CREATING A FRONT-END FOR A SPEECH RECOGNITION SYSTEM

by

RICHARD ANTHONY VIGORITO

A PROJECT

Presented to the Department of Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science.

Ashland, OR
June 2007

APPROVAL PAGE

"Creating a Front-End For a Speech Recognition System," a project prepared by Richard A. Vigorito in partial fulfillment of the requirements for the Master of Science in Computer Science. This project has been approved and accepted by:

Dr. Dan Harvey, Chair of the Examining Committee — Date 6/11/07

Dr. Lynn Ackler, Committee Member — Date 6/11/07

Mrs. Hart Wilson, Committee Member — Date 6/11/07

Dr. Joseph L. Graf, Dean of Sciences — Date 6/20/07

ABSTRACT OF PROJECT

CREATING A FRONT-END FOR A SPEECH RECOGNITION SYSTEM

by

RICHARD ANTHONY VIGORITO

In the 1950's speech recognition was believed to be a problem that would be solved within ten years. Today in 2007 computer scientists, engineers, mathematicians and physicists are still trying to achieve automatic speech recognitions (ASR) systems that are as accurate as the auditory perception of humans. Accents, slang, slurred speech, speech impediments and drawls all make it hard for computers to recognize speech. Humans have significant advantages over computers regarding speech recognition. The digital and finite nature of computers make things like "filling in the gaps" extremely difficult and less reliable. Humans do this very well. The goal of the project is to create a reliable front-end for a speech recognition system. The goal of this paper is to self contain all needed knowledge for creating the front-end. Future computer science students will reference this work as we continue the work of creating a speech recognition system.

TABLE OF CONTENTS

Chapter Page

1. Introduction 1
   1.1 ASR: A Two Part Problem 3
   1.2 The Sphinx Project 4
   1.3 The Sound Editor 5
   1.4 Project Goals 7
   1.5 Approach 8
2. Sphinx 9
   2.1 Overview 9
   2.2 Installing and Building 11
   2.3 Related Sphinx Modules 13
   2.4 Debugging Using Sphinx 14
   2.5 Sphinx Signal Representation 17
3. Filters 19
   3.1 Overview 19
   3.2 Low-Pass Filters 21
   3.3 High-Pass Filters 22
   3.4 Band-Pass Filters 23
   3.5 Moving Average Filters 24
   3.6 Preemphasis 26
4. Fourier Transform 30
   4.1 History and Purpose 30
   4.2 Decomposition 31
   4.3 Complex Transform Versus Real Transform 32

4.4 Implementations 34
4.5 Power Spectrum 37
5. Frame Blocking and Windowing 38
5.1 Frame Blocking 38
5.2 Windowing 40
6. Spectral and Cepstral Analysis 42
6.1 Overview and Purpose 42
6.2 Spectral Analysis Through the FFT 46
6.3 Mel Spectra 46
6.4 Mel Cepstra and the Discrete Cosine Transform 48
6.5 Perceptual Linear Prediction 51
7. Setting Up the SE Front End 56
7.1 Methods Involved 56
7.2 Static Variables 58
8. Testing and Results 59
8.1 Testing Setup 59
8.2 Test 1 Results 60
8.3 Test 2 Results 61
8.4 Test 3 Results 62
8.5 Test Conclusions 64
9. Conclusion 66
10. Continuing the Project 67
10.1 Front End Enhancements 67
10.2 Post Processing 69
10.3 A Quick Look Into the Back End 71
References 73

LIST OF FIGURES

Figure Page

Figure 1.1 6
Figure 2.1 9
Figure 2.2 10
Figure 2.3 13
Figure 2.4 16
Figure 2.5 17
Figure 2.6 18
Figure 3.1 20
Figure 3.2 20
Figure 3.3 21
Figure 3.4 21
Figure 3.5 22
Figure 3.6 22
Figure 3.7 22
Figure 3.8 24
Figure 3.9 25
Figure 3.10 25
Figure 3.11 26
Figure 3.12 27
Figure 3.13 28
Figure 3.14 29
Figure 4.1 31
Figure 4.2 33

Figure 4.3 35
Figure 4.4 35
Figure 4.5 36
Figure 4.6 36
Figure 4.7 37
Figure 5.1 39
Figure 5.2 40
Figure 5.3 41
Figure 5.4 41
Figure 6.1 45
Figure 6.2 47
Figure 6.3 48
Figure 6.4 50
Figure 6.5 50
Figure 6.6 51
Figure 6.7 52
Figure 6.8 52
Figure 6.9 53
Figure 6.10 55
Figure 7.1 57
Figure 10.1 70
Figure 10.2 71
Figure 10.3 72

LIST OF TABLES

Table Page

Table 1.1 5
Table 2.1 15
Table 8.1 60
Table 8.2 61
Table 8.3 62
Table 8.4 63

DEFINITION OF ACRONYMS

ac ..... Alternating current is an electrical current whose magnitude and direction vary

ACORNS Program headed by the SOU Computer Science Department to restore native languages

ASR ..... Automatic Speech Recognition is a system that converts a speech signal into a sequence of words

CMU ..... Carnegie Mellon Univeristy is a private reseach university who heads the Sphinx project

CPU ..... Central Processing Unit intereprets computer instructions and processed data

CS ..... Computer Science

CVS ..... Concurrent Versioning System is a computer revision control system

dc ..... Direct current is the constant flow of electric charge

DCT ..... Discrete Cosine Transform is a Fourier related transform that decomposes a signal into cosine waves

DFT ..... Discrete Fourier Transform is a tranform for Fourier analysis of finite-domain discrete-time signals

DSP ..... Digital Signal Processing is the study of signals in a digital representation

fc ..... Frequency-cutoff is the frequency within a filter kernel where frequencies are either accepted or rejected

FFT ..... Fast Fourier Transform is an efficient algorithm to calculate the DFT

FIR ..... Finite Impulse Response is an impulse response that has a finite number of nonzero values

HMM ..... Hidden Markov Models is a statistical model, assumed to be a Markov process, where one determines hidden parameters from observable parameters

HP ..... Hewlett-Packard Company is a global information technology company who is a a part of the Sphinx project

IDE ..... Integrated Development Environment is a computer software that aids computer programmers in developing software

IDFT ..... Inverse Discrete Fourier Transform calculates the time domain from the frequency domain

IFFT .....Inverse Fast Fourier Transform is an èfficient way to calculate the time domain from the frequency domain

IIR .....Inifite Impulse Response is an impulse response that has a infinite number of nonzero values

IPA .....International Phonetic Alphabet is a standardized representation of the sounds of spoken language

JDK .....Java Development Kit is a Sun Microsystem product to aid Java developers develop applications

JSAPI .....Java Speech API is an API for control recognizers, dictation systems and speech synthesizers

LPCC .....Linear Prediction Cepstrum Coefficients are cepstrum coefficients computed using linear prediction

MERL .....Mitsubishi Electric Research Labs aided in the Sphinx project

MFCC .....Mel Frequency Cepstrum Coefficients are cepstrum coefficients calculated by the Mel-scale

