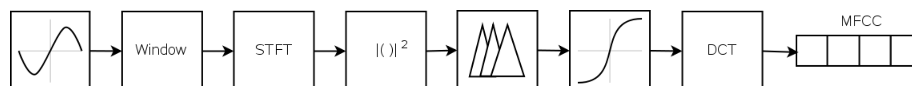

Figure 1.1: MFCC features extraction process

The input to a speech recognition system is typically a sequence of vectors composed of the mel-frequency cepstral coefficients as well as their first and second temporal derivatives, approximated by using first and second differences of neighboring frames, respectively.

The type of features vectors used for speech recognition purposes is not restricted to MFCCs. Any type of features of any dimensionality could be used successfully instead, but MFCCs are currently known to be the best single features parametrization for good speech recognition performance in HMM-based systems under most acoustic conditions.

1.6 Microphone

In a speech recognition system, a microphone substitutes the ears. The pressure waves that compose sounds are transduced into an analog electrical signal by many different means depending on the quality of the transducer, which name is a microphone. It is very important that a minimum quality of the device is used, because this can make the difference between a good functioning system and a bad functioning one. The quality is rather critical, unfortunately, because the many tests run show empirically that although the difference is insignificant to a human listener, it becomes abysmal to a recorded signal, and hence, an ASR system.

There are also some methods to attenuate the unwanted noise that a bad microphone gives, or the unwanted noise that a noisy environment offers.

Active filtering stands for one of the most attractive solutions, implemented as a noise cancellation method, using a second microphone closer to the noise source and a well tuned algorithm such as Kalman Filtering the results can be spectacular. Refer to [Kybic, 1998] for a deeper analysis on Kalman Filtering applied to speech enhancement, which may derive into a better speech recognition system. Other more modest methods such as spectral subtraction could be used as well if another microphone was available. Anyway, the implementation of these filtering techniques should not worry the reader at this point since the matter that concerns now is speech recognition.

1.7 Digitization

Since the process that the ASR software will end up doing is a digital signal process, one crucial point is the conversion form the analog world into the discrete domain. This is achieved using the Analog Digital Converters (ADC) by sampling the signal as many times per second (*i.e. sampling rate*) as necessary to accomplish the Nyquist's Theorem, which basically states that to correctly resolve a signal at some frequency, it has to be sampled at more than twice the maximum frequency that composes the signal so as to avoid aliasing. In the project, it can be demonstrated that within 8KHz there is enough information in oral speech, at least, enough to clearly distinguish the voiced phonemes. So, the sample rate used by the Sphinx-4 engine is 16KHz.

Another feature involved in digitization is the amplitude quantization process which introduces a quantization error. This error is related to the number of bits that are required to represent the real signal. Of course, the more bits the better, but the larger amount of information. A compromise must be reached in order to treat the minimum volume of information and manage an adequate sound signal quality. When using Sphinx-4, this magnitude is set to 16 bits, the same as the standard quality of a CD recording.

1.8 Spectrogram

Spectrogram is the name given to a voice print. In more common language it could be defined as the analysis of the many features that compose a voice

signal. These properties can be extracted, basically, by one of the many distinct expressions of the Fourier Transform, depending on the domain of use. This transform allows the frequency representation of the signal, so, the many sine tones that compose it.

The signal in question must be divided into frames to extract bundles of data almost stationary in order to obtain relevant features. These frames must then be windowed, that is, multiplied temporally, by a known smooth function (window) that eliminates the discontinuities in the edges of the signal and so avoids false representations.

And one last aspect worth mentioning is the *formants*, which are the frequency strong bands that move across time in a frame transformation. Each band represents a resonance in the speech production system of the person talking. It's interesting to focus on the transitions occurring because of all the phoneme pronunciations and see how they are registered by the movement of the formant bands. Each sound is represented by its corresponding formant fingerprint.

1.9 Phonemes

This linguistics related concept is any of a small set of units considered to be the basic distinctive units of speech sounds by which morphemes, words, and sentences are built. So they are the tiniest units of speech that distinguish meaning.

Phonemes may vary slightly from person to person, though even when they are articulating the same one. This is because the unique vocal register of a person. Although on a single person it may vary slightly from time to time, it must be taken for the same any time. Tolerance is the amount of variation that is allowed before something becomes unusable, or unrecognizable. For purposes of speech recognition, the same word pronounced by different speakers should still be recognized as the same word. This stands a big challenge when one of the subjects in question is a foreigner.

Another related concept which deals with "phoneme similarity" is the allophone. An allophone is one of several similar phones that belongs to the

same phoneme. A *phone* is a sound that has a definite shape as a sound wave, while a phoneme is a basic group of sounds that can distinguish words. Thus an allophone is a phone considered as a member of one phoneme.

1.10 Fluent speech

Fluent speech is one serious aspect in speech recognition, because one word blends into the next, and so on. Computers cannot easily divide the phonemes into words.

Because of fluency and coarticulation effects, the word boundaries are difficult to detect, and so, when having to split an utterance into several words, or a word into its constituent phonemes, it becomes a very difficult task to achieve. It is then when the effects of i*mondegreens* occur, which mean that two different sentences, for example, with a different written expression, are phonetically the same, and in consequence, sound the same. In more technical terms, a mondegreen is the mishearing (usually accidental) of a phrase as a homophone or near-homophone in such a way that it acquires a new meaning. A homophone is a word that is pronounced the same as another word but differs in meaning.

Another interesting aspect of speech misunderstanding is the spontaneous speech. In spontaneous speech, speakers will often utter more than one contribution during their turn of speaking. Unlike written text, there are no explicit punctuation marks that delimit one utterance from the next. Furthermore, due to the online nature of spontaneous speech, speakers sometimes need to revise what they have just said, by making a speech repair, which will help resolving their intended contribution.

One interesting and useful way to face all these problems is by the use of *grammars*. If a computer expects a determined grammatical structure, it may ease the task of finding the correct word boundaries and so splitting them successfully.

Another interesting approach to this flaw is the addition of more variables to the system, such as the GMT, the weather, etc. in order to determine

the right transcription in a given environment. For example, if someone is asked if he or she is hungry yet, it would make sense if it happened at midday, or if he or she was going to take the umbrella, if the weather forecast announced that rain was coming soon. It would be easier to understand the exemplified questions if some information was known beforehand. This would be like giving some sort of common sense to computers, which is tough, see [Lieberman, 2002] for an intuitive lecture on the matter, but any good reasoned proposal is welcome.

1.11 Understanding

Understanding natural language is part of the more renowned term called Natural language Processing (NLP), which is a subfield of artificial intelligence and linguistics. It studies the problems of automated generation and understanding of natural human languages.

Natural language generation systems convert information from computer databases into normal-sounding human language, and natural language understanding systems convert samples of human language into more formal representations that are easier for computer programs to manipulate.

In theory, natural language processing is a very attractive method of human-computer interaction and is sometimes referred to as an artificial intelligence complete problem, because natural language recognition seems to require extensive knowledge about the outside world and the ability to manipulate it. The definition of "understanding" is one of the major problems in natural language processing to evaluate.

The goal of NLP evaluation is to measure one or more qualities of an algorithm or a system, in order to determine if (or to what extent) the system answers the goals of its designers, or the needs of its users.

1.12 Response

This matter is usually taken for granted in a normal human conversation, because the slightest of the sighs or murmurs would be taken as a sign of

response. It is specially important for speech recognition application developers not to leave the computer in standby, despite internally it could be computing the strangest of the mathematical operations.

To avoid communication breakdown, there should be always a kind of response from the computer, to avoid giving the feeling of hanging up or malfunctioning, such as a sudden frozen screen.

1.13 Techniques

In order to deal with such a maddening matter, ASR systems take much advantage from the artificial intelligence methods to provide a solution to this problem. There's one historically interesting approach called the Dynamic Time Warping (DTW), which is an algorithm for measuring similarity between two sequences which may vary in time or speed. Despite of its interest, it has been displaced by other more "intelligent" techniques, such as the Artificial Neural Networks (ANN) and the Hidden Markov Models (HMM).

Neural Networks are designed as a means of somehow implementing the paradigm of human intelligence, they emulate neurons, which are the basic unit of our brain and neuron system, and let them learn from prepared environments. The virtual neurons interpolate the different tests, making an unclear internal mathematical representation, that allows them to extrapolate a new solution to a new environment never seen before. But again, despite of its interest, this method lacks specialization, and again it has been displaced by the Hidden Markov Models, which basically seek the same goals, but maximizing the interests of speech recognition given a set of premises. More about HMMs is on the come in the following chapters.

Now, given a phoneme string and a set of processed outputs from any of the intelligent solutions, it is possible to measure how well they match. The search results in an optimal alignment between the utterance and the target phonemes in oder to decode the input message.

Chapter 2

HMM-based Speech Recognition

This chapter delves into the world of Hidden Markov Models, the base for modern speech recognition systems. A first overview of their use in the speech recognition field is given, just to avoid losing sight of the goals that should be accomplished. Then the Maximum Likelihood method is presented, which is the common point shared by the several estimation theories used in pattern recognition. Afterwards the internal constitution of the HMMs is described, with the problems associated to them and the algorithms that provide solutions to these problems and finally the modeling of the real world with an uncertainty factor, which gives place to the HMM-based representation of the sounds that form the classes whose parameter values are to be estimated.

As described in the previous chapter, HMMs provide an idyllic cushion for developing speech recognition applications. [Seltzer, 2003] gives a very good and accurate description of HMM-based ASR. His thesis has been of much help for this chapter.

2.1 Maximum Likelihood

Speech recognition systems are statistical pattern classification systems. In these systems, sounds or sequences of sounds, such as phonemes or words are modeled by distinct classes. The goal of the speech recognition system is to estimate the correct sequence of classes, *i.e.* sounds, that make up the

incoming utterance, and hence, the words that were spoken.

Maximum Likelihood (ML) estimation is a "best-fit" statistical method for the estimation of the values of the parameters of a system, based on a set of observations of a random variable that is related to the parameters being estimated. Note that the parameters are not themselves random variables. Rather, they are assumed to be unknown constants.

So in state-of-the-art recognition systems, speech recognition is not performed directly on the speech signal. The speech waveform is divided into short segments or frames and a vector of features is extracted from the samples of each frame. For convenience, since the speech recognition engine used in this thesis is phoneme-based, the frames will be parts of the phonemes that build the speech utterances, and so, the values of the parameters of such phonemes, say the MFCCs of a phoneme, will be estimated in order to perform the recognition tasks.

If Z represents a sequence of features vectors extracted from a speech waveform, speech recognition systems operate according to the optimal classification, Equation (2.1), which is based on the ML method.

$$\hat{w} = \underset{w \in W}{argmax}\, P(w|Z) \tag{2.1}$$

In Equation (2.1), $\hat{w}$ is the sequence of parameters (phonemes, or words, or any other estimable variable) hypothesized by the recognition system and W is the set of all possible sequences that can be hypothesized by the recognition system. The idea is that the parameters of the phonemes may be estimated, to then create a set of possible phonemes, which could be grouped into words, to then create a set of words which in their turn could be grouped into... The story repeats recursively.

However, Equation (2.1) is not actually computed. Instead, Bayes rule is used to rewrite it as expressed in Equation (2.2).

$$\hat{w} = \underset{w \in W}{argmax}\, \frac{P(Z|w)P(w)}{P(Z)} \tag{2.2}$$

$P(Z|w)$ is the *acoustic likelihood* or *acoustic score*, representing the probability that a features sequence Z is observed given that phoneme sequence w was spoken, and $P(w)$ is the *language score*, the *a priori* probability of a particular phoneme sequence w. This latter term is computed using a *language model*. Because Equation (2.2) is to be maximized with respect to the phoneme sequence w for a given (and therefore fixed) sequence of observations Z, the denominator term $P(Z)$ can be ignored in the maximization, resulting in Equation (2.3).

$$\hat{w} = \underset{w \in W}{argmax}\, P(Z|w)P(w) \tag{2.3}$$

Note that the ML estimate (on which the HMM-based speech recognition is built) is dependent with the observed samples. This can lead to errors in estimation since the observed samples may not be a fair representation of the distribution of the random variable. This usually occurs when the number of observed samples is small. So bearing this in mind future problems should be avoided. For example, if the parameter in question to be estimated was the mean, then, it is obvious to think that two people could not represent the majority of a group of a million individuals, but a thousand people could possibly be a more fair representation of the group, or at least yield a more accurate result than the previous sample.

This degree of accuracy is given by the models, the HMMs, that build the base of a speech recognition system. Each HMM is associated with a sound unit and learns its parameters, say acoustic features. This process is called *training*. Then the resulting HMMs are used to deduce the most probable sequence of features, related to the sound units, according to the ML estimate, and this process is called *decoding*.

2.2 Basis of Hidden Markov Models

The tutorial provided by [Dugad and Desai, 1996] has been taken for reference because of its conciseness and clarity. However, the article approximates to the HMMs taking a finite, or discrete, number of distinct observation symbols, say features vectors for the case that concerns this thesis.

These features vectors are actually MFCCs, velocities of MFCCs and accelerations of MFCCs, which correspond to the first and second derivatives of the MFCCs respectively. Since they are defined in the continuous domain an infinite number of different symbols would be necessary to warp the entire continuous domain. Thus, instead of dealing with probability functions (discrete domain), probability density functions (continuous domain) are of use to define the output probabilities of the features vectors.

2.2.1 Overview

A Hidden Markov Model is a statistical model in which the system being modeled is assumed to be a Markov process (and thus a stochastic process) with unknown parameters, and the challenge is to determine the hidden parameters from the *observation sequence*. This sequence is given by the outcomes produced by the HMM in question. The extracted model parameters can then be used to perform further analysis, for example, for pattern recognition applications. Eventually, as stated in the previous chapter, speech recognition systems are pattern classification systems.

In a regular Markov model, the states are directly visible to the observer, and therefore the state transition probabilities are given parameters. For any instant of time, the process should be in any of its possible states, then the transition probabilities are only determined by this state in question, independently from the past states sequence already transited.

In a *Hidden* Markov Model, the states are not directly visible, but variables influenced by the states are visible. Each hidden state has a probability distribution over the possible output tokens. Therefore the sequence of tokens (observation sequence) generated by a HMM gives some information about the sequence of states.

In order to describe the content and behavior of a HMM accurately, the following notation is defined:

$N \Rightarrow$ number of states in the model. The amount of states in a HMM is related to the time-varying characteristics of the sound units. Sounds which are highly time-varying need more states to represent them.

$T \Rightarrow$ length of the observation sequence, *i.e.* the number of observed features vectors.

$s_t \Rightarrow$ denotes the present state s at time t.

$\pi =$ $\{\pi_i\}$ where $\pi_i = P(s_1 = i)$ is the probability of being in state i at the beginning of the experiment, *i.e.* at $t = 1$.

$A =$ $\{a(i,j)\}$ where $a(i,j) = P(s_{t+1} = j \mid s_t = i)$ is the probability of being in state j at time $t+1$ given that previously, at time t, the state was i. It is assumed that the several $a(i,j)$ are independent of time because of the memoryless system modeled by the Markov process, or in other words, they don't depend on past transitions. So the probability of going from one state to another is given only by the present state at any time. A represents the transition matrix.

$B =$ $\{b(z,i)\}$ where $b(z,i) = P(z \mid s_t = i)$ is the probability of observing the features vector z given that the present state is i. B represents the confusion matrix.

$z_t \Rightarrow$ denotes the features vector observed at instant t.

$Z =$ $\{z_1, z_2, ..., z_T\}$ denote the sequence of features vectors observed, *a.k.a.* the observation sequence.

$\lambda =$ (A, B, π) is used as a compact notation of a HMM.

Using this model, an observed sequence of features vectors $Z = \{z_1, z_2, ..., z_T\}$ is generated as follows: the experiment starts by choosing one of the states (according to the initial probability distribution π), then a features vector is determined by the probability distribution $b(z,i)$. This beginning instant of time is taken as $t = 1$ and the state and features vector determined at this moment are denoted by s_1 and z_1 respectively. After this, the following state s_{t+1} is determined (it may be same or different from s_t) according to the transition probability distribution given by the transition matrix A and again a new features vector (denoted by z_2) is determined from this new state depending on the probability distribution for that state. Continuing this up to time $t = T$, the sequence of features vectors $Z = \{z_1, z_2, ..., z_T\}$ is generated.

Another point of view could be established from the Bayesian networks' perspective since Bayesian networks are directed acyclic graphs (DAG) that represent dependencies between variables in a probabilistic model, as stated in [Ghahramani, 1998]. Then the time series models including the HMMs used in speech recognition could be viewed as examples of dynamic Bayesian networks. This is an interesting approach since there is high dependence of the output sequence on the variables that characterize the HMM, say the state individual bias, the transition probabilities between the several states and the state which is chosen to begin the observations. There is more to be found about Bayesian networks in the article referenced in this paragraph. It is not the intention of this thesis to provide a thorough study of this alternative, it is rather left for the reader to delve into it as a different approach to the matter.

2.3 Problems associated with HMM

In the end, most applications based on HMMs are finally reduced to solving the three main problems cited in the following description:

Problem 1 Given the model $\lambda = (A, B, \pi)$, the computation of $P(Z|\lambda)$, which is the probability of occurrence of the observation sequence $Z = \{z_1, z_2, ..., z_T\}$.

Problem 2 Given the model $\lambda = (A, B, \pi)$, the choice of a states sequence $I = i_1, i_2, ..., i_T$ so that $P(Z, I|\lambda)$, the joint probability of the observation sequence Z and the states sequence given the model, is maximized.

Problem 3 The adjustment or tuning of the HMM parameters $\lambda = (A, B, \pi)$ so that $P(Z|\lambda)$ (or $P(Z, I|\lambda)$) is maximized.

Problems 1 and 2 are considered analysis problems while Problem 3 is considered a synthesis (or model identification or training) problem. The algorithms described in the following sections are used to provide solutions to these problems. They are based on the Maximum Likelihood method introduced in the previous section.

2.4 Solutions to the Three Problems

This section provides a description of the main algorithms used to solve the Three Problems. Although there are other different approximations or perspectives of the algorithms in the bibliography associated with HMMs, the general idea is shared by them all. They all make use of dynamic programming techniques, which are methods for optimally solving the big problems through the overlapping of simpler subproblems and defining optimal structures. The use of these dynamic techniques take much less time than naive methods.

2.4.1 Forward-Backward algorithm

This algorithm provides a solution to Problem 1. Given a model, the algorithm computes inductively the probability of the partial observation sequence up to a time and the probability of being at a concrete state at that time.

A brute force procedure for the solution of this problem would imply the generation of all possible sequences of observed features vectors given a concrete states sequence, for all possible states sequences. Since the number of possible states sequences is geneally huge, the solution becomes intactable for realistic problems. It is then when the Forward-Backward algorithm plays an important role in this because it reaches the solution within a time complexity order of TN^2 compared to the time complexity yielded by the brute force procedure, which is TN^T.

2.4.2 Viterbi algorithm

This algorithm provides a solution to Probem 2. It is an inductive procedure in which at each instant of time it keeps the best (*i.e.* the one giving maximum probability) possible states sequence for each of the N states as the intermediate state for the desired features vector sequence Z. It finally yields the best path for each of the N states as the last state for the desired observation sequence. Out of these, the one which has the highest probability is selected. So, the gist of the Viterbi algorithm is that it estimates the optimum states sequence, and it does so by reformulating the problem accurately. It is a maximization of the Forward-Backward algorithm.

Typically the Viterbi algorithm is visually represented by a trellis diagram, where the aim is to find the best path through a matrix where the vertical dimension represents the states of the HMMs and the horizontal dimension represents the frames of speech (*i.e.* time). Figure 2.1, extracted from [Young *et al.*, 2006], represents the algorithm as described.

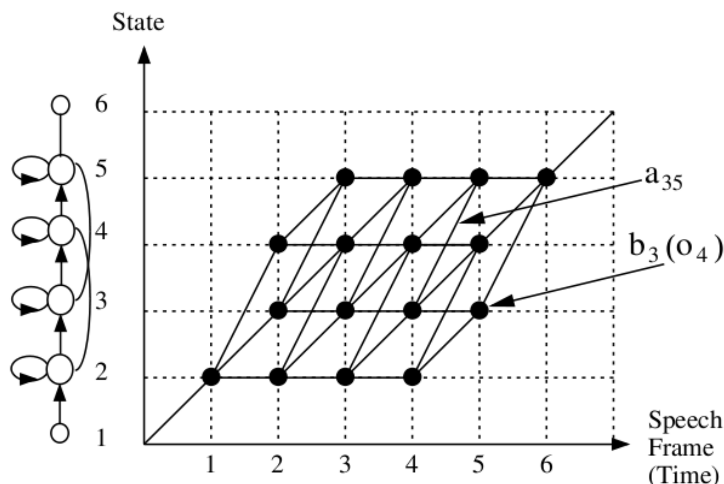

Figure 2.1: Trellis diagram of the Viterbi algorithm

Each large dot in Figure 2.1 represents the log probability of observing that frame at that time and each arc between dots corresponds to a log transition probability. The log probability of any path is computed by summing the log transition probabilities and the log output probabilities along that path. The paths are grown from left to right column by column. At time t, each partial path is known for all states enabling the computation of this partial likelihood, which operated recursivelly becomes the Viterbi algorithm.

The reason for using logs instead of the actual values comes from the fact that the direct computation of the real values ends up being very small, leading to underflow, because of the several products involved in the re-estimation process. Taking the logarithm of such small values enables the system to have more tractable results.

2.4.3 Baum-Welch algorithm

This algorithm provides a solution to Problem 3, which is a maximization of the solution to Problem 1 through the adjustement of the parameters of the model λ. This optimization criterion is called the *maximum likelihood criterion*.

To determine the parameters of a HMM it is first necessary to make a rough guess at what they might be. Once this is done, more accurate (in the maximum likelihood sense) parameters can be found by applying the so called Baum-Welch re-estimation formulæ.

Since the full likelihood of each observation sequence is based on the summation of all possible states sequences, each observation vector (features vector) contributes to the computation of the maximum likelihood parameter values for each state. In other words, each observation is assigned to every state in proportion to the probability of the model being in that state when the vector is observed. Such probability is efficiently calculated using the Forward-Backward algorithm.

2.5 Modeling the Distributions of Sequences of Features Vectors

In frame-based statistical speech recognition systems, the speech production mechanism is characterized as a random process which generates a sequence of features vectors. In Hidden Markov Model speech recognition systems, the random process which corresponds to a particular phoneme is modeled as a HMM, which can be characterized by the following:

- a finite number of states.
- a state-transition probability distribution which describes the probability associated with moving to another state (or possibly back to the same state) at the next time instant, given the current state.
- an output probability distribution function associated with each state.

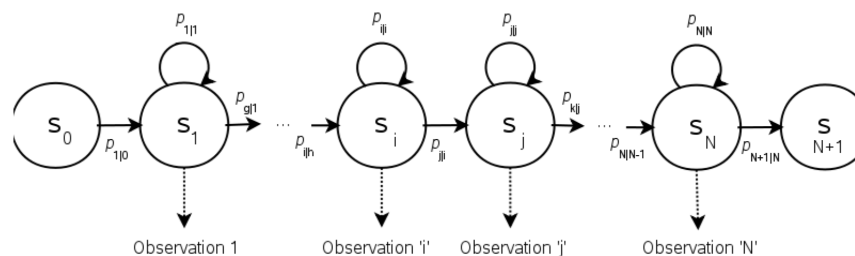

Figure 2.2: HMM example

An example of a HMM is shown in Figure 2.2. HMMs will be generally stated to have N states. The solid arrows represent the allowable transitions from each state, which in this example, and due to the sequential nature of speech, are restricted back to the current state or to the state immediately to the right, all other state transitions have probability zero, but could actually point to any state of the diagram in general if the system in question being modeled didn't have the sequential property of a speech utterance.

Due to this sequential property and the need of achieving real-time speeds, contemplating the variable speed of the different models, the HMM topology can be parameterized to be a strict left-to-right Bakis topology, which enables the HMMs to skip states.

The probability of going from state i to state j is labeled on each arrow as $p_{j|i}$. The dotted arrows point to the observations (features vectors) generated by the respective states, which have probability density distributions associated with each state. Note that the initial and final states are non-emitting. No observations are associated with these states. The final state is also an *absorbing* state since when this state is reached, no further transitions are permitted.

The statistical behavior of a HMM representing a given phoneme is governed by its state transition probabilities and the output distributions of its constituent states. For a HMM modeling phoneme w, the transition probabilities are represented by a *transition matrix*, A_w. The elements of this matrix, $a_w(i,j)$ represent the probability of transiting to state j at time $t+1$ given that state i is occupied at time t. Thus, if the HMM for phoneme w has

N states, Equation (2.4), which represents $p_{j|i}$ in Figure 2.2, can be easily deducted.

$$\sum_{j=1}^{N} a_w(i,j) = 1 \tag{2.4}$$

The state output probability distribution functions and the state transition probability distributions are usually modeled as Gaussians or mixtures of Gaussians. The number of HMM parameters to be estimated increases as the number of Gaussians in the mixtures increase resulting in less data being available to estimate the parameters of every Gaussian distribution. However, such an increase produces finer models which lead to a better recognition performance.