MIT .....Massaschusetts Institute of Technology aided in the Sphinx project

PLP .....Percuptual Linear Prediction is a mathematical operation where future values of a discrete-time signal are estimated as a linear function of previous samples

RPT-AT ..Ratio of processing time to audio time

SDK .....Software Development Kit is a tool that aides computer programmers in creating applications and software

SE .....Sound editor program created by Dan Harvey to aid in the instruction of DSP concepts

SOU .....Southen Oregon University

TTS .....Text to Speech Synthesis is the process of creating a speech signal from a stream of text

VCS .....Version Control System is a system for managing multiple revisions of the same unit of information

WER .....Word Error Rate is the percentage of false matches a ASR system produces

XML .....Extendable Markup Language is a general purpose markup language whose main purpose is to facilitate share of data across different information systems This concept has been introduced in the ASR and TTS field with the creation of voice browers, implemented by VoiceXML

CHAPTER 1

Introduction

Digital Signal Processing (DSP) is extremely important to successfully designing automatic speech recognition systems (ASR). DSP technologies help us to understand and manipulate the auditory signal. DSP is also important for applications relating to radar and sonar, image mapping, data compression, medical imaging, telephone technology and more. Many of these applications are better understood than ASR so the DSP involved in these areas lead to better results. Scientists in the field of ASR are still searching for the DSP solution that closely mimics the human auditory and perception system. These efforts will eventually make ASR systems more reliable.

This is not to say that there are no reasonably accurate ASR systems today. A good example of this is automated telephone systems. Most everyone today has experience with automated phone guidance systems; for example many banks use them. Even though they tend to be annoying, one has to admit they are reasonably accurate.

The ASR system Dragon NaturalSpeaking9 by Nuance is a current speech recognition system on the market. Dragon NaturalSpeaking9 is geared at helping people harness the power of speech recognition for use in the internet and Microsoft based applications (Nuance, 2006). Based on the demo from the Nuance website, the Dragon system seems to be very effective and accurate. Nuance is able to create text documents and respond to commands that switch between applications.

VoiceXML is a markup language that uses ASR and text-to-speech synthesis (TTS) (VoiceXML, 2006). VoiceXML defines the voice dialogue between human and computer. The XML guides computer output based on user input. The recognition and synthesis is done in environments known as voice browsers. These voice browsers consist of both ASR and TTS machines (VoiceXML, 2006). W3C, AT&T, IBM, Lucent

and Motorola have all played roles in the development of VoiceXML (VoiceXML, 2006).

Sphinx by Carnegie Mellon University is another ASR system worth mentioning. Sphinx is an open source ASR system written entirely in Java. Sphinx is considered one of the best ASR systems in the field. For this reason, we drew heavily from Sphinx in our research. The availability of its source code makes research much easier and useful than comparing papers and manuals created by other ASR systems. Sphinx code is entirely Java, which has great documentation, runs on all platforms, and will be familiar to future students working on this project.

The purpose of this project is to create a self-contained ASR system. We plan to integrate this system into the ACORNS, [A][C]quisition [O]f [R]estored [N]ative [S]peech, project. ACORNS is an on-going project spawned by the Southern Oregon University (SOU) Computer Science department. It's purpose is to aid tribal efforts to restore the number of Native American indigenous speakers. The acronym ACORNS honors those tribes whose support helped initiate this project. The acorn has historical significance to Native Americans of the Pacific Northwest because it is sacred.

The vast majority of Native Americans languages are dying out and becoming extinct. The ACORNS project hopefully will help to reverse this trend. Its focus is to create software that enables language teachers to easily create speech lessons. Students can practice their skills when they practice with these lessons.

ASR technologies will contribute significantly to the ACORNS project. Our desire is to use ASR to implement computer games and other computer aided tools. The aim is to teach Native American Languages in a user-friendly environment that is both educational and fun. Incorporating ASR technologies into these games and tools will also promote learning of spelling and grammar.

Word pronunciation tools are other closely related applications that we will integrate into the ACORNS project. These pronunciation tools will allow students to visually see how their pronunciation matches that of a fluent speaker.

Our contribution to the overall goal of creating an ASR system in ACORNS is to produce a functioning front-end. The front-end handles the DSP functions and extracts features that are useful for speech recognition. Future work to complete this project will involve back-end processing that will translate these features into recognizable units of speech.

This paper is organized as follows. Section 2 describes the use of Sphinx in this project. Section 3 describes the various filters and filter types used in DSP. Section 4 describes the Fast Fourier Transform (FFT). Section 5 explains the purpose and implementation of framing a signal and applying windowing functions. Section 6 describes spectral and cepstral analysis. Section 7 describes how we set up the Sound Editor's front-end. Section 8 discusses our tests and their results. Section 9 is the conclusion section. Section 10 will discuss possible work to be done in the future.

1.1 ASR: A Two Part Problem

The problem of speech recognition historically has been broken into two parts. These two parts are the front-end and the back-end. The front-end is where most of the DSP applications are performed. Such processing can determine such things as time, energy, amplitude but most importantly frequency. Being able to identify frequencies within a signal is key to reliable audio comparison in ASR. Each minimal unique unit of sound, a phoneme, has signature frequencies. There are various ways to process these frequencies. The ASR system front-end distinguishes frequencies and uses them to derive what is called a feature vector. Our project mimics the feature vector of Carnegie Mellon University ASR project Sphinx. The feature vector is a thirteen-element vector containing Mel Frequency Cepstrum Coefficients (MFCC), with the energy held in the very first position. The energy is representation of the power in the entire MFCC vector. We will better explain energy, and MFCC in the spectrum section of this paper. Sphinx

takes the feature vector a step further and adds a derivative and second order derivative for each of the 13 MFCC's leading to a feature vector of 39 coefficient values.

The second part of ASR is the back-end. The back-end translates the feature vector into a meaningful unit of speech (i.e., phonemes, words and/or sentences.) Our project focuses specifically on front-end processing. This leaves lots of room of for research by future SOU CS students. This research will most definitely include Hidden Markov Modeling (HMM). Markov models are state machines. These machines receive a sequence of observed states as input, and based on probabilities computes the cause of these observations. For example, given a series of phonemes as input, a HMM algorithm computes the most likely matching word or sentence. In ASR, a language's grammatical and pronunciation rules determine the makeup of the Markov model machines.

1.2 The Sphinx Project

Many popular ASR systems exist today. Carnegie Mellon University has been a leader in the field for over ten years. Sphinx is a very popular foundation for research in academia for ASR and DSP. Because of its proven track record, we chose Sphinx as the research tool for implementing this project. Sphinx also has other advantages. Sphinx is open source, extremely well documented and entirely written entirely in Java. Many of the other popular ASR systems are written in C. Many might argue that ASR systems in C are more efficient. However, we want to eventually integrate this project into ACORNS software, and ACORNS uses Java.

Speed is very important in DSP and especially ASR. Sphinx' previous systems were written in C++ and speed comparisons between the old systems and the new one seem to be comparable. Sphinx believes the positive characteristics of Java outweigh any efficiency disadvantages. Many efficiency studies exist that compare Java performance to C and C++. "Optimizing NET Compilers for Improved Java Performance" is one such study. It shows that with proper optimization Java can achieve performance levels between 23% and 83% of C using industry standard benchmarks (Cheng-Hsueh, Conte,

Johnson, Gyllenhaal & Wen-mei, 1997). This is a speed improvement of 17.3x's (Cheng-Hsueh, Conte, Johnson, Gyllenhaal & Wen-mei, 1997).

Sphinx 1 and Sphinx 2 libraries were written in C. Carnegie Mellon claims that Sphinx 2 is CMU's fastest speech recognition system (Carnegie Mellon University, 2004). Sphinx 3, written in C++, rated faster than Sphinx 4. However Sphinx 4 is most accurate speech recognition system to date. (Carnegie Mellon University, 2004).

Table 1.1 shows additional data regarding Sphinx performance. The horizontal fields we are concerned with are the concerned with are WER which is the word error rate on a single CPU configured system and RPT-AT which is the ratio of processing time to audio time of a single CPU configuration (Carnegie Mellon University, 2004). The data shows that a small dictionary yields good results in both word error rate and processing ratio. Even the 1000 and 5000 word dictionaries perform well. Sphinx' testing proves their system to be both efficient and reliable. Drawing heavily upon Sphinx implementation we hope to eventually produce comparable, or superior results.

Language Model	Vocabulary Size	WER	RPT-AT
Isolated digit recognition	11	0.168	0.03
Continuous digits	11	0.549	0.07
Trigram	79	1.192	0.25
Trigram	1000	2.88	0.5
Trigram	5000	6.97	1.22
Trigram	60000	18.756	4.4

Table 1.1

1.3 The Sound Editor

This project extends the Sound Editor (SE) framework. The SE is software developed for the ACORNS project. It's purpose is to help SOU CS students learn

various signal processing concepts. The first release of SE was in the fall of 2006. This version implemented the Fourier transform and Hamming windowing. It did not include the rest of the components needed for ASR front-end processing. As part of this project, we added these missing components. We will discuss the specifics in further chapters.

Figure 1.1 visually depicts the relationship between the classes discussed in this section. This project focused mainly on two SE classes. These are WaveDisplayPanel and SignalAnalysis. The WaveDisplayPanel class controls the visual display of a sound wave. Our frontEndDriver method is part of this class. This method handles the interaction between methods needed to implement the front-end.

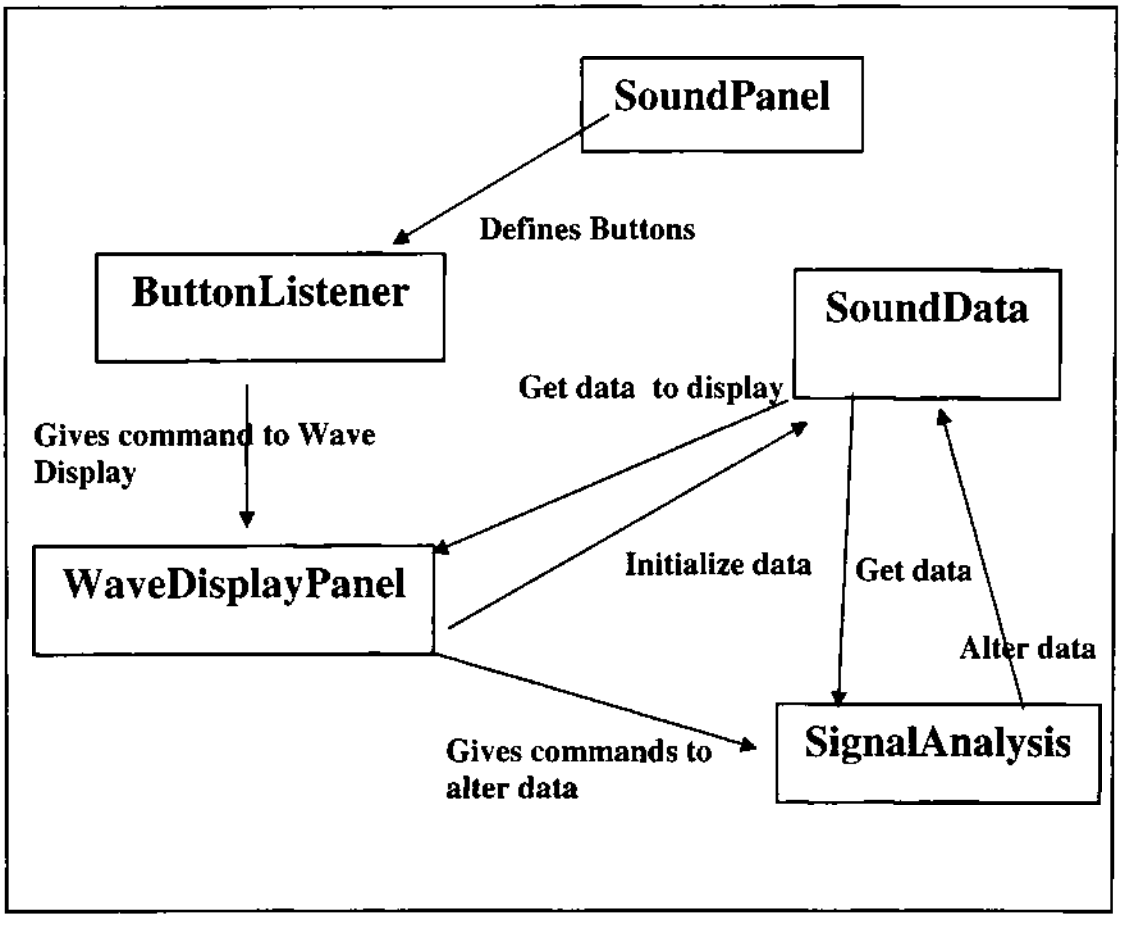

Figure 1.1

We implemented the front-end methods in the SignalAnalysis class. The following chapters provide specifics. The SignalAnalysis class is the natural place for these front-end methods. All SE methods for analyzing and manipulating signal data are in this class.

Two other SE classes of importance are the ButtonListener class and the SoundPanel class. SoundPanel is the place to define GUI buttons. ButtonListener contains logic to listen for user mouse clicks and initiate appropriate algorithms.

The SoundData class defines the signal data. This class translates an audio signal in time domain data. The time domain data then can be analyzed and manipulated using SignalAnaysis class methods

1.4 Project Goals

The ultimate goal of the ACORNS front-end is to automatically be able to recognize a set of International Phonetic Alphabet (IPA) vowels. The International Phonetic Alphabet is the recognized standard created by linguists. It defines an alphabet that contains every known sound from every known language. IPA is ideal because once we can recognize IPA phonemes, we can extend our recognition technology to any language, making our front-end truly language independent. This will involve creating appropriate dictionaries and grammars to incorporate a particular language's syntax and semantics.

The ASR problem is very difficult and man-hour intensive. It is impractical for one person to build a reliable ASR by him/herself. ACORNS is a long-term project that will involve contributions from many different faculty and students. This part of the project implements front-end functionality and DSP functions needed to create an accurate feature vector. This paper is a building block containing description of all the implementations and considerations needed to understand these front-end functions. Future research will refine this front-end for speed and performance, and stimulate research and development of the back-end.