Typically, in order to improve computational efficiency, the Gaussians are assumed to have diagonal covariance matrices, *i.e.* covariance matrices where the off-diagonal elements are all 0. Thus, the output probability of a features vector z belonging to the state i of a HMM for phoneme w is represented in Equation (2.5), where α_{ik}^{w}, μ_{ik}^{w} and Σ_{ik}^{w} are the mixture weight, mean vector and covariance matrix associated with the kth Gaussian in the mixture density of state i of the HMM of phoneme w.

$$b_w(z,i) = \sum_{k} \alpha_{ik}^{w} \Omega(z; \mu_{ik}^{w}, \Sigma_{ik}^{w}) \tag{2.5}$$

It is defined B_w as the set of parameters $\{\alpha_{ik}^{w}, \mu_{ik}^{w}, \Sigma_{ik}^{w}\}$ for all mixture components for all states in the HMM for phoneme w. B_w is sometimes referred to as the *confusion matrix*. Finally, $\lambda_w = (A_w, B_w)$ is defined as the complete set of statistical parameters that define the HMM for phoneme w. Typically, along with A_w and B_w, a third parameter identified by π_w is given, which represents the probability of being at a concrete state at the beginning of the process. But due to the sequenciality of speech, it is assumed that the first state, or the state of departue, is always the leftmost state.

To generate a sequence of N features vectors for a phoneme modeled by a HMM, the generator is assumed to transit at least through a sequence of

$N+2$ states in the HMM presented in this section, beginning with the non-emitting initial state and terminating in the non-emitting final state, or as already named, the absorbing state. At each time instant, a features vector is drawn from the probability distribution of the state currently occupied. The sequence of vectors so generated is said to be *generated by the HMM.*

For the computation of the probability that a given sequence of features vectors $Z = \{z_1, z_2, ..., z_T\}$, generated by the HMM for phoneme w, referred to as HMM$_w$, let S denote the set of all possible state sequences of length N through HMM$_w$. The total probability that HMM$_w$ generated Z can be expressed as in Equation (2.6), where $s = \{s_1, s_2, ..., s_T\}$ represents a particular states sequence through HMM$_w$.

$$P(Z|w) = \sum_{s \in S} P(Z, s|w) = \sum_{s \in S} P(Z|s)P(s|w) \tag{2.6}$$

The expression $P(s|w)$ represents the probability of a particular states sequence and is computed from the state transition matrix A_w. The expression $P(Z|s)$ represents the probability of a particular sequence of features vectors given a states sequence, and is computed from the state output probability distributions using Equation (2.5). Thus, Equation (2.6) can be rewritten as Equation (2.7).

$$P(Z|w) = \sum_{s \in S} \left(\prod_{t=1}^{T} a_w(s_t, s_{t+1}) \right) \left(\prod_{t=1}^{T} b_w(z_t, s_t) \right) \tag{2.7}$$

The substitution of Equation (2.7) into Equation (2.3) leads to the expression used to perform speech recognition, Equation (2.8).

$$\hat{w} = \underset{w}{argmax} \left\{ P(w) \sum_{s \in S} \left(\prod_{t=1}^{T} a_w(s_t, s_{t+1}) \right) \left(\prod_{t=1}^{T} b_w(z_t, s_t) \right) \right\} \tag{2.8}$$

However, for computational effciency, most HMM speech recognition systems estimate the best states sequence, *i.e.* the states sequence with the highest likelihood, associated with the estimated hypothesis. Thus, recognition is actually performed as in Equation (2.9).

$$\hat{w} = \underset{w,s \in S}{argmax} \left\{ P(w) \left(\prod_{t=1}^{T} a_w(s_t, s_{t+1}) \right) \left(\prod_{t=1}^{T} b_w(z_t, s_t) \right) \right\} \tag{2.9}$$

While this discussion has only involved the recognition of single phonemes, thus being *Isolated Unit Recognition*, as stated before the HMM framework can easily be expanded to model strings of phonemes, $w = [w_1, w_2, ..., w_T]$, say words, and in fact this is the idea to exploit the full power of speech recognition systems. If individual phonemes are modeled by unique HMMs in the manner described, then HMMs corresponding to sequences of phonemes can easily be built by concatenating the HMMs of the constituent phonemes. An example of this is shown in Figure 2.3 for an utterance composed of two phonemes.

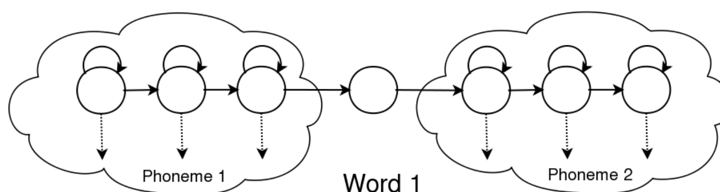

Figure 2.3: HMM concatenation example

Each model in the sequence corresponds directly to the assumed underlying symbol, which could be either whole words for so-called *connected speech recognition* or sub-words such as phonemes (as is the case in this thesis) for *continuous speech recognition*. The reason for including the non-emitting entry and exit (absorbing) states should now be evident. These states provide the linking mechanism needed to join models together.

There are, however, some practical difficulties to overcome. The training data for continuous speech must consist of continuous utterances and, in general, the boundaries dividing the segments of speech corresponding to each underlying sub-word model in the sequence are unknown. In practice, it is usually feasible to mark the boundaries of a small amount of data by hand.

All segments corresponding to a given model can then be extracted and an *isolated training* style as described in the Baum-Welch subsection above could be used. However, the amount of data obtainable in this way is usually very limited and the resultant models obtained are of poor quality. In order to improve such situation *embedded training* is used. This method uses the same Baum-Welch procedure but rather than training each model individually all models are trained in parallel, obtaining more accurate results.

Aiming to provide a good continuous speech recognition performance with the connected models the situation becomes significantly more computationally demanding and in fact impractical, because Equation (2.9) would have to be evaluated for every possible phoneme sequence in the language. To overcome this problem, the Viterbi algorithm [Viterbi, 1967] yields an effcient dynamic programming method used to obtain a locally optimal estimate of the phoneme sequence w.

Dynamic pogramming is used extensively in speech recognition. It is a method of solving an initial problem, usually a big and complex one, through the use of a recursive techique that divides the large problem into many smaller subproblems and finds the optimal solution to these subproblems by dividing them again and again until a trivial case is reached. The solution to the trivial case is found in a constant time. Then the optimal smaller solutions are overlapped to find the optimal solution for the initial big problem. This optimal structure speeds up the determination of the main solution.

In the speech recognition engine used in this thesis each phoneme (basic unit) is modeled by a 3-states HMM. More complex structures (words) are built by concatenating these phoneme HMMs. This leads to a big load of parameters needed by the HMMs. To overcome such a possible data insufficiency problem, the parameters of the Gaussian mixtures which define the output probabilities of the states of the HMM (set to 8 Gaussians per state), are shared across states of various phonemes. The states which share parameters in this manner are called *tied states* or *senones*. Their reason to be comes from the fact that if only the modeling of single base acoustic units, or basephones, is used then the system becomes impractical to be trained because of the vast number of states that result. Then to keep things manageable the HMM states are clustered into a much smaller number of groups,

which are then called senones, and all the states mapped into one senone share the same underlying statistical model. The sharing is done in such a way as to preserve the "individuality" of each HMM, in that only the states with the most similar distributions are "tied". If the degree of tying is small, a larger number of possibly dissimilar states may be tied, resulting in a few too generalized senones that will cause a poor recognition performance. On the other hand, if this degree of tying is too large, there may not be enough data available to estimate the parameters of all the Gaussian mixtures. Then an obvious compromise is to be reached here in order to achieve a good performance of the system at a reasonable amount of resources. The state tying can reduce the total number of HMM states by one or two orders of magnitude. More information on procedures for this parameter sharing can be found in [Hwang and Huang, 1993].

2.6 Key Decoding Issues

The most important issues about the efficient use of a decoder are those relating to its decoding speed, memory usage and power consumption. The performance of a decoder is often dependent on the tradeoff between settings applied to minimize resource usage and maximuze speed, those necessary to acomodate a particular size and complexity of the acoustic models, language models and dictionary, and the recognition accuracy. Many of these tradeoffs must be decided at the training stage. Further changes on settings are done in the decoder.

For managing memory efficiently there are a few strategies that should be taken into consideration in order to obtain the best possible results for a particular case. The first one is not an efficient memory management technique itself: it consists on limiting the *heapsize*, which is the amount of memory allowed to the system that executes the recognition program. It is more related to computer science than to anything else, but it is a naive way of dealing with the memory overflow problem. The rest of the methods are described in the following subsections.

2.6.1 Active Lists and Beamwidths

The proper specification of the of the size of what are called *Active Lists* enables controlling both memory usage and speed. At any time during the search, an active list is comprised of those Gaussian densities which must be explicitly computed by the decoder given the current data vector. This is a subset of all Gaussians present in the acoustic models being used by the decoder and have been reached by current paths in the trellis.

Now referring to the *Beamwidths*, the absolute beam width is conventionally a fixed number that defines the maximum size of the active list. The relative beam width varies from time instant to time instant and is defined with reference to the maximum scoring node in the decoder's trellis at the given time instant. An existing threshold determines that the nodes which have a score lower than this threshold are not allowed to propagate further.

2.6.2 Language Weight

The language weight decides how much relative importance is given to the actual acoustic probabilities of the words in the hypothesis. A low language weight gives more leeway for words with high acoustic probabilities to be hypothesized, at the risk of hypothesizing spurious words.

In order to be more precise in the explanation of this matter, a few mathematic concepts are taken in the following parts.

Language Model

Speech recognition systems treat the recognition process as one of *maximum-a-posteriori* estimation, where the most likely sequence of words is estimated, given the sequence of features vectors for the speech signal. Mathematically, this can be represented as Equation (2.10).

$$W1\ W2\ W3\ ... = argmax_{Wd1\ Wd2\ ...}\{P(Z|Wd1\ Wd2\ ...)P(Wd1\ Wd2\ ...)\} \tag{2.10}$$

In Equation (2.10) ‘W1 W2 ...’ represent the recognized sequence of words, and ’Wd1 Wd2 ...’ represent any sequence of words. The argument on the right hand side of Equation (2.10) has two components: the probability of the features vectors given a sequence of words $P(Z|Wd1\ Wd2\ ...)$, and the probability of the sequence of words itself $P(Wd1\ Wd2\ ...)$. The first component is given by the HMMs and the second component, *a.k.a.* the language component, is provided by a language model.

The most commonly used language models are the N-gram language models . These models assume that the probability of any word in a sequence of words depends only on the previous N words in the sequence. Thus, a 2-gram model or *bigram* language model would compute $P(Wd1\ Wd2\ ...)$ as stated in Equation (2.11).

$$P(Wd1\ Wd2\ Wd3\ ...) = P(Wd1)P(Wd2|Wd1)P(Wd3|Wd2)\ ... \quad (2.11)$$

Similarly, a 3-gram or *trigram* model would compute it as stated in Equation (2.12).

$$P(Wd1\ Wd2\ Wd3\ ...) = P(Wd1)P(Wd2|Wd1)P(Wd3|Wd2, Wd1)\ ... \quad (2.12)$$

Language Weight

Although strict *maximum-a-posteriori* estimation would follow Equation (2.10), in practice the language probability is raised to an exponent for recognition. Although there is no clear statistical justification for this, it is frequently referred to as “balancing” of language and acoustic probability components during recognition and is known to be very important for good recognition. The recognition thus becomes Equation (2.13).

$$W1\ W2\ W3\ ... = argmax_{Wd1\ Wd2\ ...}\{P(Z|Wd1\ Wd2\ ...)P(Wd1\ Wd2\ ...)^{\alpha}\} \quad (2.13)$$

In Equation (2.13) α is the language weight.

2.6.3 Word Insertion Penalty

The insertion penalty decides how much penalty to apply to a new word during the search. If new words are not penalized, the decoder would tend to hypothesize the smallest words possible since every new word inserted leads to an additional increase in the score of any path as a result of the inclusion of the inserted word's language probability from the language model.

In [Takeda *et al.*, 1998] it is hypothesized that the Word Insertion Penalty (WIP) compensates the probability given by a language model to the true probability. This statement comes from the merit of combining acoustic knowledge and language knowledge through stochastic modeling. According to the general Bayes rule expressed in Equation (2.14), disregarding denominator, a simple product of acoustic and linguistic probabilities gives a score to word sequence hypotheses even if the acoustic and linguistic models are estimated independently.

$$P(W|Z) = \frac{P(Z|W)P(W)}{P(Z)} \tag{2.14}$$

However, in a real system balancing between acoustic and linguistic parameters the system's performance needs to be optimized. The typical form of combining the two probabilities is shown in Equation (2.15).

$$logP(Z|W) + \alpha logP(W) - nQ \tag{2.15}$$

Equation (2.15) contains the parameters α, which represents the language weight (LW) , Q represents the word insertion penalty (WIP) and n represents the number of words included in the sequence W. As is can be seen, giving different values to these parameters can provide a solution for fine tuning and optimization.

2.6.4 Performance Evaluation

The term *accuracy* indicates the percentage of words in the test set that were correctly recognized. However, this is not a sufficient metric; it is possible to correctly hypothesize all the words in the test utterances merely by hypothesizing a large number of words for each word in the test set. The

spurious words, called insertions, must also be penalized when measuring the performance of the system. The Word Error Rate (WER) indicates the number of hypothesized words that were erronious as a percentage of the actual number of words in the test set. This includes both words that were wrongly hypothesized (or deleted) and words that were spuriously inserted. Since the recognizer can, in principle, hypothesize many more spurious words than there are words in the test set, the percentage of errors can actually be greater than 100.

Stentence accuracy is a very important measure in tasks where absolutely correct recognition is necessary, such as recognizing an identity or a credit card number, or in machines with critical-outcome responses.

Studying the errors made in the recognition process often give an idea of what might be done to improve recognition. Al least this tool provides an additional degree of analysis of the system.

2.6.5 Livemode Decoding

The livemode decoder must be optimized in many ways with settings that are very specific to a given machine and task in order to obtain good performance results. This part obeys to the main goal aimed at this thesis and it will be treated in more detail in the Practice Part of it.

Chapter 3

Sphinx-4: A Java ASR engine

In this chapter, the basics of this Speech Recognition engine are described, focusing on the parts that are relevant to the development of Magnus.

3.1 Introduction

Sphinx-4 is a flexible, modular and pluggable framework to help foster new innovations in the core research of Hidden Markov Model recognition systems. This is the main description that the authors of Sphinx-4 give and that is exactly what it is.

Sphinx-4 is a very flexible ASR system that has been programmed using Java, which makes it very attractive to the development of applications for any platform, say operating system, basically Microsoft Windows and Unix-like systems. This is one the main points that support its choice for this project. Since the Department of Education from Generalitat de Catalunya uses GNU/Linux extensively, moreover, it has its own GNU/Linux distribution, named Linkat, it has been an excellent opportunity to test the software on the most extended and supported free operating system in the world.

Sphinx-4 stands for one of the most complete, robust and stable speech recognition systems nowadays. This state of the art framework includes many different ways of resolving one single problem, which makes it very flexible software able to be implemented on almost any situation, whether it is used on an application for handicapped people, on a speech research laboratory

or inside a car, the many possibilities that offers makes it very adaptive for any environment. This is an advantage over HTK, which stands for Hidden Markov Model Toolkit, one the most consolidated speech recognition frameworks for university study. HTK has been programmed in C++ , which is a disadvantage when having to export the produced software to a platform different from the one used in development. In this way, since Sphinx-4 has been programmed in Java, which is an interpreted language, not a compiled one like C++, there's no need to compile any code for a specific platform: Java produces the named "bytecodes" which are then interpreted on any machine (any platform) where a Java Virtual Machine (JVM) is available.

The Java platform also provides Sphinx-4 with a number of other advantages:

- The rich set of platform APIs greatly reduces coding time.
- Built-in support for multithreading makes it simple to experiment with distributing decoding tasks across multiple threads.
- Automatic garbage collection helps developers to concentrate on algorithm development instead of memory leaks and dynamic memory management chaos.

On the downside, the Java platform can have issues with memory footprint. Also related to memory, some speech engines will directly access the platform memory directly in order to optimize the memory throughput during decoding. Direct access to the platform memory model is not permitted with the Java programming language. Despite of this, the Java platform has given very good results in the development and performance of the speech recognition application.

Anyway, HTK is also a very good speech recognition software, and since it is older than Sphinx-4, it should be stated that the latter resembles the first one in mode of operation and technical functionality, but by means of portability and software innovation, Sphinx-4 is the chosen one for Magnus. Moreover, the fact that Sphinx-4 is a younger project based on the former HTK may have helped its creators to improve any of the flaws that HTK could have had before. More information about HTK is available in [Young *et al.*, 2006].

And the advantage that it is also given as free software avoids the problem of needing to develop an entire system from scratch, and by its modular and pluggable implementation it incorporates design patterns from existing systems, with sufficient flexibility to support emerging areas of research interest, such as the Digital Signal Processing or the Artificial Intelligence (AI).

On specialized press – literature it can be read that regarding functionality, Sphinx-4 works equally well as another famous speech recognition system, ViaVoice from IBM, which used to be open source as well, but now the project has again been hidden from general public researchers. The advantage is that Sphinx-4 is now free to be used and improved and that its working environment is broadened because of the Java technology.

One last remark should be made, this time, concerning the free licence world. Magnus is a platform independent free software application, because it runs on a Java virtual machine, but the implementation of this virtual machine may be not free. The development platform for Magnus has been Linkat, version 2, the educational GNU/Linux distribution of the Government of Catalonia, which by default uses the Sun Microsystems Java Virtual Machine 1.5, which is not free software. But recently the specification of the Java Virtual Machine has been released, and there exists a functional free implementation, named IcedTea, which could be used instead in order to make the whole application fully compliant with the free software terms established by the Free Software Foundation. So, despite this thesis openly refers to Java as if it was the only virtual available machine, basically because the Sphinx-4 documentation uses it extensively, it should be taken into consideration the alternatives available that make Magnus a completely free software application.

This chapter explains in detail the operating modus of the Sphinx-4 engine, based on the Sphinx-4 Whitepaper and the Sphinx-4 Javadocs for the different classes that compose the application. If more information is needed please refer to [Gouvea *et al.*, 2004].

3.2 Framework – High Level Architecture

The Sphinx-4 framework has been designed with a high degree of flexibility and modularity, since each element in the system can be easily replaced or modified to test different module implementations without needing to modify other parts of the system.

The modular nature of Sphinx-4 was enabled primarily by the use of the Java programming language. In particular, the ability of the Java platform to load code at run time permits simple support for the pluggable framework, and the Java programming language construct of interfaces permits separation of the framework design from the implementation.

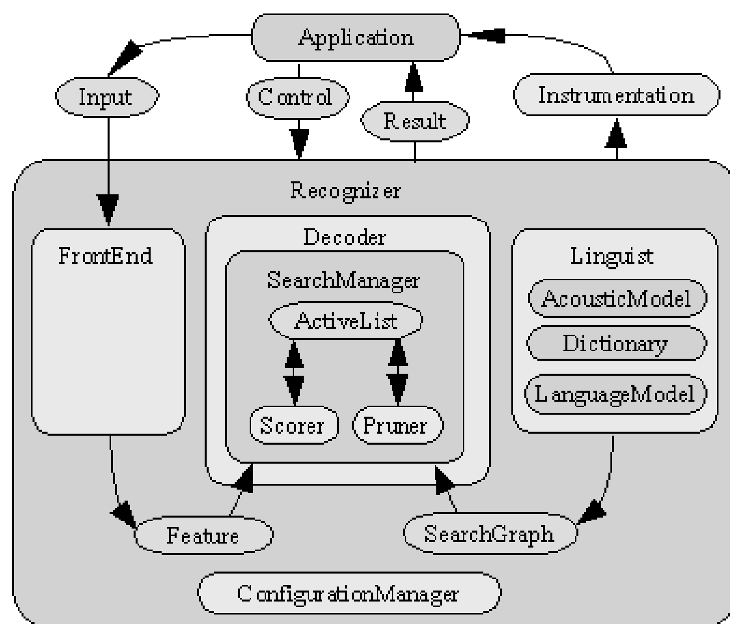

Figure 3.1: Sphinx-4 framework

The result of such a modular design is illustrated in Figure 3.1. As it can be seen, the high level architecture is relatively straightforward. From the Application's viewpoint, it puts the incoming signal data into the Input buffer, adjusts the Control register and gets the results through the Result

buffer. The Tools + Utilities module can also be used to obtain additional features. The speech recognizer itself gets the input data, it processes it through the several modules that compose its internal structure and drops the results into the Result buffer for the Application to read.

There are three primary modules in the Sphinx-4 framework: the FrontEnd, the Decoder and the Linguist. The FrontEnd takes the input data from the digitized signal (gathering), sets the data boundaries (annotating) and parameterizes it into a sequence of features (processing) to be read by the Decoder. The annotations provided include the beggining and ending of data segments and the operations performed include preemphasis, noise-cancellation, automatic gain control, end pointing, Fourier analysis, Mel spectrum filtering, cepstral extraction, etc.

The Linguist translates any type of standard language model, along with pronunciation information from the Dictionary and structural information from one or more sets of AcousticModels, into a SearchGraph. The SearchManager in the Decoder uses the features from the FrontEnd and the SearchGraph from the Linguist to perform the actual decoding, generating Results.

At any time prior to or during the recognition process, the application can issue Controls to each of the modules, effectively becoming a partner in the recognition process. These events allow the application to monitor and fine tune the decoding process.

The CongurationManager gives Sphinx-4 the ability to dynamically load and configure modules at run time, yielding a flexible and pluggable system.

To give applications and developers the ability to track decoder statistics such as word error rate, runtime speed, and memory usage, Sphinx-4 provides a number of Tools. All these features, though, are not treated in this paper. The statistics published by the Sphinx-4 original authors are taken directly to get an orientation of the system's behavior.

And finally, Sphinx-4 also provides an Utilities module that supports application-level processing of recognition results, used to quantize the per-

fomance of the system. These tools are treated in the "Regression Tests" chapter in the Practice Part.

3.3 FrontEnd

The purpose of the FrontEnd is to parameterize an Input signal (*i.e.*, speech) into a sequence of output Features. As illustrated in Figure 3.2, the FrontEnd comprises one or more sequential chains of replaceable communicating signal processing modules called DataProcessors. Supporting multiple chains permits simultaneous computation of different types of parameters from the same input signal. This enables the creation of systems that can simultaneously decode using different parameter types.

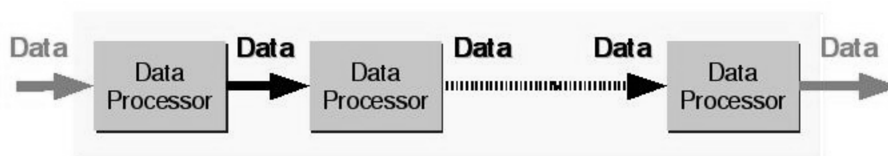

Figure 3.2: Sphinx-4 FrontEnd

Each DataProcessor in the FrontEnd provides an input and an output that can be connected to another DataProcessor, permitting arbitrarily long sequences of chains. The inputs and outputs of each DataProcessor are generic Data objects that encapsulate processed input data as well as markers that indicate data classification events such as end-points. The last DataProcessor in each chain is responsible for producing a Data object composed of parameterized signals, called Features, to be used by the Decoder.

Sphinx-4 permits the ability to produce parallel sequences of features. Sphinx-4 is unique, however, in that it allows for an arbitrary number of parallel streams.

The communication between blocks follows a pull design. With a pull design, a DataProcessor requests input from an earlier module only when needed, as opposed to the more conventional push design, where a module propagates its output to the succeeding module as soon as it is generated.

This pull design enables the processors to perform buffering, allowing consumers to look forwards or backwards in time.

Within the generic FrontEnd framework, Sphinx-4 provides a suite of DataProcessors that implement common signal processing techniques. These implementations include support for the following: reading from a variety of input formats for batch mode operation (e.g., extracting the spoken speech from a previously recorded signal), reading from the system audio input device for live mode operation, preemphasis, windowing with a Raised Cosine Transform (RCT), e.g., Hamming and Hanning windows, Discrete Fourier Transform (DFT) via the Fast Fourier Transform (FFT), Mel Frequency Filtering, Bark Frequency Warping, Discrete Cosine Transform (DCT), Linear Predictive Coding (LPC), end pointing, Cepstral Mean Normalization (CMN), Mel Frequency Cepstral Coefficient extraction (MFCC) and Perceptual Linear Prediction (PLP) coefficient extraction.

Using the ConfigurationManager it is possible to chain the Sphinx-4 DataProcessors together in any manner as well as incorporate DataProcessor implementations in any own design. As such, the modular and pluggable nature of Sphinx-4 not only applies to the higher-level structure of Sphinx-4, but also applies to the higher-level modules themselves (*i.e.*, the FrontEnd is a pluggable module, yet also consists of pluggable modules itself).

The ConfigurationManager consists basically of a source Extended Markup Language (XML) format file, which contains all the needed information to configure the system properly. The following subsections describe the specific configuration applied to the Magnus FrontEnd.

3.3.1 microphone

A microphone captures audio data from the system's underlying audio input systems and converts these audio data into Data objects.

This microphone will attempt to obtain an audio device with the format specified in the configuration file. If such device with a specific format cannot be obtained, it will try to obtain a device with an audio format that has a higher sampling rate than the configured sample rate, while the other

parameters of the format (*i.e.*, sample size, endianness, sign, and channel) remain the same. If, again, no such device can be obtained, it flags an error.

There's a configurable property for this peripheral that specifies whether or not the microphone will release the audio between utterances. On certain systems (GNU/Linux for one), closing and reopening the audio does not work too well. For this reason, this property is set to false.

3.3.2 speechClassifier

This class implements a level tracking endpointer invented by Bent Schmidt-Nielsen which can be referred to as [Schmidt-Nielsen *et al.*, 2004]. This endpointer is composed of three main steps:

1. Classification of audio into speech and non-speech.
2. Inserting SPEECH_START and SPEECH_END signals around speech.
3. Removing non-speech regions.

The first step, classification of audio into speech and non-speech, uses Bent Schmidt-Nielsen's algorithm. Each time audio comes in, the average signal level and the background noise level are updated, using the signal level of the current audio. If the average signal level is greater than the background noise level by a certain threshold value (configurable), then the current audio is marked as speech. Otherwise, it is marked as non-speech. The threshold value applied for Magnus is 13, which is an internal Sphinx-4 value that is compared to the average signal level to determine if the current audio is marked as speech or not. A lower threshold will make the endpointer more sensitive, that is, mark more audio as speech. A higher threshold will make the endpointer less sensitive, that is, mark less audio as speech.

The second and third step of this endpointer are documented in the classes speechMarker and nonSpeechDataFilter.

3.3.3 speechMarker

This class converts a stream of SpeechClassifiedData objects, marked as speech and nonspeech, into the separate regions that are considered speech. This is done by inserting SPEECH_START and SPEECH_END signals into the stream.

The algorithm is always in one of two states: 'in-speech' and 'out-of-speech'. If 'out-of-speech', it will read in audio until it hits audio that is speech. If more than 'startSpeech' amount of continuous speech is read, it is considered that speech has started, and a SPEECH_START is inserted at 'speechLeader' time before speech first started. The state of the algorithm changes to 'in-speech'.

Now consider the case when the algorithm is in 'in-speech' state. If it reads an audio that is speech, it is outputted. If the audio is non-speech, it is read ahead until 'endSilence' amount of continuous non-speech is acquired. At the point it is considered that speech has ended. A SPEECH_END signal is inserted at 'speechTrailer' time (for Magnus it has been experimentally set to 50, which represents the amount of time in milliseconds after speech ends to be included as speech data) after the first non-speech audio. The algorithm returns to 'out-of-speech' state. If any speech audio is encountered in between, the accounting starts all over again.

3.3.4 nonSpeechDataFilter

Given a sequence of Data, this class filters out the non-speech regions. The sequence of Data should have the speech and non-speech regions marked out by the SpeechStartSignal and SpeechEndSignal, using the speechMarker class. Such a sequence of Data for an utterance should look like one of the following two cases:

Case 1: Only one speech region

In the first case, illustrated in Figure 3.3 as a continuous time happening, the data stream has only one speech region.

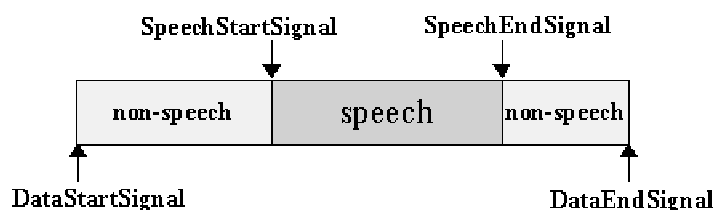

Figure 3.3: A data stream with only one speech region

After filtering, the non-speech regions are removed, and the data stream becomes Figure 3.4 with only on speech region after filtering.

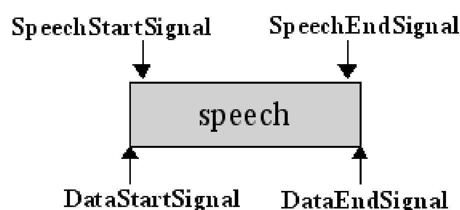

Figure 3.4: A data stream with only one speech region after filtering

Case 2: Multiple speech regions

The example of a data stream with two speech regions, Figure 3.5, is used to illustrate the case of a data stream with multiple speech regions.

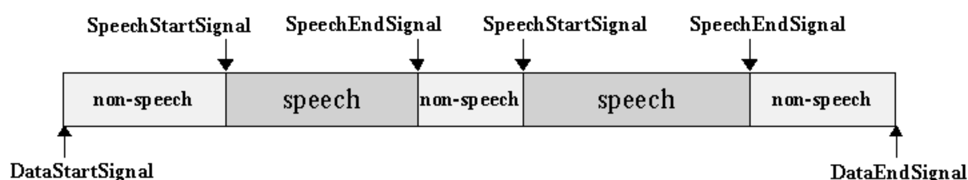

Figure 3.5: A data stream with two speech regions

This case is more complicated than the one with a single speech region. The property mergeSpeechSegments is very important for controlling the behavior of this filter. This property determines whether individual speech regions

(and the non-speech regions between them) in an utterance should be merged into one big region, or whether the individual speech regions should be converted into individual utterances. If mergeSpeechSegments is set to true, all the Data from the first SpeechStartSignal to the last SpeechEndSignal will be considered as one Utterance, enclosed by a pair of SpeechStartSignal and SpeechEndSignal (which itself becomes enclosed by a pair of DataStartSignal and DataEndSignal). All non-speech regions are removed from the stream. This gives Figure 3.6.

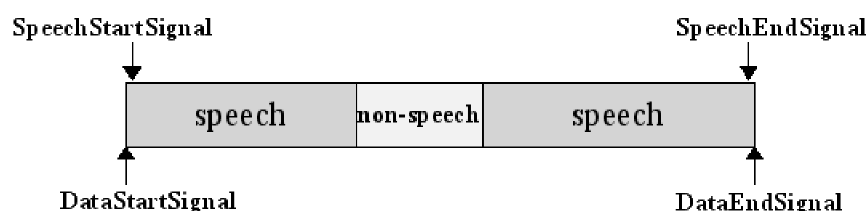

Figure 3.6: A data stream with two speech regions after filtering, when mergeSpeechSegments is set to true

On the other hand, if mergeSpeechSegments is set to false (the default), then each speech region will become an independent data stream. Pictorially, the data stream with two speech regions becomes Figure 3.7.

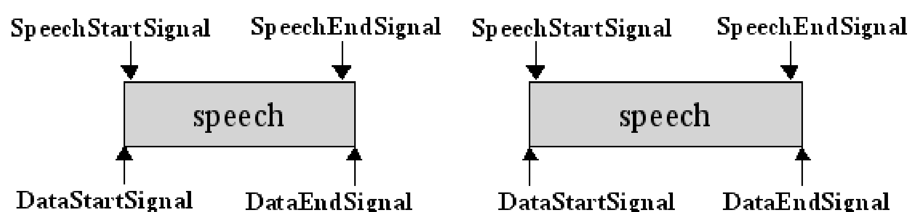

Figure 3.7: A data stream with two speech regions after filtering, when mergeSpeechSegments is set to false

3.3.5 preemphasizer

This class implements a high-pass filter that compensates for attenuation in the audio data. Speech signals have an attenuation (a decrease in intensity of a signal) of 20 dB/dec. It increases the relative magnitude of the higher

frequencies with respect to the lower frequencies because they usually contain much less energy, even though they are still important for speech recognition.

The preemphasizer takes a Data object that usually represents audio data as input, and outputs the same Data object, but with preemphasis applied. For each value $X[i]$ in the input Data object X, the Equation (3.1) is applied to obtain the output Data object Y, where i denotes the discrete time.

$$Y[i] = X[i] - \alpha(X[i-1]) \tag{3.1}$$

A common value for the preemphasis factor (α) is something around 0.97.

3.3.6 windower

This class slices up a Data object into a number of overlapping windows (usually referred to as "frames"). In order to minimize the signal discontinuities at the boundaries of each frame, each frame is multiplied with a raised cosine windowing function. Moreover, the system uses overlapping windows to capture information that may occur at the window boundaries. These events would not be well represented if the windows were simply juxtaposed.

The number of resulting windows depends on the window size and the window shift (commonly known as "frame shift"). Figure 3.8 shows the relationship between the original data stream, the window size, the window shift, and the windows returned.

The raised cosine windowing function is applied to each window. Since a window is returned, and multiple windows are created for each Data object, this is a one-to-many processor. Also note that the returned windows should have the same number of data points as the windowing function.

The applied windowing function, $W[n]$, of length N (the window size), is given by Equation (3.2).

$$W[n] = (1-a) - \left(a \cdot cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{3.2}$$

In Equation (3.2) 'a' is commonly known as the alpha value. For a value of 0.46 it results in a window named Hamming window. A value of 0.5 results

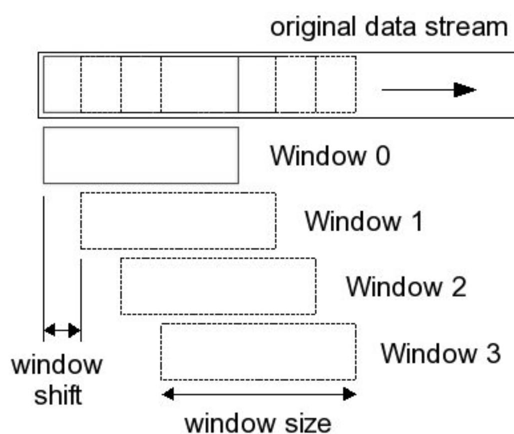

Figure 3.8: Relationship between original data, window size, window shift, and the windows returned

in the Hanning window. And a value of 0 results in the Rectangular window. The default for this system is the Hamming window, shown in Figure 3.9 with its corresponding spectral diagram. Using the default window size of 25.625ms and assuming a sample rate of 16kHz it yields 410 samples per window.

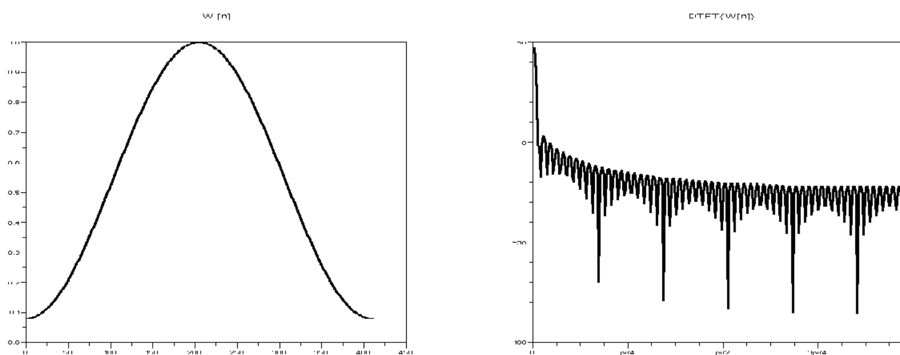

Figure 3.9: The Hamming window function with its corresponding spectral diagram

Two important features that define a windowing function (extracted from its spectral analysis) are the leakage (Lk), which is the dynamic range, thus the difference between the zero frequency magnitude of the spectral function and the magnitude of the main lobe, and the resolution (TWML), which is the frequency width of the central lobe. In the case of the Hamming window function, the Lk is 43dB and the TWML is $\frac{4\pi}{N} = 0.0306$ radians (N corresponds to the number of points taken for the window).

3.3.7 fft (Fast Fourier Transform)

This class computes the Discrete Fourier Transform (DFT) of an input sequence, using the Fast Fourier Transform (FFT) algorithm. The Fourier Transform (FT) is the process of analyzing a signal by its frequency components. The DFT is the discrete representation of the general FT and the FFT is an optimized algorithm, in terms of computational load and processing time, for sequences which length corresponds to a power of two.

As indicated in previous paragraphs, speech is analyzed at a constant frame rate by a window function. This window is the product of applying a sliding Hamming window to the signal. Moreover, since the amplitude is a lot more important than the phase for speech recognition, this class returns the power spectrum of a window of data instead of the complex spectrum. Each value in the returned spectrum represents the strength of that particular frequency for that window of data.

By default, the number of FFT points is the closest power of 2 that is equal to or larger than the number of samples in the incoming window of data. The length of the returned power spectrum is the number of FFT points, divided by 2, plus 1. Since the input signal is real, the FFT is symmetric, and the information contained in the whole vector is already present in its first half.

3.3.8 melFilterBank

The Mel frequency Filter Bank class filters an input power spectrum through a bank of a certain number of mel-filters. The output is an array of filtered values, typically called mel-spectrum, each corresponding to the result of filtering the input spectrum through an individual filter. Therefore, the length of the output array is equal to the number of filters created.

The triangular mel-filters in the filter bank are placed in the frequency axis so that each filter's center frequency follows the mel scale, in such a way that the filter bank mimics the critical band, which represents different perceptual effects at different frequency bands. Additionally, the edges are placed so that they coincide with the center frequencies in adjacent filters. Pictorially, the filter bank looks like Figure 3.10.

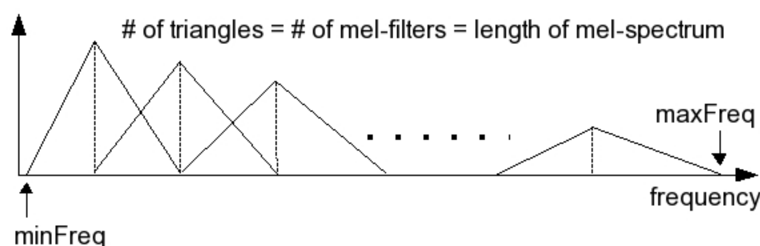

Figure 3.10: A Mel-filter bank

As it can be noticed in Figure 3.10, the distance at the base from the center to the left edge is different from the center to the right edge. Since the center frequencies follow the mel-frequency scale, which is a non-linear scale that models the non-linear human hearing behavior, the mel filter bank corresponds to a warping of the frequency axis. As can be inferred from the figure, filtering with the mel scale emphasizes the lower frequencies. A common model for the relation between frequencies in mel and linear scales is shown in Equation (3.3).

$$melFrequency = 2595 \cdot log\left(1 + \frac{linearFrequency}{700}\right) \tag{3.3}$$

The constants that define the filterbank are the number of filters, the minimum frequency, and the maximum frequency, which determine the frequency range spanned by the filterbank.

These frequencies depend on the channel and the sampling frequency that you are using. For clean speech, the minimum frequency should be higher than about 100Hz, since there is no speech information below it. Furthermore, by setting the minimum frequency above 50/60Hz, we get rid of the hum resulting from the AC power, if present.

The maximum frequency has to be lower than the Nyquist frequency, that is, half the sampling rate. Furthermore, there is not much information above 6800Hz that can be used for improving separation between models. Particularly for very noisy channels, maximum frequency of around 5000Hz may help cut off the noise. Typical values for the constants that define the filter bank are shown in Table 3.1.

Table 3.1: Values for the constants that characterize the filter bank

Constants name	Value
Sample Rate (Hz)	16000
numberFilters	40
minimumFrequency (Hz)	130
maximumFrequency (Hz)	6800

Davis and Mermelstein showed that Mel-frequency cepstral coefficients present robust characteristics that are good for speech recognition. For details, see [Davis and Mermelstein, 1980].

3.3.9 dct

This class applies a logarithm and then a Discrete Cosine Transform (DCT) to the input data, which is normally the mel spectrum obtained by the previous module. It has been proven that, for a sequence of real numbers, the discrete cosine transform is equivalent to the Discrete Fourier Transform. Therefore, this class corresponds to the last stage of converting a signal to cepstra, defined as the inverse Fourier Transform of the logarithm of the Fourier Transform of a signal. The dimensionalities of the coefficients that are actually returned are defaulted to be 13. When the input is mel-spectrum, the vector returned is the MFCC (Mel-Frequency Cepstral Coefficient) vector, where the 0-th element is the energy value.

3.3.10 liveCMN

This class subtracts the mean from all the input Data objects. CMN stands for Cepstral Mean Normalization. It does not read in the entire stream

of Data objects before it calculates the mean, but it estimates the mean from previous windows and subtracts it from the Data objects on the fly. Therefore, there is no delay introduced by liveCMN.

The Sphinx-4 properties that affect this processor are defined by the initial cepstral mean which is set to 12, the liveCMN window size which is set to 100 and the CMN shifting window, which specifies how many cepstrum after which the cepstral mean is recalculated, and is set to 160, which again is an internal value not bound to any specific unit.

$$\frac{cmnWindow}{cmnWindow + number\ of\ frames\ since\ the\ last\ recalculation} \tag{3.4}$$

The mean of all the input cepstrum calculated up to a moment is not reestimated for each cepstrum. This mean is recalculated after every CMN shifting window cepstra, and estimated by dividing the sum of all input cepstrum already processed. After obtaining the mean, the sum is exponentially decayed by multiplying it by the ratio given by Equation (3.4).

3.3.11 featureExtraction

This process is charged to the DeltasFeatureExtractor class which computes the delta and double delta of input cepstrum (or plp or ...). The delta is the first order derivative and the double delta (*a.k.a.* delta delta) is the second order derivative of the original cepstrum. They help model the speech signal dynamics (*i.e.* velocity and acceleration). The output data is a FloatData object with a float array of size three times the original cepstrum, formed by the concatenation of cepstra, delta cepstra and double delta cepstra. The output is the features vector used by the decoder. Figure 3.11 shows the arrangement of the output features data array.

Figure 3.11: Layout of the returned features

Suppose that the original cepstrum has a length of N, then the first N elements of the feature are just the original cepstrum, the second N elements are the delta of the cepstrum, and the last N elements are the double delta of the cepstrum.

Figure 3.12 below shows pictorially the computation of the delta and double delta of a cepstrum vector, using the last 3 cepstra and the next 3 cepstra

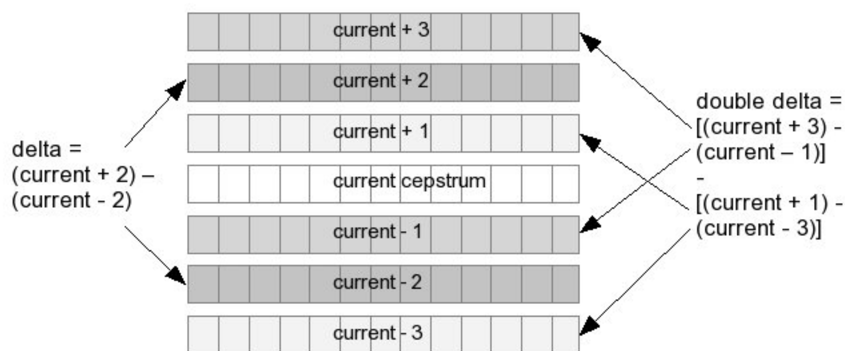

Figure 3.12: Delta and double delta vector computation

Referring to Figure 3.12, the delta is computed by subtracting the cepstrum that is two frames behind of the current cepstrum from the cepstrum that is two frames ahead of the current cepstrum. The computation of the double delta is similar. It is computed by subtracting the delta cepstrum one time frame behind from the delta cepstrum one time frame ahead. Replacing delta cepstra with cepstra, this works out to a formula involving the cepstra that are one and three behind and after the current cepstrum.

3.4 Linguist

The Linguist generates the SearchGraph that is used by the Decoder during the search, while at the same time hiding the complexities involved in generating this graph.

A typical Linguist implementation constructs the SearchGraph using the language structure as represented by a given LanguageModel and the topological structure of the AcousticModel (HMMs for the basic sound units used by the system). The Linguist may also use a Dictionary (typically a pronunciation lexicon) to map words from the LanguageModel into sequences of AcousticModel elements. When generating the SearchGraph, the Linguist may also incorporate sub-word units with contexts of arbitrary length, if provided.

Sphinx-4 provides an implementation of the Linguist that statically represents the search space as a flat graph, where each word in the vocabulary has its own branch, and its name is the FlatLinguist class. The FlatLinguist takes a grammar graph (as returned by the underlying, configurable grammar), and generates a search graph for this grammar.

3.4.1 LanguageModel

The LanguageModel module of the Linguist provides word-level language structure, which can be represented by any number of pluggable implementations. These implementations typically fall into one of two categories: graph-driven grammars and stochastic N-Gram models. The graph-driven grammar represents a directed word graph where each node represents a single word and each arc represents the probability of a word transition taking place. The stochastic N-Gram models provide probabilities for words given the observation of the previous $n-1$ words.

The format used for the implementation of the LanguageModel for Magnus is the JSGFGrammar. This format supports the Java Speech API Grammar Format (JSGF), which defines a BNF-style (Backus-Naur Form style) context-free, platform-independent and vendor-independent Unicode representation of grammars. This is a formal way to describe formal languages. Refer to [Knuth, 1964] for more details about the BNF-style.

3.4.2 Dictionary

The Dictionary provides pronunications for words found in the LanguageModel. The pronunciations break words into sequences of sub-word units

found in the AcousticModel. The Dictionary interface also supports the classication of words and allows for a single word to be in multiple classes.

Sphinx-4 currently provides implementions of the Dictionary interface to support the CMU Pronouncing Dictionary. The FullDictionary class used in Magnus creates a dictionary by reading in an ASCII-based Sphinx-3 format dictionary. Each line of the dictionary specifies the word, followed by spaces or tab and the pronuncation (by way of the list of phones) of the word. Each word can have more than one pronunciation. For example, as it appears in the Sphinx-4 Javadoc, a digits dictionary would look like:

```
ONE                  HH W AH N
ONE(2)               W AH N
TWO                  T UW
THREE                TH R IY
FOUR                 F AO R
FIVE                 F AY V
SIX                  S IH K S
SEVEN                S EH V AH N
EIGHT                EY T
NINE                 N AY N
ZERO                 Z IH R OW
ZERO(2)              Z IY R OW
OH                   OW
```

In the example, the words “one” and “zero” have two pronunciation transcriptions each.

This dictionary will read in all the words and its pronunciation(s) at startup. Therefore, if the dictionary is big, it will take longer to load and will consume more memory.

3.4.3 AcousticModel

The AcousticModel module provides a mapping between a unit of speech and a HMM that can be scored against incoming features provided by the FrontEnd. As with other systems, the mapping may also take contextual

and word position information into account. For example, in the case of triphones, the context represents the single phonemes to the left and to the right of the given phoneme, and the word position represents whether the triphone is at the beginning, middle, or end of a word (or is a word itself). The contextual definition is not fixed by Sphinx-4, allowing for the definition of AcousticModels that contain allophones as well as AcousticModels whose contexts do not need to be adjacent to the unit.

Typically, the Linguist breaks each word in the active vocabulary into a sequence of context-dependent sub-word units. The Linguist then passes the units and their contexts to the AcousticModel, retrieving the HMM graphs associated with those units. It then uses these HMM graphs in conjunction with the LanguageModel to construct the SearchGraph.

Unlike most speech recognition systems, which represent the HMM graphs as a fixed structure in memory, the Sphinx-4 HMM is merely a directed graph of objects. In this graph, each node corresponds to an HMM state and each arc represents the probability of transitioning from one state to another in the HMM. By representing the HMM as a directed graph of objects instead of a fixed structure, an implementation of the AcousticModel can easily supply HMMs with different topologies. For example, the AcousticModel interfaces do not restrict the HMMs in terms of the number of states, the number or transitions out of any state, or the direction of a transition (forward or backward). Furthermore, Sphinx-4 allows the number of states in an HMM to vary from one unit to another in the same AcousticModel.

Each HMM state is capable of producing a score from an observed feature. The actual code for computing the score is done by the HMM state itself, thus hiding its implementation from the rest of the system. The Acoustic-Model also allows sharing of various components at all levels. That is, the components that make up a particular HMM state such as Gaussian mixtures, transition matrices and mixture weights can be shared by any of the HMM states to a very fine degree (this was previously referred to as senones).

The acoustic models used in Magnus have been extracted from the Wall Street Journal articles. The structure of the features used follows the construction "Cepstra + Delta + DoubleDelta", resulting in a vector of 39 coef-

ficients as discussed in the featureExtraction subsection above, using a sampling frequency of 16KHz, 40 mel filters and a frequency range between 130Hz to 6800Hz.

3.4.4 SearchGraph

No matter how the Linguist may be implemented, the search spaces are all represented as a SearchGraph. Illustraded in Figure 3.13, the SearchGraph is the primary data structure used during the decoding process. It is composed of optionally emitting SearchStates and SearchStateArcs with transition probabilities. Each state in the graph can represent components from the LanguageModel (words in rectangles), Dictionary (sub-word units in dark circles) or AcousticModel (HMMs).

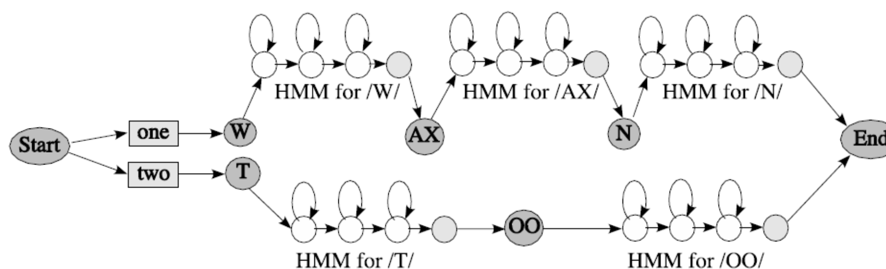

Figure 3.13: Example SearchGraph

The SearchGraph is a directed graph in which each node, called a SearchState, represents either an emitting or a non-emitting state. Emitting states can be scored against incoming acoustic features while non-emitting states are generally used to represent higher-level linguistic constructs such as words and phonemes that are not directly scored against the incoming features. The arcs between states represent the possible state transitions, each of which has a probability representing the likelihood of transitioning along the arc.