1.5 Approach

Sphinx is a very useful tool created by Carnegie Melon over a ten-year period. Since we rely heavily on Sphinx design, it is important to have a working knowledge of the various modules and how they relate. Reading the documentation and then examining the source code makes contributing to this project much easier.

Sphinx is a plug-able program meaning users can select from many different plug-ins to produce similar results but by different means. For example, there are different ways to compute a cepstrum. The Cepstrum make up and the resulting feature vector will be discussed better in Section 6.4. Analysis of the various plug-in features helped inform the front-end approach to take. Sphinx' modularity provides lots of source code to examine and draw comparisons.

CHAPTER 2

Sphinx

This section provides a basic description of the Sphinx system.

2.1 Overview

As mentioned above, Sphinx is a state of the art ASR system created by Carnegie Mellon University. Additional companies and research groups contributed significantly to this project. These companies include Sun Microsystems Laboratories, Mitsubishi Electric Research Labs (MERL), and Hewlett Packard (HP), University of California at Santa Cruz (UCSC) and the Massachusetts Institute of Technology (MIT) (Carnegie Mellon University, 2004). Sphinx is an ASR system that performs multiple speech recognition tasks via many different avenues. Mel Frequency Cepstrum Coefficients

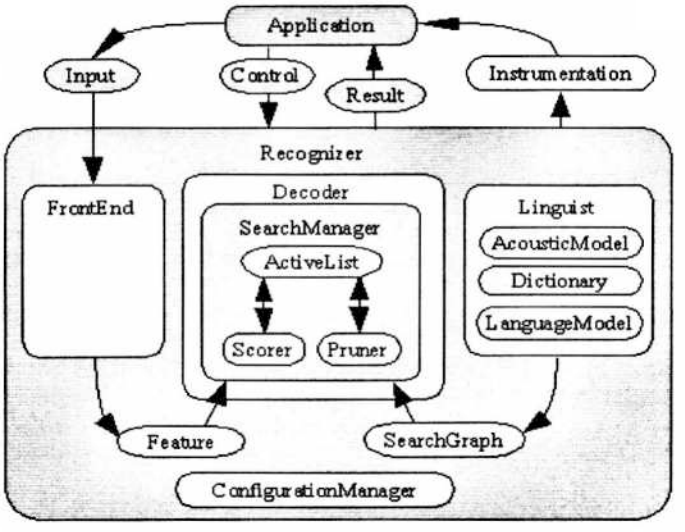

FIGURE 2.1 (Carnegie Mellon University, 2004)

(MFCC) or perceptual linear prediction (PLP) methods are possible front-end variations a user can define to compute cepstrum.

Figure 2.1 is a diagram of the architecture of Sphinx. The Configuration Manager defines Sphinx plug-ins. For example, the Configuration Manager defines what type of front-end modules Sphinx uses. It also defines what back-end search graph to use. The linguist and the back-end search graph define what dictionary of words the system can recognize. The Decoder scores and prunes feature vectors in accordance to the search graph. This is the "recognition" step, matching feature vectors to words.

As Fig. 2.1 depicts the front-end that is only a fraction of the Sphinx speech recognition solution. Figure 2.1 doesn't properly explain all of the internals of the front-end. Figure 2.2 is a diagram of the front-end and how it handles the various processes. The front-end is a series of data processors that inputs data and executes algorithms to affect a transformation leading to the output (Carnegie Mellon University, 2004). We will explain below the specific operations these data processors execute.

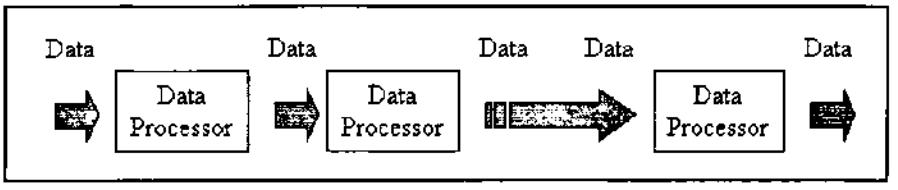

Figure 2.2

The great advantage of using Sphinx is that it is open source and written in Java. Beyond these two conveniences Sphinx is quite accurate. Many test programs are available in the *cmu.sphinx.tools* and *tests* packages to demonstrate their accuracy. The Sphinx website contains data corresponding to many bench tests (Carnegie Mellon University, 2004). Table 1.1 illustrates these results.

2.2 Installing and Building

We installed Sphinx using Netbeans Integrated Development Environment (IDE). Sphinx project documents its installation and building instructions at: http://cmusphinx.sourceforge.net/sphinx4/. For completeness, we document the steps we used as it relates to using Netbeans' method:

1. Have the correct operating system
    1. Solaris Operating Environment
    2. Mac OS X
    3. Linux
    4. Win32 operating systems
2. Have the required software
    1. Java 2 SDK, Standard Edition, v1.4.2 or better
    2. NetBeans IDE 4.0 or better
3. Build Sphinx4 using Netbeans

    4. Start NetBeans by either double-clicking on the NetBeans icon or through a command line. Select `Versioning->` `CVS->` `Check out...` from the `File` menu.

    This will pop a `Enabling Advanced VCS Command Options` window. Click `Check out...` button, pressing CTRL while clicking.

    5. Setting up a CVS Checkout & Import in Versioning
        1. Enter a workspace folder in the `Working Directory` field.
        2. Select `pserver` in the `CVS Server Type` field.
        3. Enter `cmusphinx.cvs.sourceforge.net` in the `CVS Server Name` field.
        4. Leave the default port (usually 2401) in the `Port` field.
        5. Enter `anonymous` in the `User Name` field if you are not a developer in the cmusphinx project, or your SourceForge username, if you are a developer.
        6. Enter `/cvsroot/cmusphinx` in the `Repository Path` field.
        7. Select `Use Built-In CVS Client` option, if you are not a `developer` or select `Use Built-In CVS Client` and enter `extssh` in the `Remote Shell` field, if you are a developer.
        8. Leave the `Password` field blank.
        9. Select `Module(s)` and enter `sphinx4` in the `Module(s)` field.
        10. Leave the `Revision or Tag` field blank.
        11. Leave the `Date` field blank.
        12. Click on `Force a Head Revision Match if Tag/Date not found`.
        13. Click on `OK` button. This will pop a `VCS Output` window with an active tab `Check Out....`
    6. Creating a project from CVS Resource
        1. Select `New Project...` from the `File` menu. This will pop a `New Project` window.
        2. Select `General` in `Categories` and after choose a `Java Project with Existing Ant Script`.
        3. Click on `Next` button. This will pop a `New Java Project with Existing Ant Script` window.

4. For Location browse to the workspace folder previously specified. If you browsed to the correct folder, all remained blank form will be filled automatically.

5. Click on Set as Main Project. Click on Next button. This will show Build and Run Actions tab.

6. If you followed correctly upper instructions, first three forms will be filled according to theirs label names. Click on Next button. This will show Source Package Folders tab.