The SearchGraph interface is purposely generic to allow for a wide range of implementation choices, relieving the assumptions and hard-wired constraints found in previous recognition systems. In particular, the Linguist places no inherent restrictions on the following:

- Overall search space topology.
- Phonetic context size.
- Type of grammar (stochastic or rule based).
- N-Gram language model depth.

A key feature of the SearchGraph is that the implementation of the SearchState need not be fixed. As such, each Linguist implementation typically provides its own concrete implementation of the SearchState that can vary based upon the characteristics of the particular Linguist. For instance, a simple Linguist, like the FlatLinguist used in Magnus, may provide an in-memory SearchGraph where each SearchState is simply a one-to-one mapping onto the nodes of the in-memory graph. On the other hand, a Linguist representing a very large and complex vocabulary, however, may build a compact internal representation of the SearchGraph and dynamically expand this compact representation on demand. The choice between static and dynamic construction of language HMMs depends mainly on the vocabulary size, language model complexity and desired memory footprint of the system.

The FlatLinguist is appropriate for recognition tasks that use context-free grammars (CFG) ,*a.k.a.* Phrase Structure Grammars, like the Backus-Naur form used in the JSGFGrammar. It generates the SearchGraph directly from this internal Grammar graph, storing the entire SearchGraph in memory. As such, the FlatLinguist is very fast, yet has difficulties in handling grammars with high branching factors.

3.5 Decoder

The primary role of the Sphinx-4 Decoder block is to use the Features from the FrontEnd in conjunction with the SearchGraph from the Linguist to generate Result hypotheses. The Decoder block comprises a pluggable SearchManager and other supporting code that simplifies the decoding process for an application. As such, the most interesting component of the Decoder block is the SearchManager.

The Decoder merely tells the SearchManager to recognize a set of Feature frames. At each step of the process, the SearchManager creates a Result object that contains all the paths that have reached a final non-emitting state. To process the result, Sphinx-4 also provides utilities capable of producing a lattice and confidence scores from the Result.

Like the Linguist, the SearchManager is not restricted to any particular implementation. For example, implementations of the SearchManager may perform search algorithms such as frame-synchronous Viterbi, A*, bi-directional, and so on. Magnus makes use of the SimpleBreadthFirstSearchManager class, which performs a Breadth First Search (BFS) strategy on the graph generated with the possible state transitions, beginning at the root node and exporing all the neighboring nodes in a top-down hierachial manner. This frame synchronous Viterbi search is called on each frame.

The SearchManager implementation uses a token passing algorithm as described in [Young *et al.*, 1989]. A Sphinx-4 token is an object that is associated with a SearchState and contains the overall acoustic and language scores of the path at a given point, a reference to the SearchState, a reference to an input Feature frame, and other relevant information. The SearchState reference allows the SearchManager to relate a token to its state output distribution, context-dependent phonetic unit, pronunciation, word and grammar state. Every partial hypothesis terminates in an active token.

As it is a common technique Sphinx-4 provides a sub-framework to support the SearchManager composed of an ActiveList, a Pruner and a Scorer. The SearchManager sub-framework generates ActiveLists from currently active tokens in the search trellis by pruning using a Pruner. In Magnus the ActiveLists, maintained as sorted lists, are managed with the PartitionActiveListFactory class.

The implementation of the Pruner is greatly simplifed by the garbage collector of the Java platform. With the garbage collection, the Pruner can prune a complete path by merely removing the terminal token of the path from the ActiveList. The act of removing the terminal token identifies the token and any unshared tokens for that path as unused, allowing the garbage

collector to reclaim the associated memory. In Magnus, the class in charge of this process is named SimplePruner.

The SearchManager sub-framework also communicates with the Scorer, a state probability estimation module that provides state output density values on demand. When the SearchManager requests a score for a given state at a given time, the Scorer accesses the features vector for that time and performs the mathematical operations to compute the score. The Scorer retains the information pertaining to the state output densities. Thus, the SearchManager need not know whether the scoring is done with continuous, semi-continuous or discrete HMMs. Furthermore, the probability density function of each HMM state is isolated in the same fashion. Any heuristic algorithms incorporated into the scoring procedure for speeding it up can also be performed locally within the scorer. In addition, the scorer can take advantage of multiple CPUs if they are available. The ThreadedAcousticScorer is the name of the class that implements the acoustic scorer in Magnus.

Chapter 4

Speech Enhancement

This chapter presents the speech enhancement method applied to Magnus.

[Shankar Chanda and Park, 2007] provide a proposal which has been taken as baseline for this project. Refer to this article for further details.

4.1 Introduction

Due to the variety of environments where Magnus may be launched there is the need of developing a system to provide a degree of ensurance of the performance of the application. Such performance, say intelligibility of the spoken words, is degraded as a function of the ambient noise where the program is immersed.

In order to improve its functioning [Shankar Chanda and Park, 2007] propose a low complexity system to increase the intelligibility of far-end clean speech signal to a listener who is located in such environment. Intelligibility of the spoken words is generally associated with the formant structure of speech signal.

We start from the hypothesis that the environment contains moderate or high level of noise. To mitigate the problem of degradation of the intelligibility of the spoken words by the ambient noise, a common practice is the increase of the power of speech towards a greater Signal to Noise Ratio

(SNR). However, increasing speech power often causes discomfort and listening fatigue to the listener, especially when the speech power has to be raised to a favorable SNR in presence of heavy ambient noise. Then in such scenarios the enhancement of the perceptual features of the spoken words related with speech intelligibility are called.

Experimental results referenced in the article indicate that the first formant alone is a very minor contributor to the intelligibility of speech, whereas a strong correlation is observed between the intelligibility of speech and the second formant frequency. It is also known that the consonants play more significant role compared to the vowels in carrying the speech intelligibility cues even though the consonants are significantly weaker than vowels in phonetic power. As consonants carry less phonetic power, they are more prone to be affected by noise when speech is reproduced in an environment with moderate or high ambient noise level. The different frequency bands in speech contribute differently to the intelligibility of the spoken words. Frequency range from 1.5 KHz to 3.5 KHz has more contribution in the intelligibility of the spoken words compared to the rest of the speech spectrum. This data makes sense compared to the frequencies of the first vocal formants given by [Fant, 1970] and shown in Table 4.1.

Table 4.1: Frequential limits of the vocal formants

Formant	f_1	f_2
$[a]$	750	1300
$[e]$	500	1800
$[i]$	300	2000
$[o]$	500	1000
$[u]$	300	700

In literature different speech intelligibility enhancement techniques have been proposed based on the enhancement of the perceptual cues that are associated with the intelligibility of the spoken words, but many of these systems involve high computational and storage complexity that often preclude their implementation on resource limited platforms. This can be particularly

attributed to the speech enhancement systems that are based on analysis of the spectral components of input speech in frequency domain and selective enhancement of a subset of these frequency components. It is desirable that while increasing the intelligibility of speech the enhancement system should preserve the clarity of the input speech. The intelligibility should be enhanced with minimal introduction of audible processing artifacts.

[Shankar Chanda and Park, 2007] propose a low-complexity approach to enhance the intelligibility of speech amenable for implementation in resource-constrained platforms. In this proposal, the consonants of the speech signal are enhanced by processing the input speech using a tunable band-pass shelving filter whose cutoff frequency is dynamically adjusted. The proposed system preserves the speech clarity well without introducing audible distortions. Excess sibilant levels are sometimes produced due the boost in the high frequency region of the unvoiced fricatives. To mitigate this problem a vocal de-esser is used in conjunction with the speech enhancement unit.

Finally, one last remark should be made about the article and its relation with Magnus. The speech enhancement system proposed in the paper works on the output channel of the system, thus offering an improved, enhanced, speech with repect to the input speech, but the speech enhancement unit used in this project works on the input channel, thus offering an enhanced speech to the system in order to obtain better recognition results, especially when the environment conditions are not favorable.

4.2 Speech intelligibility enhancement system

A block diagram of the proposed system is shown in Figure 4.1 as illustrated in the referenced article.

The input speech is filtered by a high pass shelving filter whose cut-off frequency is adjusted dynamically so that the level of the output speech is approximately equal to the level of the input speech. The shelving filter is having a gain greater than unity in the high frequency range whereas in the low frequency range the gain of the shelving filter is less than unity.

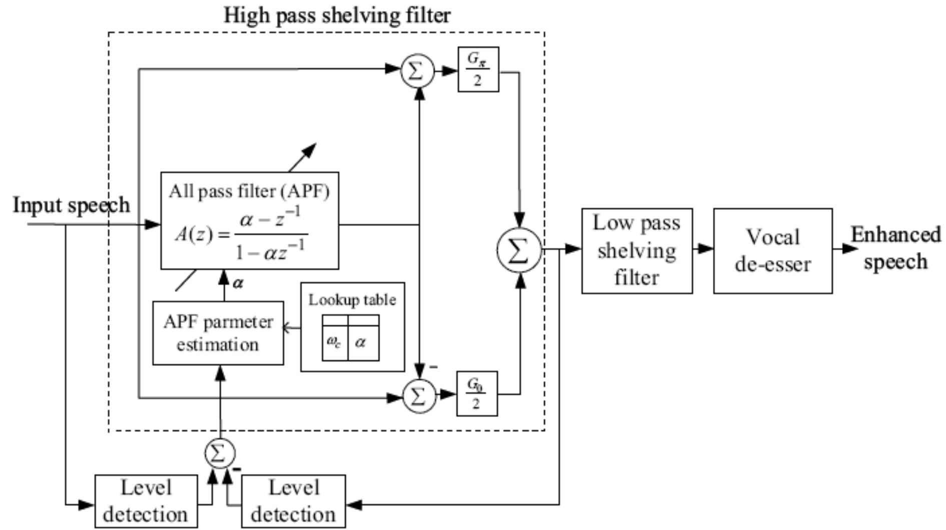

Figure 4.1: The proposed speech intelligibility enhancement system

According to the frequential particularities that exist within the transitions from a vowel to a consonant and *vice versa* the system responds consequently reshaping the input high-pass shelving filter so that the power of the consonants is boosted, resulting in an increase of the intelligibility of the speech. Vowels carry a higher energy level compared to consonants and are frequentially located in the lower part of the spectre. This justifies the use of the high-pass filter with a slight boost at the highest part of the frequential axis (and also a slight attenuation at the lowest part). The parameter that is continuously readjusted is the cut-off frequency of this filter according to the difference between the power of the plain input speech signal and the power of the filtered speech. In 4.1 the block in charge of the adjustement is the *APF parameter estimation* and the *Level detection* modules compute the power (RMS) of the input and output signals.

This difference is then mapped to a new desired cut-off frequency indicated by the *Lookup table* module and finally the *APF parameter estimation* module establishes a smooth transition over time obtained with a predefined controller. This transition is shown as a coefficients vector which is then used to dynamically tune the filter while filtering the speech signal at the same time.

The idea is to keep the input speech level equal to the filtered speech level. If this statement is verified, the consonant sounds should be boosted and thus, the intelligibility of the speech enhanced. In order to accomplish this purpose the cut-off frequency of the tunable shelving filter is changed in such a way that there is an upper-limit beyond which the cut-off frequency is not moved even if the output speech level is grater than the input speech level. As a result in case of most of the consonants the equality between the input speech level and output speech level is not satisfied and the consonants get a boost in their phonetic power.

The module in the middle of Figure 4.1 consists of a low-pass second order shelving filter with a cut-off frequency of 6KHz. This filter reduces the excess boost of high frequency components beyond its cut-off frequency produced in the previos module.

As it is referenced in [Shankar Chanda and Park, 2007], from exprerimental results it is observed that the proposed enhancement system sometimes produced excess *sibilant* levels. A sibilant level is the "ess" vocal sound that is generated while producing unvoiced fricatives such as the 's' in 'sound'. A sibilant is characterized by its predominantly high frequency content that has a sharp amplitude peak. Most of the energies of the sibilant vocals are located above 2KHz.

To mitigate the problem of excess sibilant levels a third module is attached in the signal processing chain as shown in Figure 4.1: a vocal de-esser. In this unit, the input speech is split into high and low frequency components using a second-order low-pass shelving filter with cut-off frequency around 2KHz. When the ratio between the RMS level of the high pass band frequency content and the RMS level of the low-pass band frequency content exceeds a certain threshold a decision is taken in favor of the sibilant and the de-esser gain is reduced from unity to a lower value using a predetermined release time constant. When non-sibilant vocal appears at the input of the de-esser the gain is increased towards unity with a predetermined attack time constant.

4.3 Real-Time Implementation

In the article tunable high-pass shelving filter $H_{SH}(z)$ is implemented using an all-pass filter $A(z)$, leading to a low-sensitivity realization robust to the coefficient quantization. It is shown in Figure 4.1 that $H_{SH}(z)$ has a gain G_0 at zero frequency and a gain G_π at high frequency range. Equation 4.1 and Equation 4.2 refer to the implementation of $H_{SH}(z)$.

$$A(z) = \frac{\alpha - z^{-1}}{1 - \alpha z^{-1}} \tag{4.1}$$

$$H_{SH}(z) = \frac{G_\pi}{2}(1 + A(z)) + \frac{G_0}{2}(1 - A(z)) \tag{4.2}$$

The 3dB cut-off frequency ω_c of the filter is given by Equation 4.3.

$$\omega_c = cos^{-1}\left(\frac{2\alpha}{1 + \alpha^2}\right) \tag{4.3}$$

In this implementation the cut-off frequency and the filter gain can be changed independently of each other permitting an easy tuning of the system. For different values of ω_c, the corresponding values of the all-pass filter parameter α are stored in a lookup table. The estimated cut-off frequency of the shelving filter is proportional to the difference between the output speech level and the input speech level. Finally, the parameters G_0 and G_π are experimentally determined.

Part II

Practice

Chapter 5

Architecture, main components and software distributions

This chapter presents the serveral modules that compose the application and the way they interact altogether to obtain good speech recognition results.

5.1 Architecture

As Sphinx-4 is an incredibly modular application, Magnus, which is based on Sphinx-4, has followed up becoming a pluggable modular application too. This degree of freedom in the clear definition of the various components of the system is attributed to the possibilities that the Java programming language offers. Following the way Sphinx-4 is designed, Magnus has taken a lot of advantage from this characteristic. From all these features, the degree of modularity is obtained through the use of the Java interfaces and the pluggable advantage is obtained from the advanced configuration system that Sphinx-4 uses.

In the first place, it is important to note the difference between the two main approaches developed in this project: on one hand, the base application itself, on the other hand, the Sphinx-4 extensions that provide an enhanced speech recognition system. In order to explain and show them, Figure 5.1 is provided. Figure 5.1 represents both approaches with the main blocks that represent the main processes of each approach.

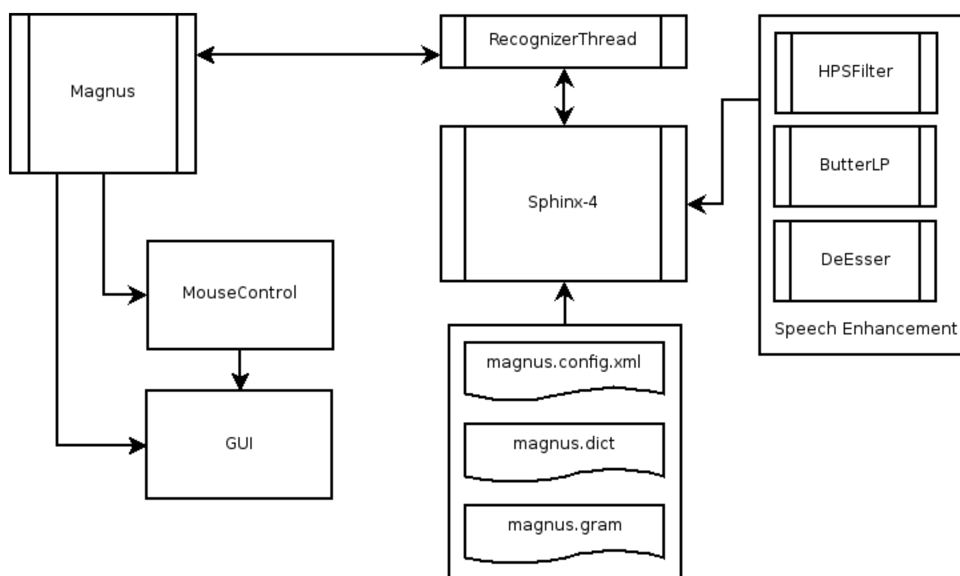

Figure 5.1: Magnus architecture and main components.

As it can be seen in Figure 5.1, the block at the top left corner of the figure, named Magnus, corresponds to the main process of the application, a well defined and sequential process. At this point, the two main approaches of the development of the project are clear: on the right han side of the figure the part that interacts with the speech recognition engine is shown, and below, the part corresponding to the interface and its usability.

5.2 Main components

Once the main parts of Magnus have been described let's focus on each of the components that build up the whole system.

5.2.1 User interface

This is the main input channel the users use to interact with the program. Moreover, the user interface is constituted by all the different components that enable Magnus to show signs of activity. This interface is basically graphical, so its is rather said to be a Graphical User Interface (GUI) although the application also sends several text messages to the standard out-

put of the system, say the screen. This standard output collects information messages as well as error messages, for what it is, to some extent, a more important means of system commmunication. Anyway, the aim of the project is to get Magnus running on as many platforms as possible, for what the GUI will be enough for the majority of users.

All the screens are launched originally by the **Magnus** class, which holds the main thread of the program. In Figure 5.1 all the screens and panels have been grouped into the GUI block. The several classes that integrate this box are implemented using the Abstract Window Toolkit (AWT) library except the JProgressBar class. They are all described below:

MyScreen This is the first screen shown by the application and it is created during its initialization. It shows the loading messages. This screen is launched by the Magnus class.

MyMenu This is the configurable menu that represents the main interface of Magnus. It is shown all the time the mouse pointer is quiet and displays the actions that the user is allowed to perform: moving upwards, downwards, leftwards, rightwards, clicking, dragging, opening a menu, switching from the mouse peripheral to the keyboard peripheral... This class also permits the customization of the look-and-feel of the program and automatically saves it on the user's computer when the application is closed, also loading it automatically whenever the user launches the program again. This menu is controlled by the Magnus class.

ConfigScreen This screen can be accessed through the MyMenu class and leads to a new screen where the background color and the size of the menu can be customized.

MyPanelLabels This is a support class for MyMenu which provides a panel with the labels of the different allowed actions.

MyPanelImages This is a support class for MyMenu that provides a panel with the images associated to the actions that are allowed to perform.

JProgressBar This is the only component that does not belong to the AWT library, but to the SWING library. It is normally used to provide a feedback of the loading progress of an application, but in Magnus it is

used as a vumeter, so to provide a feedback of the input voice signal level.

VUMeterThread This thread receives the input signal from the speech recognition engine and sets the JProgressBar accordingly so as to provide an activity feedback indicator. The Sphinx-4 Microphone class has slightly been modified/hacked so as to work along with this class.

Apart from the GUI, Magnus also makes use of the **MouseControl** class, which emulates a virtual mouse peripheral and a virtual keyboard peripheral. The acceleration of the virtual mouse pointer that Magnus controls is set by the MyMenu class through the ConfigScreen by setting a variable class of the MouseControl instance used by Magnus.

5.2.2 Sphinx-4 interaction

The interaction with the speech recognition engine is achieved through a thread. A *thread* is a different program flow that interacts with another flow (the main program flow) in a cooperative way. From the processor's point of view, or in the case of this thesis the Java Virtual Machine, a thread is a process that runs in parallel with the main application and interacts with it in such an exclusive way that it should not interfere, but in fact, it could interfere if the exclusion mechanism was inappropriate. Then, the parallelism is yielded to the processor (the operating system or the JVM) rather than to the software. Instead of emulating a parallel single process, with a thread, many processes request processor time as they are all instantiated in the kernel process scheduler. In the case of multiple processors, a true parallel execution is possible.

Magnus has a class named **RecognizerThread**, which is indeed a thread, that provides the interface between the main application and Sphinx-4, which is again launched in another thread. The communication mechanism is obtained through the use of a register and a flag. The RecognizerThread issues Sphinx-4 for new recognition results, and once they are ready, it loads the register (speechresult) with it and sets the flag (resultready) in order to alert Magnus to the new spoken command.

This class is also responsible for adapting the configuration files for Sphinx-4 to the user's computer, because this program needs to access the configuration files from the user's file system. This would be no problem if the application was to be launched always from the distributed binary packages, but if the Java Web Start service was to be used, the application would need to do it this way. This point will be dealt with more detail later in the software distributions section.

There are still three very important components developed for Magnus. They correspond to the speech recognition processing units for the FrontEnd chain: the HPSFilter, the ButterLP and the DeEsser classes, but since a complete chapter is dedicated to this enhancement technique they are rather left to be explained in due time.

5.2.3 Sphinx-4 configuration

The Sphinx-4 configuration manager system has two primary purposes:

- Determining which components are to be used in the system.
- Determining the detailed configuration of each of these components.

In order to obtain a modular and pluggable structure of the system, the configuration manager relies on a configuration file, which defines:

- The names and types of all the components of the system.
- The connectivity of these components.
- The detailed configuration for each of these components.

This configuration file is an Extended Markup Language (XML) file. Each of the components listed must implement de *Configurable* interface in order to define them correctly. The configuration data for the components are called *properties*, which are simple name/value pairs. If a property is omitted from the configuration file, the component will usually provide a default value for the property.

Sphinx-4 simple properties can be of the following types:

- boolean: the value can be either “true” or “false”.
- float: a single-precision floating point value.
- double: a double-precision floating point value.
- int: a 32 bit signed integer.
- String: a sequence of characters.
- Component: the name of a Sphinx-4 component.

In addition to these simple property types, there are two “list” types:

- String list: a list of strings.
- Component list: a list of components.

Lists are defined in a “propertylist” element. Each item in a list is defined with an item element.

The following description details the elements and attributes of the configuration file:

- <config>: the top level element. It has no attributes. It can have any number of the component, property and propertylist sub-elements.
- <component>: defines an instance of a component. This element must always have the name and type attributes.
- <property>: used to define a single property of a component or a global system property. This element must always have the name and value attributes.
- <propertylist>: used to define a list of strings or components. This element must always have the “name” element. It can be filled with any number of item sub-elements.
- <item>: The contents of this element define a string or a component name.

Hereunder, an example of the FrontEnd's basic configuration defined in the "magnus.config.xml" file:

```
<!-- ****************************************** -->
<!-- The live frontend configuration             -->
<!-- ****************************************** -->
<component name="epFrontEnd"
    type="edu.cmu.sphinx.frontend.FrontEnd">
    <propertylist name="pipeline">
        <item>microphone </item>
        <item>hpsf </item>
        <item>butterlp </item>
        <item>deesser </item>
        <item>speechClassifier </item>
        <item>speechMarker </item>
        <item>nonSpeechDataFilter </item>
        <item>premphasizer </item>
        <item>windower </item>
        <item>fft </item>
        <item>melFilterBank </item>
        <item>dct </item>
        <item>liveCMN </item>
        <item>featureExtraction </item>
    </propertylist>
</component>
```

Finally, let's notice the appearance of the "global properties", defined outside of any component, at the configuration level. Their use relies only on a matter of simplicity. These global variables can then be used in the property statements within the components' items.

For an example of a complete configuration file please refer to the appropriate file, "magnus.config.xml", found in the source code distribution of Magnus. The description of the different classes referred in the configuration file can be found in the Theory Part of the thesis (see Chapter 3).

The reader will notice that in Figure 5.1 there appear a couple of files along with the main configuration file described so far. These files are named

"magnus.gram" and "magnus.dict". They correspond to the grammar and dictionary configurations respectively.

The grammar configuration file, in the case that concerns this thesis, corresponds to a context free BNF-style grammar named Java Speech Grammar Format (JSGF). Refer to [Knuth, 1964] for more details about the BNF-style. This configuration file specifies the order the different components should have when speech was to be recognized, just as any grammar rule would specify. This grammar style can only be used with the Sphinx-4 FlatLinguist class.

The dictionary configuration file, working along with the Sphinx-4 FullDictionary class, provides a map between the speech words/units that the system is allowed to recognize and their corresponding phonetic transcription, which corresponds to the acoustic models trained by the system. With the information provided by this dictionary configuration file, the grammar configuration file and the acoustic models, the Linguist is able to build the search graph, which will be later on used by the search manager to decode the incoming speech and produce the expected results.

Again, for examples of complete dictionary and grammar configuration files refer to the corresponding files bundled in the source code distribution of Magnus.

5.3 Software distributions

Magnus can be obtained through different distribution means: the source code package, the binary package, the development checkout and the Java Web Start service.

5.3.1 Source code distribution

The source code package provides the totality of the Magnus Java source code. From this package the whole application can be built. In fact, all Magnus distributions have been produced from this package. This means of distribution is clearly oriented to developers.

In order to ease the process of compilation of the sources Apache Ant has been used. Ant is a tool for automating software building processes. The building process is described through a XML file, which indicates all the steps necessary to successfully carry out the different actions that contains. As it can be seen in Magnus, the file “build.xml” in the root folder of the project contains this information. The different actions can be assigned a predetermined order of precedence; it wouldn’t make sense to build the distribution compressed package before compiling the source code.

The different actions that can be launched from the command line at the root folder of the project are:

ant init This command calls the Sphinx-4 builfile and compiles the speech recognition engine, to then copy the needed files to the “lib” folder in Magnus. As this is the first action that Ant performs it doesn’t depend on any previous action.

ant compile This command depends on “init”. It first creates the “build” folder. Then it compiles the Magnus source code and puts the resulting bytecodes into the this folder.

ant dist This command depends on “compile”. It first creates the “dist” folder. Then it copies to the “build” folder all the images needed by the program as well as the configuration files. Finally it creates the jar distribution file by compressing the files in the “build” folder and placing the resulting jar file into the “dist” folder, previously having created the corresponding manifest file indicating the main class and the classpath.

ant run This command depends on “dist”. It launches the application.

ant document This command depends on “dist”. It is the default command, which means that if Ant is launched with no argument, this is the one taken by default. It first creates the doc folder, then creates the javadocs (documentation of the classes in HTML format) and finally dumps these file into the created folder.

ant clean This command does not depend on any other. It cleans up the Magnus application distribution by removing all the produced binary

files, leaving only the source code files. Then calls the Sphinx-4 buildfile to produce the same effect on the speech recognition engine, thus preparing the project for a complete new fresh compilation.

The source code distribution may be used to improve Magnus, or to adapt it to the needs of a particular user or for a particular purpose. The source packages available in the forge of Magnus correspond to stable and advanced enough releases of the software that deserve a versioning number. If the reader wishes to have the very latest version of Magnus, he or she would better check out the whole source distribution from the subversion repository. This alternative is described in the following subsection.

5.3.2 Development distribution

This means of distribution results in the obtention of the same kind of package as with the source code distribution, this one is indeed the source code of Magnus, but the version of the application may be a more recent release than the stable tarballs classified as the previous "source code distribution".

The reason of their existence is the use of a Control Versioning System (CVS) named Subversion. This system enables the developer to have a complete control on the versions of the source code of the software. It is a very practical and organized way of working.

If the reader (the end user) wishes to obtain the very latest version of the program, the following command shall be entered in the command line:

```
svn checkout https://forja.rediris.es/svn/csl2-magnus/trunk
```

Then a new folder will be created containing the whole checkout of the latest source code version.

Also with Subversion, the latest development version of the Scilab scripts used to produce the speech enhancement unit can be ckecked out though the following command:

```
svn checkout https://forja.rediris.es/svn/csl2-magnus/scilabworks
```

These scripts will be described in detail in the practice chapter dedicated to the speech enhancement unit.

5.3.3 Binary distribution

This means of distribution is meant for end users. The package contains the main Magnus class and all the needed dependencies. In order to launch the program successfully, the user should run:

```
java -jar -Xmx500m Magnus-<version>.jar
```

The reader will notice the extra `-Xmx500m` added in the command. This allows the JVM to use a maximum heap size of 500MB. Since Magnus, like any signal processing application, is a very intesive processing task which requires a lot of memory allocation, it is important that the system that runs such an application is able to dispose of a big amount of memory. Thus, by setting 500MB to the JVM compared to the default 256MB, the program is supposed to work more smoothly.

And one last remark about the previous command, in case the question arises: there is no need to set any classpath because the Magnus main class as well as the dependencies are in the same folder level.

5.3.4 Java Web Start distribution

This is the easiest way of running Magnus. There is no need to consciously download any package, the only action required by the user is the click on the right link lodged in the Magnus Project's Weblog.

The JWS service allows the whole automatization of the needed processes in order to automatically get the program running on any host computer. The link mentioned above points to a file that holds the configuration for getting the application. Its name is “Magnus.jnlp”. This service checks the host computer if Magnus has already been launched some time. In the case that it is the first time the program is requested, JWS will automatically

download it from the Internet and launch it afterwards. In the case that it's not the first time, JWS will compare the version held in the host computer to the version held in the servers, which is supposed to be the most recent one. Then, if the Internet version is the same as the local (host) version, Magnus will launch as usual, but if the Inernet version is a newer revision of the application, then JWS will upgrade the program automatically and launch it afterwards.

JWS is a service that empowers the security of the offered applications. Because of this, the files hosted in the servers need to be signed by a certified organization. Magnus does not provide such a certified signature, for what its developer has signed the program himself. If the reader trusts in the developer's word, then there will be no problem at all to launch Magnus. This paragraph refers to the alert that JWS issues every time an application without a certified signature is about to be launched. The official weblog of the project is presently available at:

`http://magnusproject.wordpress.com/`

If any sudden change happened, always trust the developer's homepage available at:

`http://www.salle.url.edu/~st12809/`

These sites are sure to host the true Magnus application, so there is no fear to be had of any malicious piece of code. Stay calm. :)

Chapter 6

Speech Enhancement Modules

This chapter treats in detail the analysis and design of the speech enhancement module for magnus based on the description provided in the Theory Part.

In order to work in an adequate and confortable environment adapted to the study and development of signal processing algorithms, the chosen framework has been Scilab, a very powerful open source platform for numerical computation, developed and maintained by INRIA, the French National Institute for Research in Computer Science and Control. Scilab is available at `http://www.scilab.org`.

6.1 Overview

The speech enhancement module for Magnus represents an extension to the FrontEnd processing chain of Sphinx-4. According to the documentation of the speech recognition engine, any module that aims to successfully integrate into the processing chain of the system must implement the "Configurable" interface, thus becoming a pluggable module, *i.e.* a component, like all the rest of the modules of the system.

According to [Shankar Chanda and Park, 2007], the speech enhancement module proposed improves the intelligibility of far-end clean speech signal to a listener who is located in such environment. The module is composed of three sub-modules: a tunable high-pass shelving filter, an excess boost

reducer and a vocal de-esser. Each of these sub-modules has been implemented as an independent data processor in the Sphinx-4 FrontEnd chain by extending the BaseDataProcessor class, which implements the Configurable interface. These components, like any abstract DataProcessor, implement elements common to all concrete DataProcessors, such as name, predecessor, and timer.

The life cycle of a component is as follows:

1. Class parsing: The class file is parsed in order to determine all its configurable properties.

2. Construction: The (empty) component constructor is called in order to instantiate the component. Typically the constructor does little, if any work, since the component has not been configured yet.

3. Configuration: The component's newProperties method is called with a PropertySheet containing the properties (usually taken from an external configuration file). The component should extract the properties from the property sheet.

And when it comes to interconnecting the different components, instead of hardcoding which subcomponents a particular subcomponent is interacting with, the component should use the configuration manager to provide the hookup to another component by defining configuration properties.

6.2 Tunable high-pass shelving filter

This component can be identified under the HPSFilter class name. It has two configurable properties:

- PROP_HPSF_Gz: This property defines the value of the gain of the filter at 2KHz. Its default value is arbitrarily set to 0.7.

- PROP_HPSF_Gp: This property defines the value of the gain of the filter at 6KHz. Its default value is arbitrarily set to 1.7.

As it can be seen in the above itemization, the frequency dynamic tuning range of the filter is enclosed between 2KHz and 6KHz. This responds to the range left between the highest vocal formant frequency and the domain of the consonant sound frequencies.

Figure 6.1 shows the diagram of this first module of the speech enhancement chain.

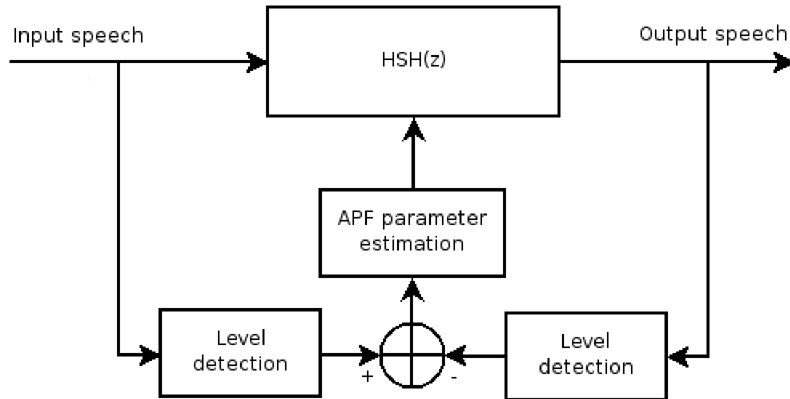

Figure 6.1: Tunable high-pass shelving filter diagram.

6.2.1 Level detection

This sub-module is implemented as a HPSFilter class function. It merely calculates the Root Mean Square (RMS) of the input signal. As usual, for a frame j of n elements (samples) it is calculated as Equation (6.1).

$$RMS(frame_j) = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} frame_j^2[i]} \tag{6.1}$$

6.2.2 APF parameter estimation

This sub-module is implemented as a HPSFilter class function. It first determines the new cut-off frequency by applying Equation (6.2).

$$newFc = presentFc - 833 * (RMS_{input} - RMS_{output}) \tag{6.2}$$

From experimental results it is shown that at the most sibilant part of speech, the difference in speech RMS levels stated in Equation (6.2) doesn't exceed the 0.6 value. By applying this calculation the system needs to process at least 4 frames to run through the whole allowed frequency range (from 2KHz to 4KHz). Each frame provided by the sound card brings around 10ms of speech, which results in a 40ms system delay until the filter is tuned stable. Taking into account that a reference phoneme lasts around 30ms at least, this method yields a comparable order of magnitude in its response. The reason for doing so is to prevent the estimator from oscillating around a desired cut-off frequency and thus distabilizing the system.

Once the new frequency is determined, it is checked that it does not exceed the limits permitted by the tunable filter. If it does, then the new cut-off frequency is truncated to the upper or lower bounds of the filter.

After finally determining the new cut-off frequency that the tunable high-pass shelving filter should have, a smooth transition from the old cut-off frequency to the new one should be obtained. In order to do so, a controller with two complex conjugate poles is proposed. The aim of this design is the obtention of a system with a step response similar to the one shown in Figure 6.2.

The design of such controller starts with the definition of two variables: τ, which indicates the time of stabilization of the system (this thesis associates this time with 5τ as if the system was an electric circuit), and tr, which indicates the rise time of the system, the time that passes by during the transition from the 10% to the 90% of the difference between the old output value and the new one.

Once the system's desired constants have been set, Equation (6.3) and Equation (6.4) can be computed. These equations compute two new variables that will be used in the definition of the controller in the Laplace domain.

$$\chi = \frac{-0.8 + \sqrt{0.8^2 + \frac{10\ tr}{\tau}}}{5} \tag{6.3}$$

$$w_0 = \frac{0.8 + 2.5\chi}{tr} \tag{6.4}$$

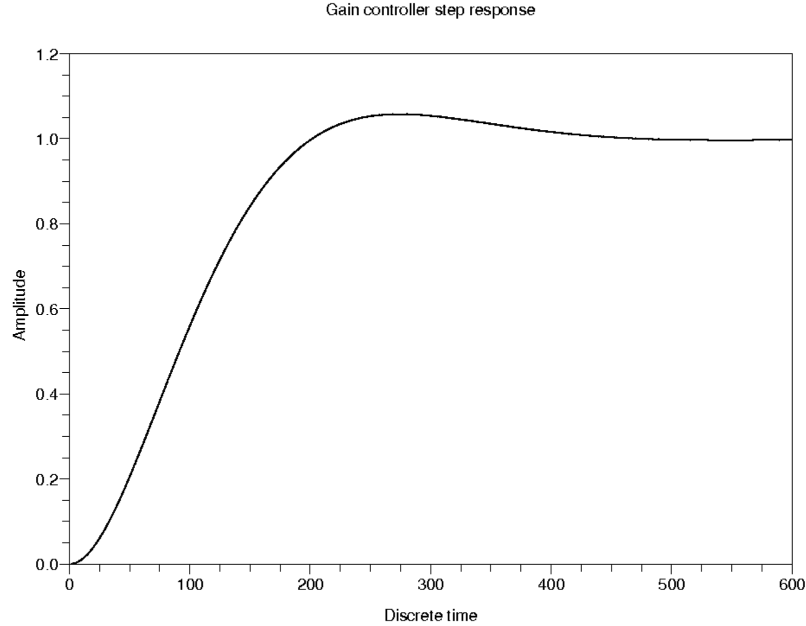

Figure 6.2: Step response of the desired controller.

Then, the controller $M(s)$ can be obtained in the Laplace domain through Equation (6.5).

$$M(s) = \frac{w_0^2}{s^2 + 2\chi w_0 s + w_0^2} \tag{6.5}$$

Since the system that results from Equation (6.5) is defined in a continuous domain it can't be used. Thus it needs to be discretized. In order to do so, the bilinear transform is proposed. This transform does not provide accurate results at high frequencies, the frequency warping is linear at low frequencies and logarithmic at high frequencies as it is shown in [Oppenheim *et al.*, 1983], but since the maximum frequential information used will be 4KHz, that is still considerably lower than the Nyquist's frequency, wich results at 8KHz (notice that the sampling frequency of the speech signals is 16KHz). Equation (6.6) defines the bilinear transform.

$$H(z) = H\left(s = \frac{2}{T}\frac{z-1}{z+1}\right) \tag{6.6}$$

Parameter T in Equation (6.6) refers to the sampling period, thus $T = 16000^{-1} = 62.5\mu s$. Finally, once the controller is designed, according to the sign of the frequency increment that must be applied to the cut-off frequency of the tunable high-pass shelving filter, a predetermined controller is created. In the case that the increment is positive, the controller $M(z)$ is defined with $\tau = 6ms$ and $tr = 6ms$ resulting in Equation (6.7), but if the increment is negative then the controller is defined with $\tau = 6ms$ but $tr = 20ms$ and thus obtaining Equation (6.8). This distinction corresponds to the specifications given at [Shankar Chanda and Park, 2007].

$$M(z) = \frac{0.0001107 + 0.0002215z + 0.0001107z^2}{0.9793837 - 1.9789408z + z^2} \tag{6.7}$$

$$M(z) = \frac{0.0000265 + 0.0000531z + 0.0000265z^2}{0.9793820 - 1.9792758z + z^2} \tag{6.8}$$

In order to ease the disposal of processing tasks through static filters, as is the case with Equation (6.7) and Equation (6.8), in Magnus a new class named GenericFilter has been produced.

Once the adequate controller is declared (instantiated with the Generic-Filter class), a step signal is created with the present and the future cut-off frequencies. Then this signal is convoluted through the controller, and from the resulting output signal the final frequency transition vector is obtained.

As mentioned before, this vector corresponds to a smooth frequency transition, the values it contains represent instant discrete cut-off frequencies. In order to apply them to the filter they need to be transformed into the α parameter required by the All Pass Filter (APF), which is then used to create the high-pass shelving filter. Equation (6.9) presents the map between the cut-off frequencies and their corresponding α values, bearing in mind that $w_c = \frac{2\pi f_c}{16000}$.

$$\alpha = \frac{1 - sin(w_c)}{cos(w_c)} \tag{6.9}$$

As it can be derived from Equation (6.9) there's a discontinuity at 4KHz; α tends to ∞ or $-\infty$ depending on the sign taken for the solution of the 2nd degree equation. That's why it has been taken 3.9KHz as the upper bound cut-off frequency limit allowed for the tunable filter to acquire. Although this value may be arbitrarily taken, it is close enough to accomplish its goal, and as it can be seen in Figure 6.3 the whole frequential range is kept very linear.

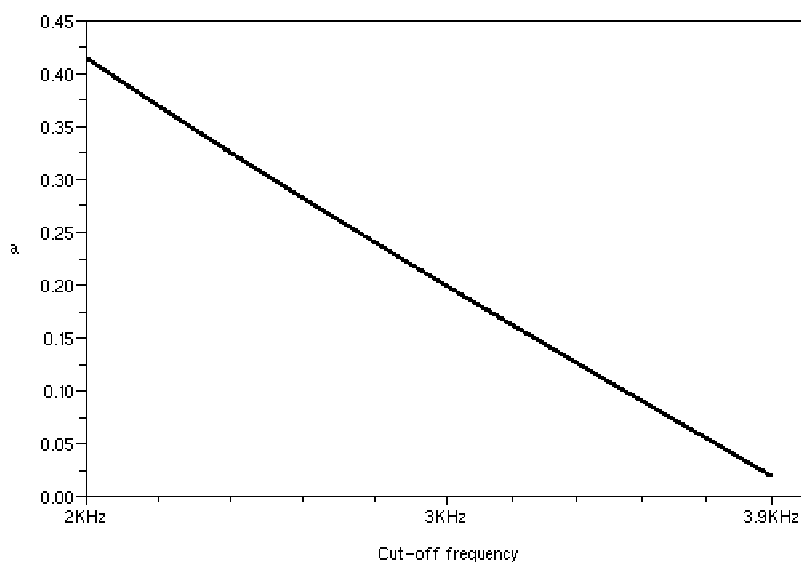

Figure 6.3: Map between the cut-off frequency and the α parameter.

In the end the vector of α values is obtained. These values correspond to the instant cut-off frequencies for the temporal length of the frame in question.

6.2.3 High-pass shelving filter

At this point all the necessary elements to produce the high-pass shelving filter are ready. In order to enable the filter to process the frames with the time-varying α parameter the inverse Z transform (using equations in

differences) needs to be applied to the proposed filter, thus yielding Equation (6.10) and Equation (6.11) as the resulting filter expressed in the discrete time domain.

$$\begin{aligned} A[n] &= -G_\pi \alpha[n] - G_\pi - G_0 \alpha[n] + G_0 \\ B[n] &= G_\pi \alpha[n] + G_\pi - G_0 \alpha[n] + G_0 \end{aligned} \tag{6.10}$$

$$y[n] = \alpha[n]\ y[n-1] + \frac{B[n]}{2}\ x[n] + \frac{A[n]}{2}\ x[n-1] \tag{6.11}$$

Finally, Equation (6.11) can be computed iteratively through all the elements of the incoming frames (input speech) to produce the outcoming processed frames (output speech) as shown in Figure 6.1.

6.3 Excess boost reducer

The second sub-module of the speech enhancement process consists of a 6KHz second-order low-pass filter. The aim of this filter is the reduction of the excess boost produced by the previous part, the high-pass shelving filter. Since the speech signals don't contain much information within the high part of its spectrum, the cut-off frequency of this filter is fixed at 6KHz. Moreover, the acoustic models used in the speech recognition engine have been created with samples previously filtered at about 6KHz. It would not make sense to be using different acoustic features.

According to the Theory Part (see Chapter 4) this filter should be a shelving second-order low-pass filter, thus a Butterworth design has been taken for its implementation. Equation (6.12) presents the transfer function of this filter in the discrete domain.

$$HLPF(z) = \frac{0.3423570 + 0.6847139z + 0.3423570z^2}{0.1780545 + 0.1913733z + z^2} \tag{6.12}$$

In Magnus this component can be identified under the ButterLP class name and it has no configurable properties. It has been implemented using an instance of the GenericFilter class mentioned in the previous section.

6.4 De-esser

The last sub-module of the speech enhancement unit is the de-esser, identified under the DeEsser class name. Figure 6.4 shows the structure of the de-esser proposed in this thesis.

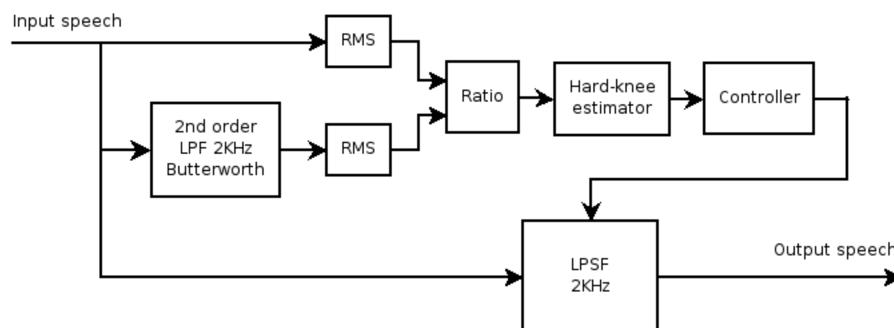

Figure 6.4: De-esser structure.

The goal of a de-esser is to mitigate the excess of high frequency power produced by a high frequency signal. Typically this effect takes place with the presence of loud "ess" sounds produced by fricative phonemes. These loud annoying sounds are called "sibilants". A sound is considered sibilant when its appearance causes discomfort to the listener. Since the first submodule of the speech enhancement unit outputs a gain at high frequencies these sibilant sounds are likely to appear, and thus, they must be treated.

Figure 6.4 provides a set of elements that detect the presence of the sibilants and act in consequence by reducing the gain at high frequencies of the low-pass shelving filter. The processing pipeline that deals with the sibilants and provides a feedback is called a *side-chain* .

6.4.1 Side-chain

The side-chain proposed in the de-esser is composed of three stages: the determination of the ratio between the signal level and the level of the lower part of the signal spectrum, the estimation of the attenuation with a hard-knee compressor and the determination of a smooth transition between the old attenuation values and the new ones.

On the first stage, the lower part of the spectrum of the signal is separated with a second-order low-pass filter created with the Butterworth design, thus resulting in Equation (6.13).

$$HLPF(z) = \frac{0.3423570 + 0.6847139z + 0.3423570z^2}{0.1780545 + 0.1913733z + z^2} \tag{6.13}$$

Then the levels of the input signal and the filtered signal are obtained through the computation of the RMS of the signals. Afterwards the ratio r between these two values is obtained as is expressed in Equation (6.14).

$$r = \frac{RMS(input_signal)}{RMS(filtered_signal)} \tag{6.14}$$

On the second stage, the attenuation is estimated with a *hard-knee* compressor. This artifact responds with an abrupt function, defined in different ranges, as is shown in Figure 6.5.

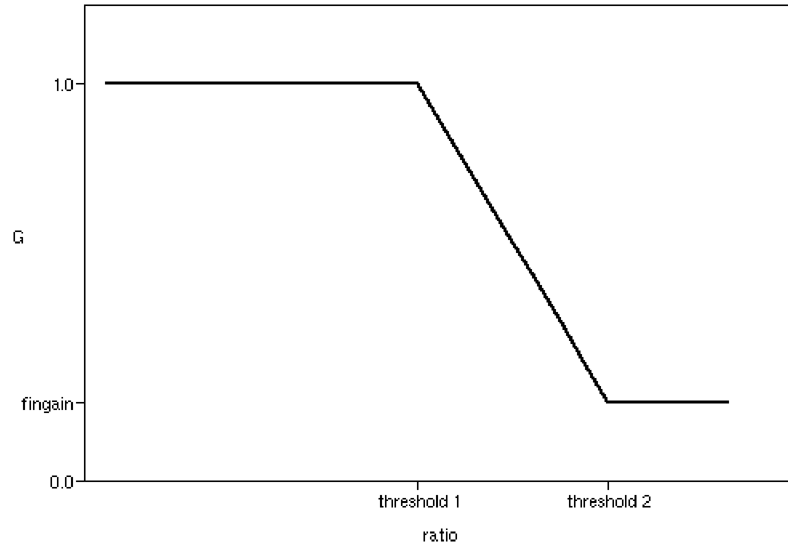

Figure 6.5: Hard-knee compressor function.

In Figure 6.5 the values are assigned through Sphinx-4 properties, whether via the properties sheet or via the default values. The DeEsser class accepts the following properties:

- PROP_DE_TH1 for "threshold1". If the ratio is kept smaller than this threshold the de-esser is transparent. Its default value is set to 1.5.

- PROP_DE_TH2 for "threshold2". If the ratio is bigger than the previous threshold but still smaller than this one, the hard-knee assigns a new gain value according to the predefined attenuation function. Its default value is set to 3.

- PROP_DE_SLOPE for "slope". This is the slope of the linear attenuation function. Its default value is set to -0.67.

- PROP_DE_CNT for "constant". This corresponds to the independent term of the linear attenuation function. Its default value is set to 2.

- PROP_DE_FINGAIN for "fingain". This property represents the final gain that the compressor sets if the ratio exceeds "threshold2". Its default value is set to 0.

On the third stage, a step function is constructed with the present and the new estimated attenuations. Then a controller $MSC(z)$ with two conjugate poles is created, defined with $\tau = 6ms$ and $tr = 10ms$, thus resulting in Equation (6.15).

$$MSC(z) = \frac{0.0000594 + 0.0001189z + 0.0000594z^2}{0.9793827 - 1.9791449z + z^2} \tag{6.15}$$

Finally the vector of attenuations as a function of discrete time is obtained from the result of the convolution between the step signal and the side-chain controller, a process again carried out by an instance of a GenericFilter. This vector is then applied to the real-time filter process of the low-pass shelving filter.

6.4.2 Low-pass shelving filter

This filter is obtained from the design proposed in the high-pass shelving filter of the first speech enhancement module. Since the gains at low and high

frequencies are configurable, it has been observed that instead of assigning $G_0 < G_\pi$ and thus obtaining a high-pass filter, if the inequalty is inversed the resulting filter is low-pass.