7. Click on Add Folder... button for Source Package Folders and select lib folder. Add com and edu folder following same operations, then press OK.

8. Click on Add Folder... button for Test Package Folders and select tests folder. Add demo folder following same operations, then press OK.

9. Select JDK 1.4 in the Source level field. Click on Next button. This will show Java Sources Classpath tab.

10. Click on Finish button.

7. Setup JSAPI 1.0

1. Before you build Sphinx-4, it is important to setup your environment to support the Java Speech API (JSAPI), because a number of tests and demos rely on having JSAPI installed.

8. Add JSAPI 1.0 as a jar to the NetBeans Project

1. While the previous step sets up a jsapi.jar file in the lib directory, this steps makes jsapi.jar visible within the NetBeans Project. Note that in the instructions that follow, we assume that the project name is sphinx4. We named it while creating a project under NetBeans earlier.

2. Select Projects from the Window menu.

3. Select Properties by right-clicking on Sphinx-4 project.

4. Select Java Sources Classpath option in the left column. This will show up the Java Sources Classpath on the right hand side of the window.

5. Click on the Libraries tab and then, click on Add JAR/Folder... button. This will pop a browsing window.

6. Select jsapi.jar by navigating to lib directory, and then click Open button on the browsing window.

7. Next, click on OK button. This will add jsapi.jar as a jar to the NetBeans Project.

9. Add Sphinx-4 specific jars as jars to the NetBeans Project

1. This step makes the TIDIGITS_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar, WSJ_8gau_13dCep_8k_31mel_200Hz_3500Hz.jar, and WSJ_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar in the lib directories visible within the NetBeans Project.

2. Select Properties from the Project menu. This will pop a Project Properties - Sphinx-4 window.

3. Select Java Sources Classpath option in the left column. This will show up the Java Sources Classpath on the right hand side of the window.

4. Click on Add JAR/Folder... button. This will pop a browsing window.

5. Select TIDIGITS_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar by navigating to lib directory, and then click Open button on the browsing window.

6. Select WSJ_8gau_13dCep_8k_31mel_200Hz_3500Hz.jar by navigating to lib directory, and then click Open button on the browsing window.

7. Select WSJ_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar by navigating to lib directory, and then click Open button on the browsing window.

8. Next, click on OK button on the `Project Properties - Sphinx-4` window. This will add these jar files as jars to the NetBeans Project.
9. At this point, you should be able to run and debug applications such as demo.sphinx.hellodigits using NetBeans.

Once building is complete, the Sphinx application is ready to execute. Test your installation by executing Sphinx test programs. HelloWorld.java, HelloDigits.java and HelloNGrams.java, are sample programs that exist in the *cmu.sphinx.tools* package.

2.3 Related Sphinx Modules

Our front-end mimics the Sphinx approach. We implement our front-end as an extension to the ACORNS Sound Editor. It is important to understand the design and structure of the Sphinx architecture. Figure 2.3 is a diagram of the Sphinx front-end. A frame is section of the input audio signal to which processing is applied. The Sound Editor's front-end is based of the Sphinx4 design so all processing is done within a frame.

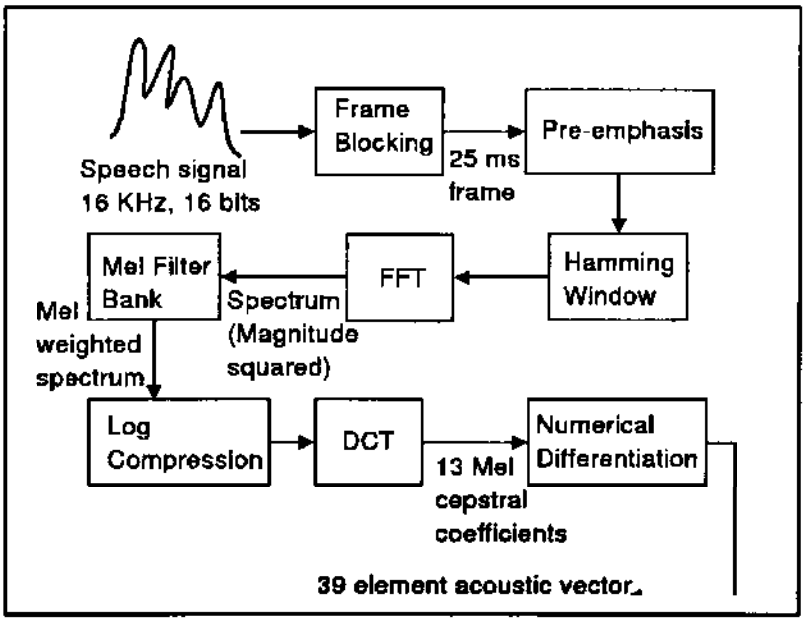

Figure 2.3 (Binu, 2004)

Table 2.1 presents a spreadsheet that briefly describes each of the Sphinx modules and their Sound Editor (SE) counterparts. Block 1, frame blocking as mentioned above and in chapter 5, breaks a signal into smaller sections so processing is done to small parts that sum to the signal. Block 2, preemphasis discussed in section 3.5, emphasizes higher frequencies. Block 3, Hamming window, reduces the influence frame edges have on the frame as a whole. Chapter 5 discusses why this is important. Block 4, FFT, takes frames that are in the time domain and translates them into the frequency domain power spectrum. We discuss its implementation and importance in chapter 4. Mel spectrum, block 6, tries to mock the critical frequencies human ears pick up. Block 7, log compression, and block 8, mel cepstrum, are often defined together. In our implementation, log compression is a module within the mel cepstrum module. Log Compression turns the non-linear signals linear (Seltzer, 1999). The theory behind cepstrum is that human sound is made up of two signals, the sound from the vocal cords and distortion from the sound track. Cepstrum calculation separates these two signals.

2.4 Debugging Using Sphinx

It is important to verify that our front-end produces results which closely match those of Sphinx. Therefore, at each front-end step we produced output from Sphinx and compared that output to our front-end. Once the appropriate print statements were in place we ran the FeatureFileDumper.java tool Sphinx provides for front-end analysis. This is done by right clicking the build.xml in cmu.sphinx.tools.feature and selecting *cepstra producer*. The FeatureFileDumper class has other selections available such as *plp_producer* or all but since the Sound Editor computes MFCC's we call the *cepstra_producer*, which produces MFCC's. Choosing the *cepstra producer* insures that the modules Sphinx calls corresponds to the modules in the Sound Editor front-end. Performing this step is shown on Fig. 2.4.