Then by fixing the α parameter to 2KHz as explained in due section and leaving G_π as a function of the attenuations vector, Equation (6.16) and Equation (6.17) can be determined to perform the de-essing process in real-time.

$$\begin{aligned} A[n] &= -\alpha\ G_\pi[n] - G_\pi[n] - \alpha + 1 \\ B[n] &= \alpha\ G_\pi[n] + G_\pi[n] - \alpha + 1 \end{aligned} \tag{6.16}$$

$$y[n] = \alpha\ y[n-1] + \frac{B[n]}{2}\ x[n] + \frac{A[n]}{2}\ x[n-1] \tag{6.17}$$

6.5 Speech enhancement results

With Scilab, two environments are shown through a simulation, both with satisfactory results. The first one, shown in Figure 6.6, presents a speech utterance with a sibilant. As it can be seen, the middle-higher part of the spectrum is raised compared to the original spectrum. In this situation the most important element of the speech enhancement chain is the tunable high-pass shelving filter.

The second environment presents the system's response with a sibilant chunk of speech. Figure 6.6 also displays this situation, thus showing that different enhancement effects can take place in the same speech frame. At the highest part of the spectrum (above $\frac{2\pi}{3}$ which corresponds to 6KHz) the enhanced speech is almost inexistent. This is due mainly to the effect of the excess boost reducer.

Finally, Figure 6.7 shows the signals as function of discrete time, highlighting the results offered by the excess boost reducer and the de-esser submodules. Since the de-esser varies its transfer function through time, this effect is shown as not all the speech chunks are affected by the same amount of attenuation.

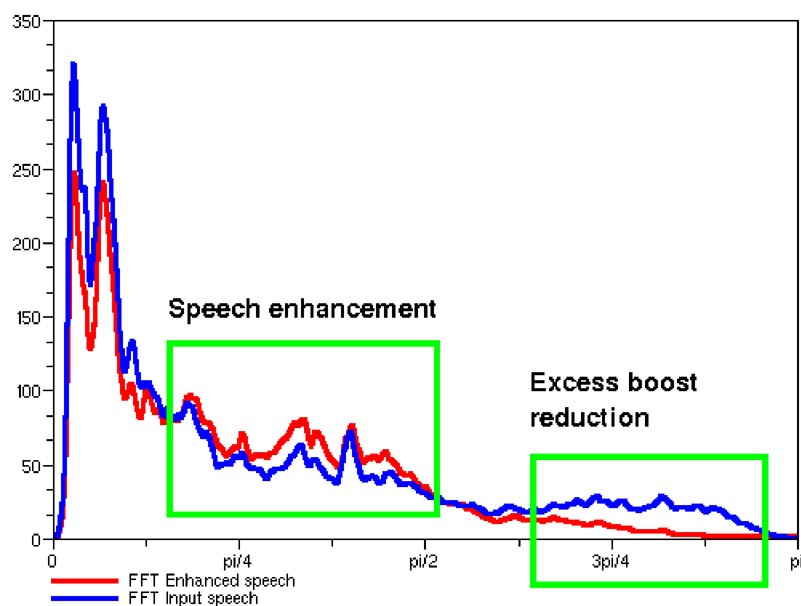

Figure 6.6: Spectral plot of an enhanced speech utterance with a sibilant.

By showing these two different environments the system proves to perform successfully at increasing the presence of the consonants in a speech signal. The system also proves not to be doing so blindly, because if the input signal is sibilant, the system responds with a decrease (of such gain) instead of an increase.

In the development package of these speech enhancement modules there are some other audio samples, prepared with Audacity, with other disturbing effects (echo, bass boost, amplifier tone, chirp, ...) that can be taken as examples to test the peformance of the speech enhancement modules proposed. They are available in the "scilabworks/audio" folder.

In order to produce the disturbing effects Audacity has been fed with a collection of LADSPA plugins provided by Steve Harris (Steve Harris' LADSPA Plugins) and Tom Szilagyi (TAP-plugins). Extensive information about general LADSPA plugins and their plugins in particular can be found in their

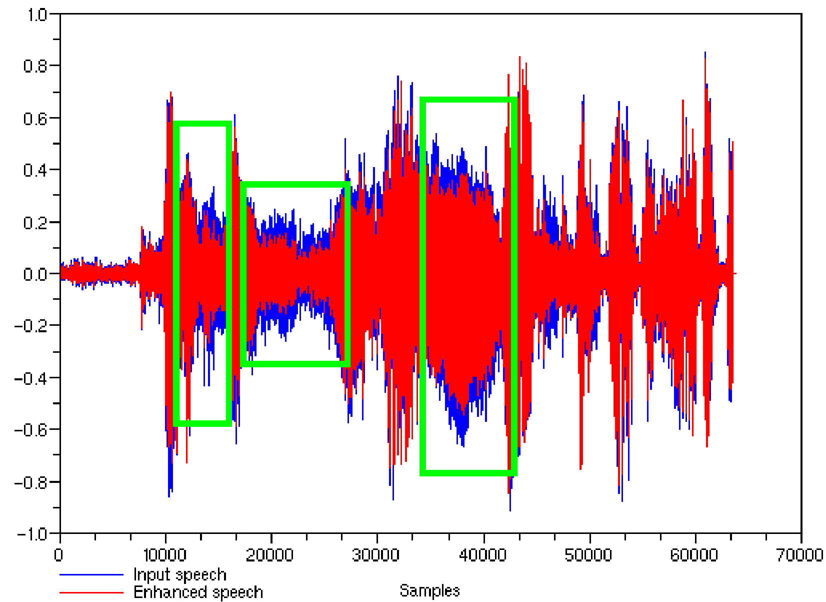

Figure 6.7: Temporal plot of a sibilant speech utterance. The most sibilant parts of the chunk have been highlighted.

respective project's websites:

- Steve Harris' LADSPA Plugins: `http://plugin.org.uk/ladspa-swh/docs/ladspa-swh.html`
- TAP-plugins: `http://tap-plugins.sourceforge.net/`

Since the goal of the speech enhancement unit has already been proven to be working correctly, the rest of these annoying effects have not been included in the thesis.

Chapter 7

Regression Tests

This chapter explains the setting up of the regression tests built for Magnus in order to quantize its improvements with respect to Sphinx-4, the original speech recognition engine upon Magnus has been built. The test data has been obtained at the Department of Education of the Government of Catalonia.

7.1 Setting up a regression test

The regression tests are run in "batch mode", which means that the recognition process does not take place in real time, getting the input speech from a microphone. In this mode the speech signals are previously recorded in audio files and then the appropriate commands are launched in order to obtain the results of the regressions.

Each batch mode regression test consists of the following components:

- Test data: the audio or cepstral data to perform the test on.
- Batch file: this text file lists the location of all the test files, as well as their transcriptions.
- Acoustic models and dictionary.
- Configuration file: specifies the configuration of the system used to test the data.

- Grammar file: this is a BNF-style grammar file (JSGF).
- Batch-mode Recognizer: this is the Sphinx-4 batch-mode recognizer.

All the files and information described in this chapter are available under the "test" folder name which can be found in the source code distribution of Magnus.

7.1.1 Input audio files

Sphinx-4 can handle only raw data. The audio defaults to 2 bytes/sample, at 16000 samples per second. The files are expected to be raw binaries without header. The Java platform assumes big endian order, always. Although these parameters can be changed, they are maintained for all speech files.

The audio files have been recorded with Audacity, a free audio editor and recorder. More information about this program is available at: `http://audacity.sourceforge.net/`. As stated before, the sampling frequency has been kept at 16KHz and the data samples at 16 bits PCM, little-endian, that is the default WAV file format as the uncompressed export format. PCM refers to Pulse-Code Modulation, which is a digital representation of an analog signal where the magnitude of the signal is sampled and quantized regularly at uniform intervals.

In order to obtain the desired audio format files for the regression tests, a free sound tool named Sox has been used. Please check the project's website for more information about Sox at: `http://sox.sourceforge.net/`. To convert the WAV audio file format into RAW data, while exchanging the order of the samples' bytes, the following command has been run on the command line:

```
sox <filename>.wav -x <filename>.raw
```

7.1.2 Batch file

A batch file is a text file that contains the list of files to be processed, with the transcription for each file. One line for each file, where the first element in a line is the file name, which can be an absolute or relative path, and

includes the file extension. And after the file name, the words that make up the transcription for the audio. Sphinx-4 uses the transcription provided here to compute the system's accuracy after each sentence is processed.

An utterance's processing produces a hypothesis, which is then compared to the transcription and a summary of the results is reported. Aspects like the alignment, the extra insertions and the missing ones are treated in this analysis.

The various batch files written for magnus are available under the "test/batchfiles" folder. The file named "deptedu.batch" contains all the original sound files recorded at the Department of Education, while the files named "deptedu.malevoices.batch" and "deptedu.femalevoices.batch" separate the male speakers from the female speakers. This distinction is important because male and female voices have different signal properties, and thus, the speech recognition system may not respond in the same way for the two signal types.

7.1.3 Acoustic model and dictionary

The acoustic model used for the regressions corresponds to the Wall Street Journal (WSJ) produced by the build process of Sphinx-4. This model has been created with recordings from this journal by the developers of Sphinx-4 at Carnegie Mellon University, so the phonemes trained belong to the American Engish phonetic language. The name of the model is "WSJ_8gau_13dCep_16k_40mel_130Hz_6800Hz".

The dictionary has been prepared specifically to recognize Catalan commands. Although the acoustic models defined above are created using a different language, the phonemes can be regrouped to form the desired words. The file that contains this information is named "magnus.dict" and can be found under the "cfg" folder in the source code distribution of Magnus. As a means of exposing the various commands available (so far), the file is shown as follows:

```
<sil>           SIL          <silenci>
CLICK           K L IH K     clic
```

```
CLICK(2)        K L IY K          clic(2)
STOP            S IH              si!
STOP(2)         S IY              si!(2)
LEFT            AH S K EH R AH    esquerra
LEFT(2)         AH S K EH R EH    esquerra(2)
RIGHT           D R EH T AE       dreta
RIGHT(2)        D R EH T ER       dreta(2)
UP              AH M UW UW N      amunt
DOWN            AH B AA AY        avall
DOWN(2)         AH V AA AY        avall(2)
MENU            M AE N UW         menu
MENU(2)         M AA N UW         menu(2)
DOUBLE          D AO P L AH       doble
DOUBLE(2)       D AO P L AE       doble(2)
DRAG            AH R UW S EH G AE arrossega
DRAG(2)         AH R UW S EH G AH arrossega(2)
KEYBOARD        T AH K L AH T     teclat
KEYBOARD(2)     T AH K L AE T     teclat(2)
MOUSE           R AH T UW L IY    ratoli
MOUSE(2)        R AH T UW L IH    ratoli
```

As it is shown above, the file maintains the configuration established by Sphinx-4 in order to interact with the FullDictionary class of the Linguist. According to the future objective of the project to bring the program to other languages, a goal that is sometimes referred to as "internationalization", Magnus works all the time with English commands but presents them in the corresponding language, so it translates them before it shows them, but internally, as a means of simplicity, the language used is English.

7.1.4 Configuration file

As usual, the configuration files describes the elements that have a role in the recognition process. When it comes to batch mode recognition, the elements are not the same as live mode recognition.

In the "test/configfiles" folder of the source code distribution of Magnus there are several configuration files that correspond to the proposed regres-

sion tests. The files named "deptedu.config.nose.<wsj|rm1>.xml" refer to the tests that do not incorporate the extra speech enhancement sub-modules, but the files named "deptedu.config.se.<wsj|rm1>.xml" incorporate them. These two types of configuration files are used to quantize the improvement, if there is some, in the speech enhancement process proposed.

Bear in mind that the Sphinx-4 configuration files shall provide a static folder path for the location of the grammar and static file paths for the location of the dictionary and the filler. In order to ease the setting up of these paths, at the beginning of each test configuration file there are three root properties that define the location of the mentioned elements.

The only difference between these two types of files is the presence of the three speech enhancement artifacts desribed in the previous chapter: the HPSFilter, the ButterLP and the DeEsser. The instrumentation tools are the same, and they are described in the following section.

7.1.5 Grammar file

Sphinx-4 uses the Java Speech API Grammar Format (JSGF) to perform speech recognition using a BNF-style grammar. JSGF grammars are used with the FlatLinguist Sphinx-4 class. The features of JSGF include:

- Using other grammar rules within a grammar rule.
- The OR "|" (pipe) operator.
- The grouping "(...)" operator.
- The optional grouping "[...]" operator.
- The zero-or-many "*" (Kleene star) operator.
- An associated probability.

Under the "cfg" folder name in the source code distribution of Magnus there can be found the file named "magnus.gram". This text file, written with the JSGF specification, holds the specific grammar structure for Magnus.

One of the most simple and intuitive structures of grammars are the typical command and control structures, like the one that shows the grammar file of Magnus:

```
#JSGF V1.0;

/**
  * JSGF MAGNUS Grammar in Catalan
  */

  grammar magnus;

  public <tokens> = ( LEFT | RIGHT | UP | DOWN | STOP | CLICK |
                     MENU | DOUBLE | DRAG | KEYBOARD | MOUSE )*;
```

The command and control structures with no special features like the ones described above (groupings, probabilities...) result in very simple and naive grammar graphs, which yield fast recognition results at the expense of poorer accuracy results. It's a matter of finding the optimum tradeoff between these two features for the desired application.

7.1.6 Batch-mode recognizer

This Sphinx-4 class decodes a batch file containing a list of audio files to decode. The audio data should be 16-bit big-endian (by default), 16KHz and PCM-linear data. This is the class that is launched when running the regression tests. In the "test" folder there is a simple shell script (written in bash) to launch the tests with the appropriate commands in order to be in concordance with the folders structure of the source code organization. The name of the file is "deptedu.test.sh". This script accepts two parameters, sets the correct classpath and then launches the basic command:

```
java BatchModeRecognizer <xmlConfigFile> <batchFile>
```

Of course, the two parameters that the script accepts are the configuration file and the batch file.

7.2 Instrumentation tools

Sphinx-4 can be configured to output various collections of information that may be useful for developers (mainly), which includes warning and error messages, tracing messages, recognition results, *et cetera*. The output of the various types of instrumentation information is controllable from the configuration file.

The following subsections describe in detail the serveral instruments used to track this information.

7.2.1 Logger

Well behaved Sphinx-4 components output informational messages via the Sphinx-4 logger. These have a level of importance associated with them. Some messages indicate severe problems, some messages are warnings, some are informational, and some are fine level tracing messages. The complete set of log levels are:

- SEVERE (highest value): an error occurs that makes continuing the operation difficult or impossible.
- WARNING: an anomalie has occured, but the operation is continuing.
- INFO: general information.
- CONFIG: information about a components configuration.
- FINE: tracing messages.
- FINER: finer grained tracing messages (lots of output).
- FINEST (lowest value): finest grained tracing messages (huge amounts of output).

These levels of information are configured through the value (indentified above with capital letters) of the "logLevel" property, which can be associated to the root level at the XML configuration hierarchy thus establishing the default configuration of the components that contains, or otherwise can be associated to each component separately, thus allowing different components to have different configurations.

The logLevel configured at the "cfg/magnus.config.xml" file is WARNING while at the "test/configfiles/deptedu.config.<nose|se>.wsj.xml" is INFO. This denotes the need of information required when running a regression test.

Once the logging output is set to INFO at the root level of the test configuration files, the recognizer has to be linked to the instrumentation tools in order to be able to output the different aspects that can be tracked. These tools have to be itemized in the recognizer's propertylist named "monitors". These monitors are detailed in the following subsections.

7.2.2 Accuracy tracker

One of the prime methods of measuring the overall quality of a speech recognition system is the recognition accuracy. This statistic shows how well the sentence hypotheses produced by the recognizer match the actual transcripts of what was spoken. Obviously, recognition accuracy can only be reported when the transcripts are available as well. This last sentence responds for the structure of the batch files.

The component in charge of tracking the accuracy performance is the "accuracyTracker" class. This component has two properties sets: the "showAlignedResults" property, which displays the alignment of the recognition results, and the "showRawResults" property, which shows the results before being altered by any other process.

With this configuration of the accuracy tracker lots of information is provided for every audio file listed in the batch file. The following itemization presents all the various qualities that could be analysed by this component:

- REF (reference): The reference or transcript. This is what should be recognized.
- HYP (hypothesis): The result that is generated by the recognizer. This is what was recognized.
- ALIGN_REF (aligned reference): The reference text, where mismatches between the reference and the hypothesis are highlighted.

- ALIGN_HYP (aligned hypothesis): The recognized text with mismatched text highlighted.
- RAW (raw text): The actual text recognized, including all filler words such as silences, coughs, lip smacks, breaths and so on.
- Accuracy (word accuracy): The number of matching words compared to the total number of words in the input as a percent.
- Errors (word error count): The total number of word errors.
- Sub (substitution count): The total number of substitution errors. A substitution error occurs when one word is replaced by another.
- Ins (insertion count): The total number of insertion errors. An insertion error occurs when an extra word is inserted in the hypothesis.
- Del (deletion count): The total number of deletion errors. A deletion error occurs when a word is missing in the hypothesis.
- Words (reference word count): The total number of words expected.
- Matches (matching word count): The total number of matching words.
- WER (Word Error Rate): This is equal to the percentage of the ratio between the sum of the substitutions plus insertions plus deletions and the total number of words expected.
- Sentences (reference sentence count): The total number of sentences.
- SentenceAcc (sentence accuracy): This is equal to the percentage of the ratio between the total number of matching words and the total number of sentences.

The accuracy tracker will also show summary statistics information at the end of a run (when the recognizer is deallocated).

7.2.3 Speed tracker

Another important aspect of speech recognition is the speed of recognition. The speed tracker will track and report statistics related to the speed of recognition. The speed tracker is added to the set of monitors in the recognizer, in the same way that the accuracy tracker is added, with the class name of "speedTracker".

The speed tracker is setted with three properties: the "showSummary", "showDetails" and "showResponseTime". They all have self explainable names. The data output by the speed tracker are itemized below:

- This time audio: The length of time (in seconds) of the current audio.
- This time proc: The time spent processing this audio.
- This Speed: The ratio between the processing time and the audio time.
- Total time audio: The time for all audio.
- Total processing: The time spent processing all audio.
- Total Speed: The ratio between the total processing time and the total audio time.
- Response Time: The average (Avg), maximum (Max) and minimum (Min) response times encountered.

Response times are useful when running in a live-mode situation where front-end buffering latency can affect the perceived performance of the system. These are the times from when the front-end first encounters a packet of audio, until it is delivered to the decoding portion of the recognizer.

The speed tracker can also be configured to dump out low level timing data for various aspects of the recognition process as many of the components in the Sphinx-4 system will collect detailed timing statistics. This part, though, is not treated as the goal of the thesis is not to provide a faster system, but a more accurate and reliable one.

7.2.4 Memory tracker

For some applications, and Magnus is among them, the overall memory footprint of the recognizer is important. The MemoryTracker class is used to track the memory usage of Sphinx-4. It is added to the set of monitors in the recognizer in the same way as the other instrumentation components.

The memory tracker ouputs five data items listed in the following itemization:

- Mem Total: The total amount of memory allocated to the JVM.
- Free: Of the Mem Total how much is currently not being used.
- Used This: How much memory is currently being used.
- Used Avg: The average amount of memory used.
- Used Max: The maximum amount of memory used.

Chapter 8

Speech Recognition Results

This chapter deals with the preparation of the regression tests, their performance and the results obtained from them, thus evaluating the quality of the speech recognition application.

8.1 Word Error Rate

This is the rate used to evaluate the quality of the system. It is defined as Equation (8.1), as stated in the previous chapter.

$$WER = \frac{substitution_errors + deletion_errors + insertion_errors}{total_number_of_words_expected} \quad (8.1)$$

The reader may be enticed to think of the WER as a complementary rate of the Accuracy. This is not actually true, despite the complementary rate of the Accuracy may be close to the actual WER sometimes. But the WER is a more restrictive rate. While the Accuracy could be higher than 90%, the WER could still be higher than the 10% that remains. The smaller the WER, the better the system.

The recognized word sequence can have a different length from the reference word sequence (which is supposed to be the correct one). Thus the WER is derived from the Levenshtein distance [Woltzenlogel, 2007], working at the word level instead of the phoneme level. The Levenshtein distance is a metric for measuring the amount of difference between two sequences, say

two strings. This distance is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single word in the case that concerns this thesis (originally the operations deal with single characters).

Then the problem of the difference in length between the recognized word sequence and the original one is solved by first aligning the recognized word sequence with the reference (spoken) word sequence using a dynamic string alignment. The WER can then be computed as stated in Equation (8.1).

8.2 Audio files workbench

As is stated in [Shankar Chanda and Park, 2007] the speech enhancement module provides an improvement of the intelligibility of far-end clean speech signal to a listener who is located in such environment. Then, since the listener is far away from the point of signal origin, say the speaker, the produced speech is prone to be modified by the environment. This fact gives place for the determination of which environments are more adequate for the speech enhancement process proposed.

It could be said that it is interesting to know in which environments the speech enhancement process provides better speech recognition results compared to the original system without such speech enhancement process.

The speech signals obtained for the regression tests have been recorded at the Department of Education in an acoustically insulated room, thus the quality of the recordings is supposed to be more than acceptable. Then in order to bring the speech enhancement process goal to success, these recorded speeches need to be modified to emulate the environment's acoustic pollution.

The basic tools chosen to do the speech tracks modifications are Sox and Audacity. Let's first define the basic format used in Sox to treat the RAW audio files:

```
<format> = -t raw -r 16000 -s -w -x
```

This syntax is only valid in this thesis. The whole parameters chain must be typed when running the commands.

Table 8.1 presents all the audio files that build the workbench with its corresponding time length. They have all been recorded at the Department of Education. These files are available under the "test/soundfiles" folder.

Table 8.1: Audio files workbench

Track name	Time length(sec)
alex.raw	28.32
anna.raw	29.94
enric.raw	22.72
francesc.raw	23.94
jaume.raw	38.93
joan.raw	42.45
laura.raw	26.02
manel.raw	25.42
marc.raw	34.51
pablo.raw	27.63
roberto.raw	39.34
sandra.raw	38.19
sara.raw	28.56
silvia.raw	23.97

One thing should be noted: since the tack named "joan.raw" is the longest one (42.45 seconds), its length is taken for reference for all the noise files that will be produced.

The white and pink noises have been produced with Sox using the following command:

```
sox <format> joan.raw <format> -v 0.01 <wn|pn>01.raw
    synth <whitenoise|pinknoise>
```

Obviously "wn" is related to "whitenoise" and "pn" is related to "pinknoise".

The rest of the noises (blue, red, violet, grey and babble) have been produced with a combination of Audacity and Sox, because the generation of these noises is not directly available through Sox. The original noise samples are obtained from the Wikipedia, then after checking that the noises have a correct frequential distribution, they have to be downsampled with factor 3 with Audacity as the original files are sampled at 48KHz. Afterwards, the dynamic range of the signals is set from -1 to 1 and finally the tracks are exported as WAV.

Then Sox can convert the generated WAV file (little-endian) into a RAW file (big-endian) while adjusting the volume of the signal at the same time with the command:

```
sox noise.wav <format> -v 0.01 noise.raw
```

Finally, the whole set of noises is obtained, with the desired amplitude value of 0.01 in both cases: the noises produced directly by Sox and the noises downloaded from the Internet. This 0.01 value refers to the volume (amplitude) of the signals. It has been chosen empirically in order to obtain a tractable amount of noise. If the noise volume exceeds this volume, the results are so bad that no analysis can be performed on them.

In the last step to complete the creation of the workbench, the noises have to be mixed with the speech signals in order to emulate the different environments of study. This step is carried out with Sox with the command:

```
soxmix <format> speechfile.raw <format> noise.raw
       <format> noisyspeechfile.raw
```

The reader should note the big advantage in using a command-line program like Sox instead of any other sound editor or Digital Audio Workstation (DAW) with a GUI: Sox can be included in a shell script. For example, when having to apply the same signal process to a list of files, *i.e.* the files of the workbench, it's far easier to do it with a single instruction than

having to edit each file separately. This is achieved with the shell instruction:

```
for i in *.raw; do <command>; done
```

The argument `<command>` can refer to any of the commands shown above. Note that the RAW files in the one-line script are referred to as `$i`.

8.3 Regression tests results using the Wall Street Journal acoustic models

This section is dedicated entirely to the results and the conclusions that can be extracted from the conducted regression tests using the Wall Street Journal (WSJ) acoustic models included in the distributions of Sphinx-4. These acoustic models have been trained with the database CSR-I (WSJ0) Complete, authored by John Garofalo, David Graff, Doug Paul and David Pallett as it is shown in the website of the Linguistic Data Consortium (LDC). The database consists primarily of read speech with texts drawn from a machine-readable corpus of Wall Street Journal news text. The texts to be read were selected to fall within either a 5000-word or a 20000-word subset of the WSJ text corpus, so the database is rather abundant. The WSJ acoustic models obtained are used by default in the distributions of Sphinx-4.

The different environments proposed in this section are tested sequentially, following a certain order and taking into account the previous results in the conclusions. In order to run the tests, once the speech files have been created, the script named “test/deptedu.test.sh” has to be run with two arguments: the configuration file and the batch file.

This thesis presents three perspectives of study: the presence/abscence of sources of noise, the gender of the speakers and the use of the speech enhancement unit. These three variables determine the several environments considered, to be then compared according to the resuting WER so as to extract the pertinent conclusions.

The summary statistics dumped by Sphinx-4 are available in the “test/results” folder.

8.3.1 Acoustic insulated environment

Due to the environment characteristics, the summary statistics should yield the minimum WER possible since the speech tracks are the ones with the best acoustic quality. The recordings have been performed in an acoustically insulated room, thus the speech sound files contain the most clean speech signals. All the rest of the signals (the ones that emulate a predefined environment) have been created from these ones. Table 8.2 shows the obtained results.

Table 8.2: Recognition results obtained in an acoustic insulated environment

Gender	Process	WER(%)	Speed($\times$RT)	Memory(MB)
Male	No SE	15.94	0.69	63.13
	SE	18.36	0.73	63.15
Female	No SE	22.61	0.70	63.16
	SE	27.83	0.73	63.12

[1]Note: SE and RT stand for Speech Enhancement and Real Time, respectively.

As it can be seen in Table 8.2 the resulting WER is not very low (compared to what would be desirable) despite of the quality of the speech recordings. This can be attributed to the fact that the majority of the speakers had no previous training at all with the system. In fact, in only one speech track (out of fourteen) the speaker had previous experience with Magnus. This track corresponds to the system's developer.

As it can be observed for both genders, the speech enhancement unit worsens a little bit the system's performance. The speech enhancement unit is supposed to do its job when there's noise pollution mixed with the speech signal. So, if the environmental conditions differ (in this case the noise pollution is absent) it makes sense that the enhancement unit fails to do what it is supposed to do, or even impoverish the system's performance as is the case.

Here a noise pollution sensor should be included in order to detect when is it worth to turn on the speech unit system and when this enhancement unit will worsen the performance of the main system.

One last observation: the higher WER in the female voice recordings. This may be due to the set of acoustic models used to perform the recognitions: the WSJ models. As stated in [Adbulla and Kasabov, 2001], in speaker independent speech recognition system, as Magnus aims to be, the speaker variability is rather undesirable. This responds to the vast number of training speech samples required to build accurate acoustic models. Male and female speeches differ considerably in the average pitch frequency of the speaker. That's why a gender dependent HMM for each word is proposed, with significant improvement in word recognition accuracy. Please refer to the cited article for further details. Sphinx-4 does not provide such gender separation, so recognition accuracy differences are likely to appear.

The four summary statistics files dumped by Sphinx-4 are available with the "originals" particle prepended to the files.

8.3.2 White noise polluted environment

In order to obtain a first approximation of the effects that the noise added to the recorded signals can produce over the recognition performance of Magnus, a white noise is proposed. White noise is characterized for having a uniform distribution over the frequential axis — a flat frequency spectrum in linear space. Then, both the higher and lower parts of the signals are equally polluted. White noise is kind of "bright" and not terribly relaxing, but is very effective for masking other sounds.

The maximum peak amplitude of the noise signal is set to 0.01. Figure 8.1 shows the frequential plot of the white noise produced for the tests.

With this environment the speech recognition results are not expected to be as good as the results obtained with the clean speech recordings because of the noise pollution added to the whole spectrum uniformally. Table 8.3 shows the performance summary statistics obtained.

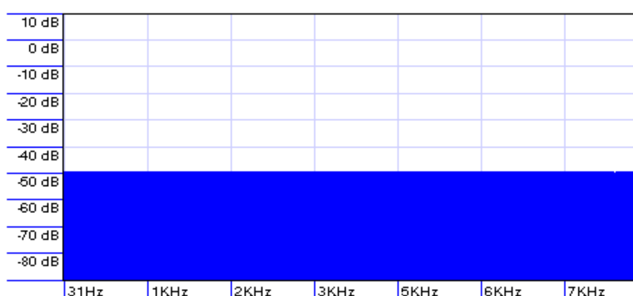

Figure 8.1: Frequential analysis of white noise

Table 8.3: Recognition results obtained in a white noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	21.74	0.79	63.14
	SE	22.71	0.80	63.21
Female	No SE	40.87	0.83	63.13
	SE	40.00	0.84	63.17

As it can be been in Table 8.3, with respect to the clean speech environment the WER has clearly grown for both genders, being the female results almost twice as the male results. This ratio did not exist in the orignal environment.

For the male voices both the non enhanced speech results and the enhanced ones have incremented their WER in 6 points approximately, being the enhanced results a point higher than the non enhanced results.

For the female voices, the enhanced results are slightly lower than the non enhanced results, but they both almost double the male voices results. The complete results are available with the “wn01” particle prepended to the corresponding files in the “test/results” folder.

In summary, the white noise polluted speech samples decrease the performance of the speech recognition system, being the female results much more accentuated than the male ones. Let’s believe that exists a difference

between the noise polluted frequential ranges of the signals. In this environment, both these ranges have been equally affected: the vowels have been polluted as the vowel formant frequencies are located below 2KHz and the consonants have also been polluted as their frequencies range from 2KHz to 6KHz.

The speech enhancement module has had a little impact in this environment. It has contributed to the noise increase in the higher part of the spectrum, thus emulating a kind of blue noise. Since the noise distribution in this part of the spectrum is quite high, the signal to noise ratio is then quite small small. Later on this new color of noise will be analysed, but according to this results, the expectations are not very good.

The following subsections propose different noise environments to note the difference (if there is any) between a high frequency or a low frequency polluted speech recording when it comes to speech recogniton performance.

8.3.3 Blue noise polluted environment

This environment proposes a blue noise pollution that affects mainly the consonants but still has some effect on the vowels. Blue noise is just high-pass filtered white noise. It sounds really screechy and artificial. Blue noise's power density increases 3 dB per octave with increasing frequency (density proportional to f) over a finite frequency range. Having set its maximum peak at 0.01, Figure 8.2 shows the frequential plot of the blue noise produced for the tests. Bear in mind that although the horizontal axis shows frequencies they are rather angular frequencies and should be expressed in radians, but for convenience, the Audacity's notation is maintained.

If the results obtained with it are different (higher or lower) than the results obtained with the white noise polluted environment, the proposed supposition of frequential selectivity of noise pollution will be proved. Table 8.4 shows the performance summary statistics obtained.

As it is expected, the results vary from the previous environment, which proves that noise pollution frequential selectivity affects the speech recognition performance. Moreover, a concentration of the noise energy at the

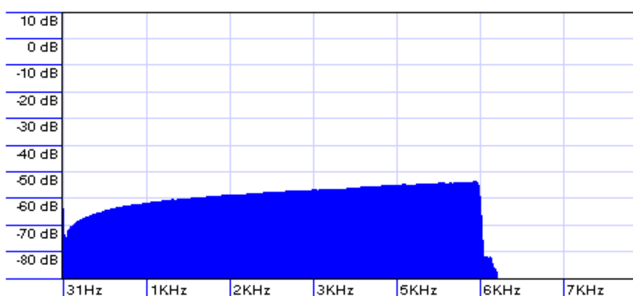

Figure 8.2: Frequential analysis of blue noise

Table 8.4: Recognition results obtained in a blue noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	24.16	0.56	63.15
	SE	24.64	0.82	63.12
Female	No SE	39.13	0.84	63.13
	SE	38.26	0.86	63.15

higher part of the spectrum could induce to state that this part is more sensitive to noise pollution as the results obtained are worse than with an equal frequential noise distribution. This statement has to be compared to its counterpart (the noise concentration at the lower part of the spectrum) to be proven true.

The speech enhancement unit provides slightly better results for female voices but slightly worse results for male voices. This time, the ratio between the gender results is less than the double. Still, the results are unacceptable.

The complete results are available with the “bn01” particle prepended to the corresponding files in the “test/results” folder.

8.3.4 Violet noise polluted environment

This environment proposes a violet noise pollution that affects only the consonants. The energy distribution of this noise pollutes the higher part

of the spectrum. Violet noise's power density increases 6 dB per octave with increasing frequency (density proportional to f^2) over a finite frequency range. Having set its maximum peak at 0.01, Figure 8.3 shows the frequential plot of the violet noise produced for the tests.

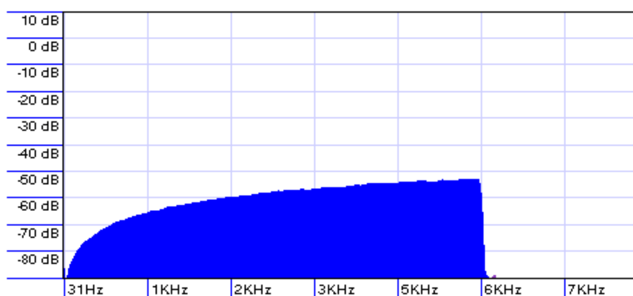

Figure 8.3: Frequential analysis of violet noise

If the WER results obtained with it are even higher than the results obtained with the blue noise polluted environment it will state that the more noise pollution to the higher part of the spectrum the worse the recognition results. Table 8.5 shows the performance summary statistics obtained.

Table 8.5: Recognition results obtained in a violet noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	28.99	0.79	63.13
	SE	29.47	0.83	63.22
Female	No SE	44.35	0.85	63.12
	SE	43.48	0.87	63.12

As it is shown in Table 8.5 the results are even worse: if the noise pollution energy is displaced to the higher part of the spectrum, thus affecting the consonants, the speech recognition results are badly worsened.

The complete results are available with the "vn01" particle prepended to the corresponding files in the "test/results" folder.

8.3.5 Pink noise polluted environment

This environment proposes a pink noise pollution that affects both consonants and vowels, with more presence on the latter. The energy distribution of this noise pollutes mainly the lower part of the spectrum, but still has some effect on the higher part. The frequency spectrum of pink noise is flat in logarithmic space — this noise has equal energy decrease (or increase) per octave (density proportional to f^{-1}). This means that the volume decreases logarithmically with frequency at the rate of 3 dB per octave. Usually pink noise is made by low-pass filtering white noise but sounds more natural than white noise (it sounds like rushing water or ocean surf) and is quite relaxing. It's often used for ambience in electronic music, and as a test signal for "tuning" sound reenforcement systems (many equalizers and audio spectrum analyzers have built-in pink noise generators). Having set its maximum peak at 0.01, Figure 8.4 shows the frequential plot of the pink noise produced for the tests.

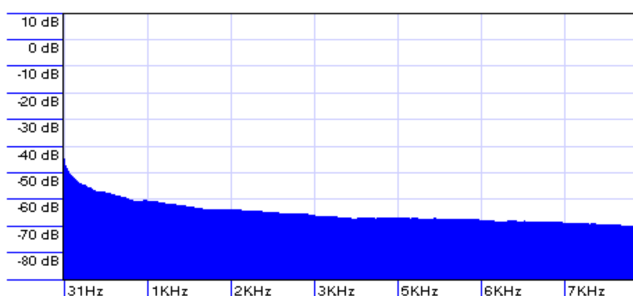

Figure 8.4: Frequential analysis of pink noise

If the results obtained with it are lower than the results obtained with the blue or violet noise polluted environments it will state that the more noise pollution to the lower part of the spectrum the better the results, as this part is more robust. If the results obtained are similar to the blue noise pollution it will state that the system is least degraded by noise pollution when its distribution is non selective with frequency. Table 8.6 shows the performance summary statistics obtained.

As it is shown in Table 8.5 the results are most shocking: not only the WER is lower than in the blue noise pollution environment, it is even lower

Table 8.6: Recognition results obtained in a pink noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	9.66	0.72	63.15
	SE	9.18	0.73	63.12
Female	No SE	18.26	0.66	63.13
	SE	16.52	0.75	63.14

than in the acoustic insulated environment.

In this situation the speech enhancement module provides an improvement in both genders, although in the female the improvement is greater despite of the higher value (0.5% vs. 1.7%).

This is a strange environment, it doesn't make much sense to obtain better results than with the original recordings. One possible situation where the addition of noise produces a positive effect is the "dithering" when quantizing a very low signal, but this is not exaclty the same case. Maybe this WER improvement has something to do with it, or the fact that pink noise is used for tuning reenforcement systems as desribed above.

The complete results are available with the "pn01" particle prepended to the corresponding files in the "test/results" folder.

8.3.6 Red noise polluted environment

This environment proposes a red noise pollution that affects vowels exclusively. The energy distribution of this noise pollutes the lower part of the spectrum. It could be viewed as the counterpart of violet noise. Red noise is a very bassy (heavily low-pass filtered) kind of noise. This sounds like a low rumble — a subway train going by or a noisy air-conditioning system. The definition of red noise is not as precise as that of white and pink noise, and the term mostly refers to low-pitched noises. It has a power density decrease of 6 dB per octave with increasing frequency (density proportional to f^{-2}). Brown noise is also the sound made by a "random walk" which makes the

amplitude of a waveform travel up and down at random. It can be generated by an algorithm which simulates Brownian motion or by integrating white noise. Having set its maximum peak at 0.01 Figure 8.5 shows the frequential plot of the red noise produced for the tests.

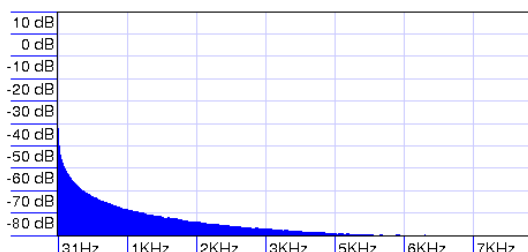

Figure 8.5: Frequential analysis of red noise

It is expected that with this new environment the results may be lower than with the pink noise environment, sticking to the supposition that the lower part of the spectrum is the most robust one and by draining all noise sources the results will be most splendid. Table 8.7 shows the performance summary statistics obtained.

Table 8.7: Recognition results obtained in a red noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	20.29	0.82	63.12
	SE	23.67	0.95	63.15
Female	No SE	35.65	0.82	63.20
	SE	39.13	0.79	63.26

As it is can be seen in Table 8.7 the results do not correspond to the expectations: the results are worse than in the pink noise pollution environment. But anyway, they are not as bad as with the violet noise pollution or even the blue noise pollution (now the comparison can be made), which means that the lower part is indeed more robust that the higher part. But according to this last experiment, the WER reaches a lower bound limit that is reached

by focusing the pollution on the vowels but still leaving some noise for the consonants.

The complete results are available with the “rn01” particle prepended to the corresponding files in the “test/results” folder.

8.3.7 Grey noise polluted environment

This environment proposes a grey noise pollution that affects both vowels and consanants in a special manner. The energy distribution of this noise follows a psychoacoustic equal loudness curve (such as an inverted A-weighting curve) over a given range of frequencies, giving the listener the perception that it is equally loud at all frequencies. Having set its maximum peak at 0.01 Figure 8.6 shows the frequential plot of the grey noise produced for the tests.

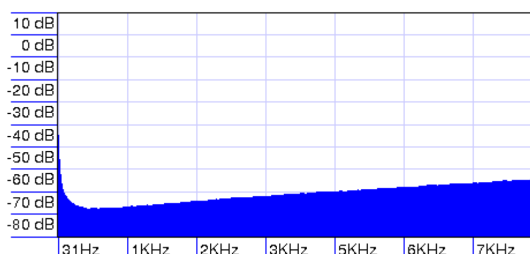

Figure 8.6: Frequential analysis of grey noise

From the previous results it is expected that with this new environment the results may be quite similar to the results obtained for the white noise pollution environment. The decrease in energy of the central part of the spectrum is distributed to the edges thus becoming a distribution 50% similar to the blue noise and 50% similar to the pink noise. Table 8.8 shows the performance summary statistics obtained.

As it is can be seen in Table 8.8 the results are more similar to the ones obtained with the blue noise pollution than with the white noise pollution. Anyway, it’s not surprising that the performance decreases despite of the impovement that the speech enhancement unit introduces in the male voices.

Table 8.8: Recognition results obtained in a grey noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	23.19	0.77	63.16
	SE	21.74	0.86	63.15
Female	No SE	36.52	0.91	63.12
	SE	36.52	0.91	63.32

The complete results are available with the “gn01” particle prepended to the corresponding files in the “test/results” folder.

8.3.8 Babble noise polluted environment

This subsection proposes a babble noise polluted environment obtained from a cafeteria of an IKEA store in Germany. The sound sample has been obtained from “the Freesound Project”, its name is “IKEA Cafeteria.wav” and it has been added by “inchadney” on March 4, 2007, available at `http://www.freesound.org`.

This audio file has required a little more processing than the others. First, it has been converted from stereo to mono, then amplified to use the whole dynamic range, then downsampled and finally changed its speed, thus affecting both Tempo and Pitch using a 175% change. This last step was not necessary for the noise samples, because the downsampling does not change the stochastic properties of the signal. Finally, having set its maximum peak at 0.01, Figure 8.7 shows the frequential plot of the babble noise produced for the tests.

This environemnt is insteresting because it is based on the real world: the sample babble noise used is the clean recording of a cafeteria. Table 8.9 shows the performance summary statistics obtained.

As it is can be seen in Table 8.9 the results are just slightly worse than with the acoustic insulation. They are not still as bad as with the white noise polluted environment, but the speech enhancement unit does not provide

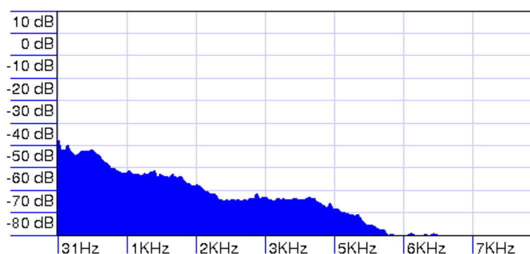

Figure 8.7: Frequential analysis of babble noise

Table 8.9: Recognition results obtained in a babble noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	18.36	0.66	63.15
	SE	18.84	0.85	63.12
Female	No SE	26.09	0.78	63.12
	SE	30.44	0.93	63.12

any improvement compared to the non enhanced speech. The only aspect to highlight is that while the majority of samples have got worse (with respect to the acoustic insulation results), the male enhanced speech has remained almost the same.

The complete results are available with the "babble01" particle prepended to the corresponding files in the "test/results" folder.

8.3.9 Babble noise mixed with pink noise polluted environment

This subsection proposes a noise mixture between babble noise pollution and pink noise pollution. The reason to do it comes from the good results obtained with a pink noise polluted environment. It intends to profit from the stochastic characteristics of pink noise to surpass the problems that arise in other noise polluted environments.

The idea is that a new speech enhancement submodule is produced and set in front of the existing speech enhancement chain. This submodule produces pink noise continuously that is mixed with the incoming speech, which is the babble noise in this environment. The new signal statistic characteristics should provide a performance increase in the speech recognition system, if this systems proves to be working successfully.

Setting the noise mixture's maximum peak at 0.01, Figure 8.8 shows the frequential plot of the babble noise mixed with pink noise produced for the tests.

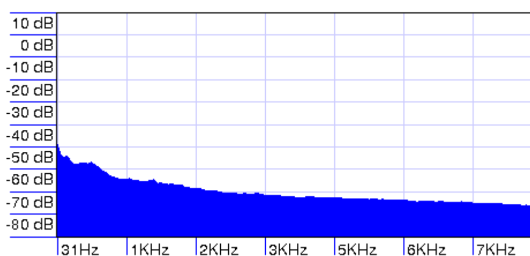

Figure 8.8: Frequential analysis of babble noise mixed with pink noise

As it can be seen in Figure 8.8 the babble noise has almost been completely masked by the pink noise, which has a higher power level. The stochastic nature of pink noise does not change the statistic properties of the speech signals. Thus, the speech recognition system can process the data successfully. As pink noise has yielded the best results, it's logic to believe that the resulting noise distribution (babble + pink) yields better results than with the original noise distribution alone (babble). Table 8.10 shows the performance summary statistics obtained.

As it is can be seen in Table 8.10 the results are just slightly worse than with pink noise alone, being still better than the clean speech signals recognition results.

The addition of the pink noise generator before the existing speech enhancement unit proves to be a valid new enhancement method to improve

Table 8.10: Recognition results obtained in a babble noise mixed with pink noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	11.59	0.73	63.14
	SE	11.59	0.89	63.15
Female	No SE	23.48	0.95	63.12
	SE	22.61	0.93	63.20

the recognition results, at least for this system and in the presence of babble noise.

The complete results are available with the "babble01_pn01" particle prepended to the corresponding files in the "test/results" folder.

The addition of pink noise is though a strange way of achieving good speech recognition results. Let's be prudent and don't generalize: maybe the improvement is only manifest in a local environment. This last statement has to be demonstrated in order to validate it or reject it otherwise.

Let's suspect that the addition of pink noise modifies the signals in such a way that when scored against the acoustic models the values obtained are high. This is possible, good speech recognition results would be obtained but that wouldn't imply that the new system is valid for any case. The problem is that the original training speech recordings, the ones used in the production of the WSJ acoustic models, are not freely available. Thus, they cannot be analysed in order to determine if they have an inherent pink-like noise that would justify the improvements obtained.

In order to overcome this inconvenience, another set of acoustic models is used for this experiment: the Resource Management (RM1) acoustic models.

8.4 Regression tests results using the Resource Management acoustic models

This section is dedicated to the results and the conclusions that can be extracted from the new conducted regression tests using the Resource Management (RM1) acoustic models available in the downloads section of Sphinx-4.

These acoustic models have been created with the Resource Management database published by the LDC (Linguistic Data Consortium), who does not allow redistribution of the audio. Nevertheless, LDC offers a few acoustic samples before the whole set is bought, but anyway, these won't even be necessary to demonstrate the existence of a dependency of the results with the acoustic models (and the features of the training sequences used).

The Resource Management (RM1) speech samples, authored by P. Price, W.M. Fisher, J. Bernstein and D.S. Pallett, consist of read sentences modeled after a naval resource management task. The complete corpus contains over 25000 utterances from more than 160 speakers representing a variety of American dialects. The material was recorded at 16KHz with 16-bit resolution. This database is not as abundant as the WSJ, it is rather labeled as a Medium Vocabulary database. Note that while the RM1 database weights 8.4MB, the WSJ database weights 11MB.

The file named "RM1_13dCep_16k_40mel_130Hz_6800Hz.jar" is located at the "sphinx4" package in the downloads page of the project, hosted at SourceForge.net. This package contains the acoustic models created with the RM1 database. In Magnus, it can be found in the "lib" folder.

In order Magnus to use these models, in the "test/configfiles" folder, the configuration files "deptedu.config.<nose|se>.rm1.xml" have been prepared to do so.

In order to run the tests and retrieve the results, the *modus operandi* is the same as the previous section.

8.4.1 Acoustic insulated environment

The first situation proposed is the speech recordings in the acoustically insulated environment, as a reference of the performance of the system (using the new set of acoustic models) in the most favorable environment. Table 8.11 shows the obtained results.

Table 8.11: Recognition results obtained in an acoustic insulated environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	33.82	0.07	31.16
	SE	38.65	0.07	32.19
Female	No SE	42.61	0.08	29.92
	SE	45.22	0.08	28.96

As it can be seen in Table 8.11 the resulting WER is not low at all, worse than with the WSJ acoustic models (33.82% vs. 15.94% for male speech with no S.E. and 42.61% vs. 22.61% for female speech with no S.E.), and the speech enhancement process does not improve the results, it makes them even worse. These results respond to the mid-size vocabulary database, the smaller the training set, the less accurate the acoustic models. The only positive aspect of having inaccurate acoustic models is the fast processing of the data (0.07×RT vs. 0.7×RT) and the low memory footprint of the system (31MB vs. 63MB). On certain applications (such as resource-limited devices) these facts would be in a trade-off situation, but on Magnus this is not the case: accuracy is priorized all the time.

The complete results are available with the "original" particle prepended to the corresponding files in the "test/results" folder. Note at the ending of these files, the "rm1" particle to distinguish the files that correspond to this section from the files that correspond to the previous section, which have the "wsj" particle instead.

8.4.2 Pink noise polluted environment

This is the most critical environment: on one side, if the results obtained here yield a recognition improvement then the addition of pink noise actually proves to be a means of speech enhancement; on the other side, if the results yield a recognition decrease then the pink noise addition only proves to increase the recognition performance on a particular environment, not being a generic method of speech enhancement applicable to other situations. Table 8.12 shows the obtained results.

Table 8.12: Recognition results obtained in a pink noise polluted environment

Gender	Process	WER(%)	Speed(×RT)	Memory(MB)
Male	No SE	30.92	0.07	32.65
	SE	35.27	0.06	33.12
Female	No SE	46.09	0.07	32.58
	SE	48.70	0.07	32.94

As it can be seen in Table 8.12 the WER results are generally worse than with the acoustic insulation. Although the male speech recordings have resulted in a light increase (30.92% vs. 33.65% with no S.E. and 35.27% vs. 38.65% with S.E.), the female speech recordings have worsened considerably (46.09% vs. 42.61% with no S.E. and 48.70% vs. 45.22% with S.E.). These are not the splendid results that were supposed to appear if the addition of pink noise had a speech enhancement effect in all environments.

The complete results are available with the "pn01" particle prepended to the corresponding files in the "test/results" folder.

In summary, according to the experiments carried out with the RM1 acoustic models, the addition of pink noise to the speech signals doesn't improve the system's recognition accuracy, although with the WSJ acoustic models the effect seems to be the contrary. This fact may be attributed to the resulting features of the speech and pink noise mixtures, they must resemble the original speech training samples in some way, and then, since the speech

recognition system makes use of statistic methods to score the similarity between the incoming speech utterances and the acoustic models the resulting scores end up being more accurate.

The addition of pink noise may not be a generic speech enhancement method, but since the recognition results with the WSJ acoustic models seem to be more accurate than without the pink noise addition, and Magnus makes use of these WSJ acoustic models to perform the recognitions, the use of the pink noise addition to the incoming speech is justified.