Module	Function	Sphinx Class	SE Implementation
Frame Blocking	Break signal into small overlapping pieces so we can more closely identify the different frequencies.	Did not use Sphinx as a source of research	Within the frontend method in WaveDisplayPanel.java
Pre-Emphasis	A high pass filter that compensates for a decrease in signal of time, attenuation, by putting an emphasis on the higher signals [3].	Preemphasizer.java	Preemphasis method in SignalAnalysis.java
Hamming Window	Function to reduce the influence of frame boundaries.	RaisedCosineWindower.java	ApplyWindow and CreateHammingWindow methods in SignalAnalysis.java
FFT	A very efficient algorithm to calculate the frequency domain from a time domain.	DiscreteFourierTransform.java	FFT Method in SignalAnalysis.java
PowerSpectrum	Its the particular strength of a frequency within the given frame [3].	Handled within DiscreteFourierTransfrom.java	PowerSpectrum method in SignalAnaysis.java
Mel Spectrum	The filter to more closely mimic the critical bands of human hearing	MelFilter.java and MelFilterBank.java	produceMelFilter.java, buildMelFilterBank.java and applyMelFilterBank.java all within SignalAnalysis.java
Log Compression	The last stage of converting spectra to cepstra is as the Inverse FFT (IFFT) of the log of the FFT. Within a series of real numbers using the DCT is equal to using the FFT.	Within DiscreteCosineTransform.java	Logcompression method in SignalAnalysis.java
Mel Cepstrum	Same definition as Log Compression	DiscreteCosineTransform.java	ComputeMelCosine, applyMelCosines and getMFCCs methods in SignalAnalysis.java

Table 2.1

As stated above the Sphinx front-end is plug-able. Computing MFCC's is not the only front-end option. Other historically famous front-end approaches are perceptual

linear prediction (PLP) and a hybrid of the MFCC's and PLP (Gold & Morgan, 2000). Sphinx allows for comparison between the two approaches and that is why Sphinx provides *all and PLP* selections. Our front-end uses MFCC only, we do not use these other options. MFCC's were chosen because over small data sets MFCC's perform better than PLP coefficients (Hogberg, 1997).

We verified our algorithms by comparing outputs of each SE module to the corresponding module of Sphinx. We began at the beginning of the chain of front-end modules. Once these results were verified we could mover to the next step in the chain. These steps were reproduced until all modules within the front-end were verified as matching.

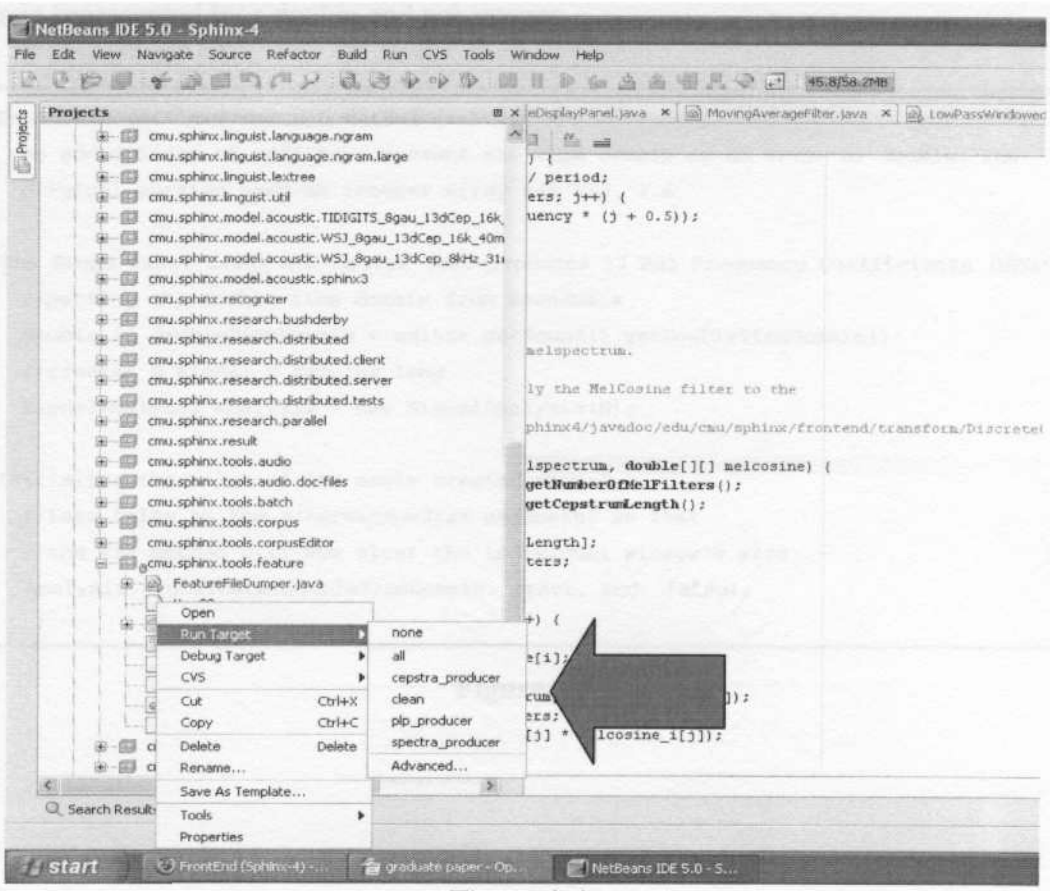

Figure 2.4

2.5 Sphinx Signal Representation

Sphinx and the SE implementations break the entire signal into small pieces, called frames. Sphinx's implementation of this slightly differs from our original SE. The FFT expects input's length to be the closest power or 2 greater to or equal to the frame length. In the original implementation of the SE this step was taken care of in the SignalAnalysis constructor. We extended the entire signal to a power of two, not each individual frame. These slight differences matter and affect the values of our feature vector. Handling this discrepancy we put the frame adjustment in the FFT method of the SignalAnalysis class. This resolved mismatching data. Figure 2.5 describes changes to the original algorithm. Figure 2.6 shows the additional code needed to make the time domain represented by a double and not integer.

```
1. SoundData getTimeFromAudio method
     We added lines of code to represent the time domain as an array of double. The
     original version used an integer array see Fig. 2.6

2. The Sound Panel front end driver that produces 13 Mel Frequency Coefficients (MFCC's)
     //getting the double time domain from SoundData
     double [] doubleTimeDomain = editor.getSound().getDoubleTimeDomain();
     //creates a signal N samples long
     SignalAnalysis analysis = new SignalAnalysis(N);

3. Initialize the value of the newly created signal.
     //Pass false to the alterWindowSize parameter so that
     //the fft method will now alter the individual window's size
     analysis.signalData(doubleTimeDomain, start, end, false);
```

Figure 2.5

```
//Extract Sound data from the Sound File and store in the time domain
//returns Integer array of amplitude values or null.
Method: getTimeDomainFromAudio no parameter returns an array of ints

IF (audioData array is null or frame Size In Bytes equals 0)  return null;
END IF

int numChannels = channels;

timeDomain = new int[audioData.length/frameSizeInBytes];
doubleTimeDomain = new double[audioData.length/frameSizeInBytes];
FOR (int f=0; f<timeDomain.length; f++)
     timeDomain[f] equals the time value done by calling getTimeValue
     doubleTimeDomain[f]=cast to double (timeDomain[f]);
END LOOP

// this step is important because the front-end values expect double values.
//unlike ints using doubles no precision is lost
doubleTimeDomain = bytesToValues(audioData,0,audioData.length,2,true);
return timeDomain;

Method: getDoubleTimeDomain returns the double time domain instantiated in
getTimeDomainFromAudio
```

Figure 2.6

CHAPTER 3

Filters

Filters are important algorithms that scientists apply to signals. These algorithms reject, allow, emphasize or de-emphasize certain frequencies. Filters also are very useful in minimizing signal distortion and noise.