8.5 Regression tests results using the WSJ acoustic models and an internal pink noise generator

This section brings to practice the idea of adding pink noise to the incoming speech utterances in order to obtain better recognition results when scored against the WSJ acoustic models. This method is only appropriate for the WSJ acoustic models, other acoustic models (such as the RM1 tested in the previous section) don't yield any better results.

In order to generate pink noise, the Voss algorithm is proposed. The following subsections deal with the description of this algorithm, its application in the development of Magnus and the results obtained with it.

8.5.1 Voss algorithm

This algorithm is described in [Gardner, 1978]. It creates pink noise by adding a series of white noise sources at successively lower octaves.

The Voss algorithm adds up several uniform random number generators that are evaluated in octave time intervals. The pattern that is seen in Table 8.13 shows (discrete) time moving horizontally to the right, and each line being a random number, or rather a square-wave sampled white noise source. Every pattern sample contains a white noise sample. Each row is updated at half the rate of the row above and the top row is updated every sample.

Table 8.13: Pattern of evaluation of the random number generators

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x		x		x		x		x		x		x		x		x
x				x				x				x				x
x								x								x
x																x

Each column in Table 8.13 represents a discrete time instant. Then, in order to produce the desired pink noise, at each time instant the sum of the different white noise sources (each row of the table) are summed.

By modelling each row as a sample and hold, or zero order hold (ZOH), applied to a white noise source, it is possible to work out each row's contribution to the overall spectrum. The output of the pink noise generator is then the sum of N rows. In Table 8.13, N equals 5.

Note that the analytic resolution of these functions is not trivial. Let's rather follow a numeric method, because it is possible to experimentally determine the Power Spectral Density (PSD) of the resulting pink noise by taking the DFT of a number of samples of the pink noise generator.

In order to do so, the file "test/tools/PinkNoiseNumbers.java" has been coded to ease the process. This source code file has to be manually compiled with the command:

```
javac PinkNoiseNumbers.java
```

Once the binary file "PinkNoiseNumbers.class" is produced, the program is launched with the command:

```
java PinkNoiseNumbers
```

This piece of code implements the Voss algorithm in Java and produces a set of 704000 samples, that correspond to an audio length of 44 seconds with

a sampling frequency of 16KHz. This audio length is more of less the same as the audio length of the rest of the noise sound files prepared for the tests. The amount of data that contains should be enough to produce accurate statistics. Nota that the pink noise produced is set to a range of '327', which corresponds to the 0.01 value in proportion to the dynamic range of the signal (2^{15}), like the rest of the noises prepared for the tests, thus $0.01 \cdot 2^{15} = 327$.

In order to obtain a text file with all the files generated by PinkNoiseNumbers, the standard output is redirected to a file named "data" with the following command:

```
java PinkNoiseNumbers > data
```

Once the data file is ready, Scilab is used to produce a WAV file with the noise generated by PinkNoiseNumbers. The whole code used is shown as follows:

```
da = read('data',-1,1);
t = da - mean(da);
tt = t./max(t);
wavwrite(tt',16000,'voss_pink_noise')
```

The short code shown above first loads the file in a variable named 'da', then substracts the mean of the data to all the values and stores the resulting vector in a variable called 't', then normalizes the resulting wave and finally writes the WAV audio file named "voss_pink_noise.wav".

Afterwards, Sox is used to convert the resulting WAV file into a low volume RAW file as seen previously and finally Audacity is used to perform the frequential analysis on the resulting audio (pink noise). Figure 8.9 shows the spectral plot of the resulting pink noise distribution.

As it can be seen in Figure 8.9, at first sight there's no much difference between this plot and the plot obtained for the previous pink noise. So, it is assumed that the Voss algorithm generates pink noise successfully. Let's see how it behaves with the speech recognition system in the following subsections.

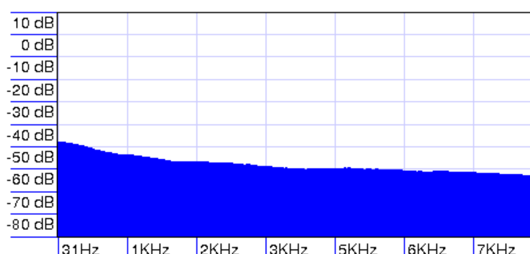

Figure 8.9: Frequential analysis of the pink noise generated by the Voss algorithm

Please refer to [Gardner, 1978] for further details about the Voss algorithm.

8.5.2 Pink noise generator implementation

The implementation of the pink noise generator in Java for Magnus is based on the Voss algorithm.

A first general implementation of the pink noise generator is produced ("src/PinkNoiseGen.java") to then instantiate it and use it in a new module for the frontend pipeline of Magnus ("src/PinkNoiseAdd.java").

Note that the pink noise numbers produced by the internal generator range from 0 to 327 as calculated previously. In order to transform these numbers to actual pink noise, a similar procedure to the procedure prepared in Scilab has to be applied in this new module: the pink noise numbers are substracted the mean of the distribution (which equals $\mu = 159.7705$) and the result is multiplied by 2 in order to reach the 1% of noise signal aimed for all the rest of the test noises. Equation (8.2) describes this prodeure mathematically.

$$y[n] = x[n] + 2 \cdot (pink_noise_number[n] - \mu) \tag{8.2}$$

Finally a new configuration file is prepared ("dept-edu.config.se.wsj.apn.xml") in order to use this module in the new configuration for Magnus. The results obtained with this new architecture are shown in the following section.

8.5.3 Impact of the internal addition of pink noise

The several tests proposed in the previous sections are run again with the new configuration for Magnus in order to determine the degree of improvement, in the case that there is any. The procedure is the same using the new configuration file. The results of the recognitions are dumped in the "test/results" folder with the "apn" particle appended to the corresponding files.

Table 8.14 compares the several results obtained with the new configuration to the best results obtained with the previous configuration. Note that the new configuration uses the WSJ acoustic models, the speech enhancement modules and the new internal pink noise generator based on the Voss algorithm.

Table 8.14: Results obtained with the internal pink noise generator (PNG)

Acoustic environment	Male speech WER(%)	Female speech WER(%)
Insulation	7.73 vs. 15.94	18.26 vs. 22.61
White noise pollution	28.50 vs. 21.74	40.00 vs. 40.00
Blue noise pollution	18.84 vs. 24.16	36.52 vs. 38.26
Violet noise pollution	20.77 vs. 28.99	36.52 vs. 43.48
Pink noise pollution	21.26 vs. 9.18	33.91 vs. 16.52
Red noise pollution	19.32 vs. 20.29	32.17 vs. 35.65
Grey noise pollution	22.22 vs. 21.74	32.17 vs. 36.52
Babble noise pollution	20.77 vs. 18.36	33.04 vs. 26.09

[2]WER(%) with PNG vs. WER(%) without PNG

In order to clarify the deviations of the results shown in Table 8.14, the differences between the values of the best old recognition results and the values of the new recognition results are displayed on a chart in Figure 8.10.

One of the first things that stick out in the chart shown in Figure 8.10 is that the results in the babble noise polluted environment are not the same as the results obtained with the babble noise and pink noise mixture analysed

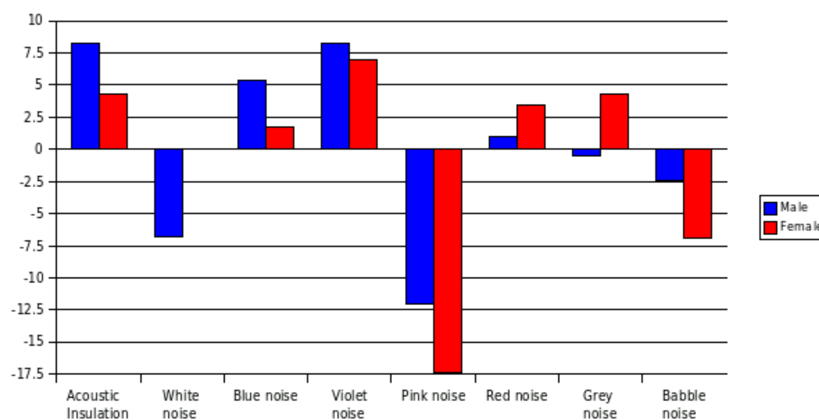

Figure 8.10: Results deviations with respect to the abscence of the pink noise generator

previously without an internal pink noise generator (a WER of 20.77% and 33.04% vs. 11.59% and 22.61%). This may be due to the precision of the arithmetic and the conditioning of the equations in the algorithms, apart from the imprecisions introduced by the approximation to pink noise of the Voss algorithm. In the present architecture the noises mixture takes place in the frontend of Magnus while previously it took place in Sox having prepared the audio samples with Audacity. Different initial conditions lead to different results, obviously.

Analysing the chart as a whole, with the internal pink noise generator only three out of eight environments show a decrease in the accuracy of the system. Curiously, the worst results correspond to the pink noise polluted environment, which indicates that the increase in accuracy shown in the previous tests is also dependent with the amplitude of the noise, as is shown than an excess of pink noise pollution produces the worst results.

The remaining five (out of eight) acoustic environments justify the use of the internal pink noise generator. Let's highlight the results obtained with the acoustic insulated environment: while originally the WER is 15.94% and 22.61%, these rates are improved with the external addition of pink noise achieving 9.18% and 16.52%, but with the internal addition of pink noise the rates even decrease to 7.73% and 18.26% (the recognition enhancement

only happens in the male voices, but considering that the external addition of pink noise is a kind of imaginary situation, the improvement should be evident).

In summary, as shown in the previous results, the internal addition of pink noise generally yields better results than without any noise addition. That's why the main configuration file for Magnus makes use of the pink noise generator. Despite of the poor recognition results obtained, it must be considered that the subjects of study for the experiments had no experience at all with the program. Once an user gets accustomed to dealing with Magnus, inconsciously gets improving his/her speech to gain a higher control over the application.

Chapter 9

Conclusions and Future Work

This is the last chapter of the thesis. It first deals with the conclusions obtained through the study and development of Magnus, a speech voice recognition system to control the mouse pointer and arrow keys of a keyboard with Catalan voice commands. Lastly, merged with the conclusions, the future work for the project conclude the thesis.

The Preface of this thesis states some goals that now shall be revised. The development of a Java application to control de mouse pointer of a PC has been accomplished, the mathematical theory concepts have been collected, understood to an extent, and explained in this thesis, which in its turn also explains in detail the development of this application. These objectives have been successfully achieved: Magnus runs fairly well and the documentation is fairly abundant and complete. They have some limitations, though, but the temporal restriction to hand in this thesis has keept some improvements still in the "to-do" list.

Firstly, relating to the GNU/Linux development environment. The experience of working under a UNIX-like platform couldn't have been more splendid. The trade-off between the complexity of the system and the potentiality that it offers grows bigger from day to day. It has been of crucial use for the production of this software application. Moreover, the high scalability that the system offers has enabled such an intensive processing application to run on resource limited computers.

Relating to Java, it is a very powerful programming language, without a doubt. The object-oriented programming and strict typification attributes enable the code to be most organized, thus permitting an useful and intelligible source code distribution. The virtual machine that runs the applications allows the program to be fairly independent of the host system, thus easing the monitoring processes. The multi-platform portability of the code is direct, once the (binary) bytecodes are obtained, they can be directly distributed. The vast amount of documentation on the Interet permits the learning of a whole new bunch of possibilities. The only setback for code purists is the automatic memory management. This may be somewhat a flaw, if the programmer aims to have a complete contol of the memory allocation processes involved, mostly for optimization processes. The lack of the programmer's control mechanisms to manage memory may derive in excessive memory footprints, which in no case are of any good. Please refer to [Kuzemin *et al.*, 2003] for another point of view of the pros and cons of using Java in scientific ccomputations.

Relating to Sphinx-4, the speech recognition engine used in Magnus to perform the recognition processes. This is a magnificient tool. At first, it may look a bit odd, due to the lack of documentation, especially when compared to HTK. But as one delves into the program discovers a most complete application that is worth to learn. Sphinx-4 gets the most out of Java, it profits from all the elegant advantages that the programming language offers, thus producing, in turn, an elegantly coded speech recognition engine. By the use of the Java interfaces, Magnus extends its modules to create the whole set of new speech enhancement processes to perform the research activities. It yields very good means of plug-and-play architecture that facilitates the testing experiments, as well as a whole set of instrumentation tools to score the quality of the system.

Relating to the acoustic models used by Magnus, these acoustic models have been created with American English speakers, with lots of different dialects, indeed, but the language used in Magnus is Catalan, not English. This aspect will prevent Magnus from achieving high accuracy results, in principle. As it is common sense, and also demonstrated in the previous chapters of this thesis, the recognition results are highly dependent with the acoustic models. It is obvious that if the acoustic models had been "recorded

in" Catalan the results would have been better (but considering that the subjects that collaborated in the project for the tests had no experience at all with the program, the obtained results may not be that bad). Since this seemed to be an obvious statement, the goals of Magnus were focused on other aspects of the speech recognition chain (making use of the speech enhancement techniques) in order to improve the results, but the creation of a good database in Catalan to train a new set of acoustic models is one of the aspects that could improve the performance of Magnus.

Relating to the speech enhancement modules proposed. The practice tests conducted conclude that the effect of the three speech enhancement modules developed have little enhancement effects on the speech recognition results, maybe a deeper investigation on the ambient noise conditions would have revealed something of interest to improve them. Fortunately, while running some tests it was discovered that with the addition of pink noise to the incoming speech signals the accuracy results were improved. This may seem substantially crazy, the addition of noise goes in the opposing direction of the classic goals in engineering! But it makes sense when the environment, the speech recognition engine, works statistically. If the acoustic models had been trained with noise polluted speech samples, then it would make sense to add noise to the testing speech samples in order to score higher accuracy results. This is a supposition that has not been proved since these training samples, the Wall Street Journal database, are not freely available.

Relating to the pink noise generator. The Voss algorithm produces pink noise but the obtained results with this pink noise, or possible approximation to pink noise, are not as good as expected. Apart from the direct and naive attributions to the precision of the arithmetic and the conditioning of the equations of the algorithms, this fact could be attributed to the order that the random generators change the load in the Voss algorithm. Some pink noise samples add up more random variables than others (see the pattern related to the Voss algorithm in the previous chapter) and this can cause large discontinuities in the wave when lots of the values change at once. In order to overcome this possible problem, other algorithms such as the Voss-McCartney algorithm would have to be tested and analyzed in detail. More information about the Voss-McCartney algorithm is available at: `http://www.firstpr.com.au/dsp/pink-noise/`.

Relating to the grammars. Magnus uses single word command-like constructs to interact with the speech recognition engine. This results in simpler grammar graphs that provide faster results, but this may induce the system to compute more errors. In order to improve the system's accuracy with this aspect, some research would have to be conducted with the aim of developing more complex grammars.

Relating to the documentation. In order to produce high quality documentation, the LaTeX typesetting system has been chosen to do the job. It includes features designed for the production of technical and scientific documentation. LaTeX is the *de facto* standard for the communication and publication of scientific documents, and since this thesis aims to provide a high quality source of information of the speech recognition application designed, as stated in the Preface, the choice of LaTeX is almost direct. The structurization, citation and referensation requirements have been covered by LaTeX with great success.

Finally, there is some other future work, in the form of improvements, that don't fit in the general aspects discussed in the previous paragraphs. They are listed in the following "to-do" list:

- Client-server architecture migration: this interesting feature would provide an additional degree of scalability.
- Language portability through Java I18N: the internationalization of Magnus would enable the application to be used by a larger community of end-users.
- Visual interface: the use of the SWING library instead of AWT would display a nicer look and feel (the graphical user interface).

Magnus is free software. It can be redistributed, studied and modified under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at the user's option) any later version. Magnus hopes to be useful to society and welcomes any volunteer to improve the application by fixing the cited improvements and/or proposing any (of the many) improvements that the program still needs.

Bibliography

[Adbulla and Kasabov, 2001] Abdulla, W. H. and Kasabov, N. K., *"Improving speech recognition performance through gender separation"*, Artificial Neural Networks and Expert Systems International Conference (ANNES), pp. 218–222, 2001, Dunedin (New Zealand).

[Colton, 2003] Colton, D., *"Automatic Speech Recognition Tutorial"*, Undergraduate elective course in Automatic Speech Recognition (CS 441) taught at Brigham Young University Hawaii, June 2003, Laie, Hawaii (U.S.).

[Davis and Mermelstein, 1980] Davis, S. and Mermelstein, P., *"Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences"*, IEEE Transactions on Acoustics, Speech and Signal Processing, (ISSN: 0096-3518), vol. 28 (4), pp. 357–366, August 1980, Santa Barbara, California (U.S.).

[Dugad and Desai, 1996] Dugad, R. and Desai, U. B., *"A tutorial on Hidden Markov Models"*, Technical Report No.: SPANN-96.1, Indian Institute of Technology, May 1996, Bombay (India).

[Fant, 1970] Fant, G., *"Acoustic Theory of Speech Production: With Calculations Based on X-Ray Studies of Russian Articulations"*, (ISBN: 978-9027916006), Mouton, 2nd Edition, 1970, The Hague (The Netherlands).

[Gardner, 1978] Gardner, M., *"White and Brown Music, Fractal Curves and One-Over-f Fluctuations"*, Mathematical Games, Scientific American, pp. 16–32, April 1978, New York, New York (U.S.).

[Ghahramani, 1998] Ghahramani, Z., *"Learning Dynamic Bayesian Networks, Adaptive Processing of Sequences and Data Structures"*,

Lecture Notes in Artificial Intelligence, (ISBN: 978-3-540-64341-8), Springer-Verlag, vol. 1387, pp. 168–197, 1998, Toronto (Canada).

[Gouvea *et al.*, 2004] Walker, W., Lamere, P., Kwok, P., Raj, B., Singh, R., Gouvea, E., Wolf, P. and Woelfel, J., *"Sphinx-4: A Flexible Open Source Framework for Speech Recognition"*, SML Technical Report Series, No. TR-2004-139, Sun Microsystems Laboratories, November 2004, (U.S.).

[Hwang and Huang, 1993] Hwang, M. Y. and Huang, X., *"Shared-distribution hidden Markov models for speech recognition"*, IEEE Transactions on Speech and Audio Processing, (ISSN: 1063-6676), vol. 1 (4), pp. 414–420, October 1993, Pittsburgh, Pennsylvania (U.S.).

[Knuth, 1964] Knuth, D. E., *"Backus Normal Form vs. Backus Naur Form"*, Communications of the ACM, (ISSN: 0001-0782), vol. 7 (12), pp. 735–736, December 1964, Pasadena, California (U.S.).

[Kuzemin *et al.*, 2003] Kuzemin, A. Ya., Minajlo, N. D., Safonov, I. M. and Shulika, A. V., *"Using Java in ingeneering and scientific computations and in designing systems"*, 5th International Workshop on Laser and Fiber-Optical Networks Modeling, 2003. Proceedings of LFNM 2003, (ISBN: 0-7803-7709-5), pp. 93–94, September 2003, Alushta, Crimea (Ukraine).

[Kybic, 1998] Kybic, J., *"Kalman Filtering and Speech Enhancement"*, Master's Thesis, Czech Technical University, 1998, Prague (Czech Republic).

[Lieberman, 2002] Lieberman, H., *"Common Sense Reasoning for Interactive Applications"*, MIT Media Lab Course, 2002, Cambridge, Massachusetts (U.S.).

[Nawab and Quatieri, 1987] *"Short-time Fourier transform, Advanced topics in signal processing"*, Prentice-Hall Signal Processing Series, (ISBN: 0-13-013129-6), Prentice-Hall, pp. 289–337, 1987, Upper Saddle River, New Jersey (U.S.).

[Oppenheim *et al.*, 1983] Oppenheim, A. V., Schafer, R. W. and Buck, J. R., *"Signals and Systems"*, Prentice-Hall Signal Processing Series,

(ISBN: 978-0137549207), Prentice Hall, 1st Edition, 1983, New Jersey (U.S.).

[Rabiner, 1989] Rabiner, L. R., *"A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition"*, Proceedings of the IEEE, (ISSN: 0018-9219), vol. 77 (2), pp. 257–286, February 1989, Murray Hill, New Jersey (U.S.).

[Rabiner and Juang, 1991] Rabiner, L. R. and Juang, B. H., *"Hidden Markov Models for Speech Recognition"*, Technometrics, vol. 33 (3), pp. 251–271, American Statistical Association, August 1991, Murray Hill, New Jersey (U.S.).

[Schmidt-Nielsen *et al.*, 2004] Divi, V., Forlines, C., Van Gemert, J., Raj, B., Schmidt-Nielsen, B., Wittenburg, K., Woefel, J., Wolf, P. and Zhang, F., *"A Speech-In List-Out Approach to Spoken User Interfaces"*, Proceedings of Human Language Technology Conference (HLT 2004), pp. 113–116, Association for Computational Linguistics, May 2004, Boston, Massachusetts (U.S.).

[Seltzer, 2003] Seltzer, M., *"Microphone Array Processing for Robust Speech Recognition"*, Ph.D. Thesis, Carnegie Mellon University, July 2003, Pittsburgh, Pennsylvania (U.S.).

[Shankar Chanda and Park, 2007] Shankar Chanda, P. and Park, S., *"Speech Intelligibility Enhancement Using Tunable Equalization Filter"*, IEEE International Acoustics, Speech and Signal Processing, (ISSN: 1520-6149), (ISBN: 1-4244-0728-1), vol. 4, pp. 613–616, April 2007, Honolulu, Hawaii (U.S.).

[Takeda *et al.*, 1998] Takeda, K., Ogawa, A. and Itakura, F., *"Estimating Entropy of a Language from Optimal Insertion Penalty"*, 5th International Conference on Spoken Language Processing, paper 0456, November-December 1998, Sydney (Australia).

[Viterbi, 1967] Viterbi, A. J., *"Error bounds for convolutional codes and an asymptotically optimum decoding algorithm"*, IEEE Transactions on Information Theory, (ISSN: 0018-9448), vol. 13 (2), pp. 260–269, April 1967, Los Angeles, California (U.S.).

[Woltzenlogel, 2007] Woltzenlogel, B., *"An Approximate Gazeteer for GATE based on Levenshtein Distance"*, Proceedings of the Twelfth ESSLLI Student Session, pp. 200, August 2007, Dublin (Ireland).

[Young *et al.*, 1989] Young, S. J., Russell, N. H. and Thornton, J. H. S., *"Token Passing: A simple conceptual model for connected speech recognition systems"*, Technical Report CUED/FINFENG/TR38, Cambridge University, July 1989.

[Young *et al.*, 2006] Young, S., Evermann, G., Gales, M., Hain, T., Kershaw, D., Liu, X., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V. and Woodland, P., *"The HTK Book (for HTK Version 3.4)"*, Cambridge University Engineering Department, December 2006, Cambridge (U.K.).

Thematic Index

accuracy, 29
acoustic likelihood, 14
acoustic score, 14
acronym
- ADC, 7
- AI, 33
- ANN, 11
- APF, 80
- ASR, 2
- AWT, 65
- BFS, 54
- BNF, 49
- CFG, 53
- CMN, 46
- CVS, 72
- DAG, 17
- DAW, 103
- DCT, 6
- DFT, 37
- DSP, 33
- DTW, 11
- FFT, 37
- FT, 44
- GUI, 64
- HMM, 11
- HTK, 32
- IT, vi
- JSGF, 70
- JVM, 32
- JWS, vi
- LPC, 37
- LW, 29
- MFCC, 5
- ML, 13
- PCM, 90
- PLP, 37
- PNG, 126
- PSD, 123
- RCT, 37
- RM1, 119
- RMS, 77
- SNR, 57
- STFT, 5
- TWML, 44
- WER, 30
- WIP, 29
- WSJ, 91
- XML, 37, 67
- ZOH, 123

algorithm
- baum-welch, 20
- Dynamic Time Warping, 11
- forward-Backward, 18
- Viterbi, 25
- Voss, 122

aliasing, 7
allophone, 8
Audacity, 90

Bakis topology, 21
bigram, 28

cepstral coefficients, 5
confusion matrix, 22

de-esser, 83
decoding, 14
dynamic programming, 25

embedded training, 25
error
    deletion, 97
    insertion, 97
    substitution, 97

features extraction, 5
filter
    active, 7
    Kalman, 7
    mel, 5
formant, 8
frames, 5

grammar, 9

hard-knee, 84
homophone, 9

isolated training, 25

language model, 14
language score, 14
Levenshtein distance, 100
log mel spectrum, 6

Mel Scale, 5
mel scale, 5
mel spectrum, 5
mondegreen, 9
morpheme, 8

N-gram, 28
noise
    babble, 115
    blue, 108
    grey, 114
    pink, 111
    red, 112
    violet, 109
    white, 106

observation sequence, 15

phone, 8
phoneme, 8

quantization, 7

sampling rate, 7
Scilab, 75
senones, 25
sibilant, 60
side-chain, 83
Sox, 90
speech
    fluent, 9
    spontaneus, 9
    voiced, 4
    whispered, 4

theorem
    Nyquist, 7
thread, 66
tied states, 25
tolerance, 8
training, 14
transform
    bilinear, 79
    Discrete Cosine, 6
    Fourier, 8
    Short Time Fourier, 5
transition matrix, 21
trigram, 28