3.1 Overview

DSP filters help accomplish two specific tasks. These are signal separation and signal restoration (Smith, 2003). This is important in speech recognition. Speech separation helps remove noise and signal restoration combines frames to make up the original signal. Any filter performed in this project will be done digitally. Analog filter implementations do exist. Our implementations execute on a PC computer. Due to the digital nature of computers we use digital filters. Fortunately this approach is more desirable because digital filters can achieve thousands of times better performance than analog filters (Smith, 2003).

Before diving into filter specifics and implementation details it is important to understand the filter jargon that is associated with the field. All linear filters have an impulse response, a step response and a frequency response. Knowing any one of these characteristics is sufficient to be able to calculate the others (Smith, 2003). A linear filter applies a linear function to a time-varying input signal. The Impulse Response is the exit signal of a linear system whose input signal is the Delta Function. The Delta Function is an all zero signal with the exception of one lone point at point (0,1). The Step Response is the exit signal of a linear system whose input signal is the discrete integration of the impulse response. The Frequency Response is the exit signal of a linear system whose input signal is the FFT of the impulse response. Each of these responses is important

because they describe complete information about the filter but in a different way. They define how the filter will respond in different circumstances (Smith, 2003). In the case of linear filters the easiest way to implement a digital filter is by convolving the input signal with the impulse response (Smith, 2003). Linear Systems are systems that have the properties of additivity and homogeneity. Convolution is mathematical way to get a third signal from the combination of two.

h[n] in Fig. 3.1 is an impulse response to the delta function. Java code for convolution is written in Fig 3.2, where x is the input signal, h is the impulse response and y is the output (Harvey, 2006).

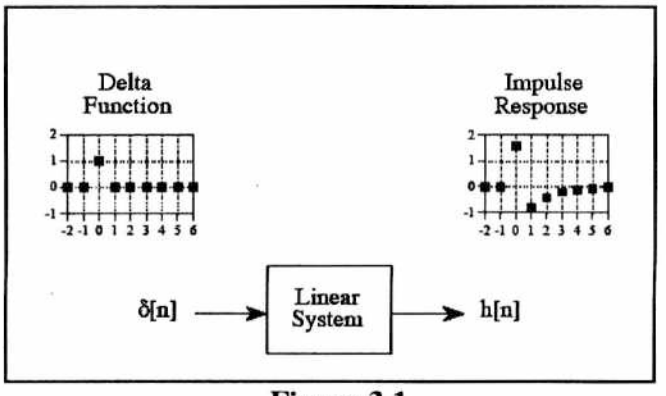

Figure 3.1

```
int [] convolve(int [] x, int [] h){
      int[] y = new int [x.length+h.length];
      for(int i=0; i<x.length; i++)
          for(int j=0; j<h.length; j++)
                y[i+j] += x[i]*h[j];
      return y;
}
```

Figure 3.2

An alternate approach to filter creation is recursion. The convolution approach uses points in the input signal and filter kernel to compute each point in the output. A filter kernel is the impulse response of a filter that uses convolution. Recursion uses a

similar but more efficient approach. The recursion method saves calculations by using already calculated output points (Smith, 2003), (Harvey, 2006).

3.2 Low-Pass Filters

Low pass filters allows low frequencies to pass while reducing or eliminating high frequencies. These reduced frequencies are the frequencies above the frequency cutoff. Figure 3.3 is a picture of a Low-Pass Filter. Figure 3.3 represents a graph that plots the amplitudes that the filter rejects and passes. The left side of the figure represents the lower frequencies.

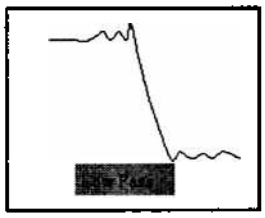

Figure 3.3

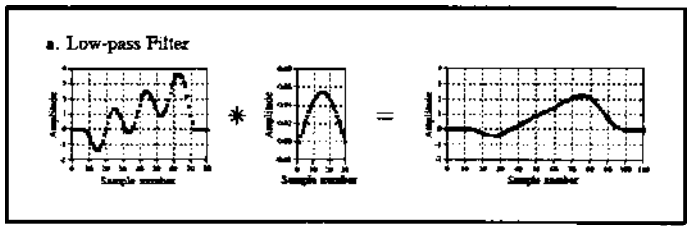

Figure 3.4 (Smith, 2003)

Figure 3.4 illustrates how a Low-Pass Filter works. The output signal is the input signal convolved with the impulse response. The impulse response is a smooth arc. The sum the arc's value equals 1. This example demonstrates how the low pass filter allows the lower frequencies through. We can see its affect in the output signal. Notice how the output signal shows the gradual increase in amplitude but not the individual phases.

An example of a filter kernel is defined Eq. 3.1 below, where *fc* is the frequency cutoff and *M* is the length of the filter. Important to note the filter length must be even. Java code for Eq. 3.1 is shown in Fig 3.5.

$$h[i] = K\frac{\sin(2\pi f_c(i - M/2))}{i - M/2}\left[0.42 - 0.5\cos\left(\frac{2\pi i}{M}\right) + 0.08\cos\left(\frac{4\pi i}{M}\right)\right]$$

where fc= Frequency Cutoff, M =Filter Length

Equation 3.1

```
private void getFilterKernel(int filterLength){
      final double M = filterLength-1;
      final double FC = .14;     //frequency cutoff between 0.0-0.5
      double n;
      for(int i = 0;i<filterLength;i++){
          if(i-M/2==0)  sinc[i]=2.0 * Math.PI * FC;
          else sinc[i]=Math.sin(2.0*Math.PI * FC * (i - M/2)) / (i -M/2);
          sinc[i]=sinc[i] * (.54 -.46 * Math.cos(2 * Math.PI * i/M));
      }
}
```

Figure 3.5

3.3 High-Pass Filters

High-Pass Filters allow high frequencies to pass while lower frequencies are reduced or eliminated. Figure 3.6 is a High-Pass Filter. The x-axis is frequency axis and the y-axis represents amplitude. Frequencies below the frequency cutoff are rejected while frequencies above the cutoff are allowed to pass.

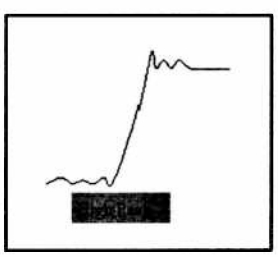

Figure 3.6

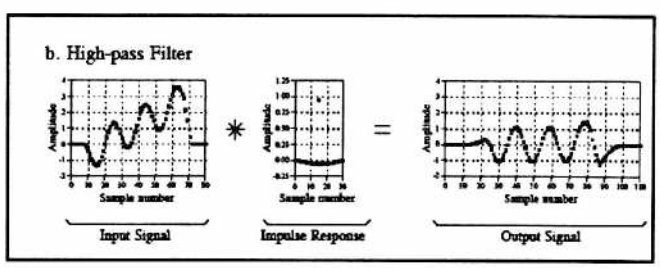

Figure 3.7

Figure 3.7 is an example of a High-Pass Filter in action. Here the impulse response is a negative dip with amplitude of 1 at point 0. This figure is very similar to a Delta Function but not an exact match. This allows higher frequencies to pass into the output signal. The output signal represents the sinusoidal nature of the original signal. This contrasts with the low pass filter that omitted this information but retained the increase in amplitude.

Creating High-Pass Filters can be done in two ways. The first solution is called a High Pass Spectral Inversion Filter. To use spectral inversion, create a low pass filter, filter the solution and subtract the low pass signal from the original. The second solution is called High Pass Spectral Reversal Filter. Spectral reversal takes a low pass filter and changes the sign of every other sample (Harvey, 2006). This works because changing every other sample is the same as multiplying by a sine wave with the Nyquist Frequency.

3.4 Band-Pass Filters

Band-Pass Filters are a mesh of both high pass filters and low pass filters. Band-Pass Filters allow the only a small amount of frequencies to pass. The opposite of Band-Pass is Band-Reject. Band Reject rejects frequencies in a particular range. Convolving a low pass filter with a high pass filter creates band-Pass filters. Figure 3.8 is illustrates how a band pass filter melds low and high pass filters. The top half of the figure creates the band pass affect by a cascading stages approach. The signal runs through a low pass filter then a high pass filter. The bottom half of the figure convolves the low and high pass filters. This accomplishes the band-pass in a single step.

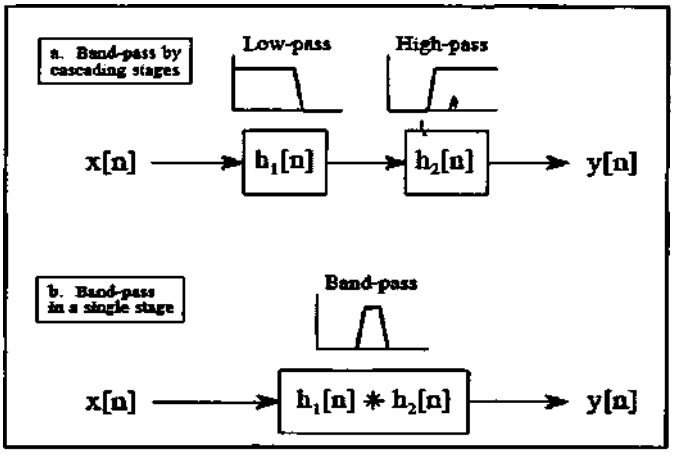

Figure 3.8 (Smith, 2003)

3.5 Moving Average Filters

Moving Average Filters are the easiest filters to understand and use. They are the most common filters in DSP (Smith, 2003). They replace a given sample with the average value of the surrounding samples. Moving average filters are very useful in DSP because they execute very fast and are effective at removing noise from a signal (Harvey, 2006). Unfortunately, long kernels lose edge sharpness and generate bounces in the frequency domain. For these reasons, moving average filters are good for use in the time domain, but not good for spectral analysis.

There are things to consider when implementing the moving average filter. Longer filters require more points at the end where quality is lost. For example, a 101-point moving average filter's 50 first points and 50 last points will not be of the same quality of the inner section of the signal. The converse of this is a 51-point filter will only have 25 less accurate points at the ends. However in comparison the inner section of the 101-point filtered will contain less noise than the 51-point filtered data.

$$y[i] = 1/M \times \sum_{j=0}^{M-1} x[i+j]$$

where x[] = Input Signal,

y[] = Output Signal, M = Filter Length

Equation 3.2

Figure 3.9 depicts two graphs. The first image is the input signal with a lot of noise. The preceding graph is the output signal, which is the input signal ran through the moving average filter. The rightmost graph shows significant noise reduction produced by applying a moving average filter. Figure 3.10 is Java code performing the moving average filter defined by Eq. 3.2.

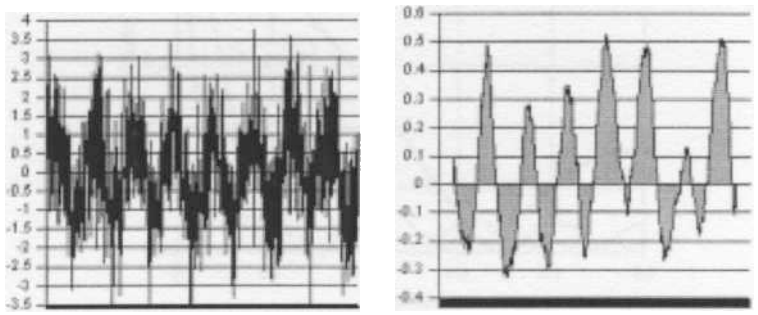

Figure 3.9

```
Int[] average(int x[]){
      int [] y[x.length];
      for (int i=50; i<x.length-50;i++){
          for(int j=50; j<=50;j++)
                  y[i] += x[i+j];
          y[i] /= 101;
      }
}
```

Figure 3.10

Moving average filters are very effective noise reducers. Its exceptional performance in the time domain compromises its performance in the frequency domain. This conclusion is based on the slow roll-off and the amount of stopband attenuation.

Slow roll-off means that transition band is very broad. The stopband are the frequencies that the filter is designed to block. Stopband attenuation is the amount of which frequencies in the stopband are reduced in amplitude (Smith, 2003). Well designed high pass, low pass and band pass filters all have very narrow transition bands. These narrow transition bands are why they do so well in the frequency domain.

Figure 3.11 depicts how much roll off and stopband attenuation is present with three moving average filters with 3, 11, and 31 points respectively. The fewer points the greater the roll-off. Fewer points also lead to greater bounce in both the passband and the stopband.

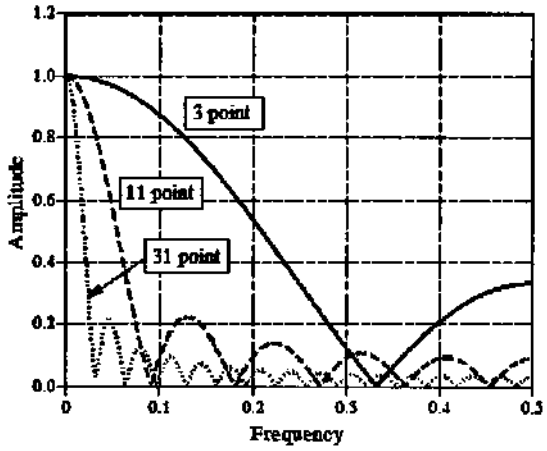

Figure 3.11 (Smith, 2003)

3.6 Preemphasis

A preemphasis filter is an example of a high pass filter. The preemphasis filter is an important module in the Sphinx front-end and the Sound Editor front-end. In audio processing there is a high amount of attenuation, and a preemphasis filter compensates for this attenuation (Carnegie Mellon University, 2004). Attenuation in audio processing is product of the dc offsets in the speech signal (Gold & Morgan, 2000). The preemphasis filter emphasizes the high frequencies while deemphasizing the low. Figure

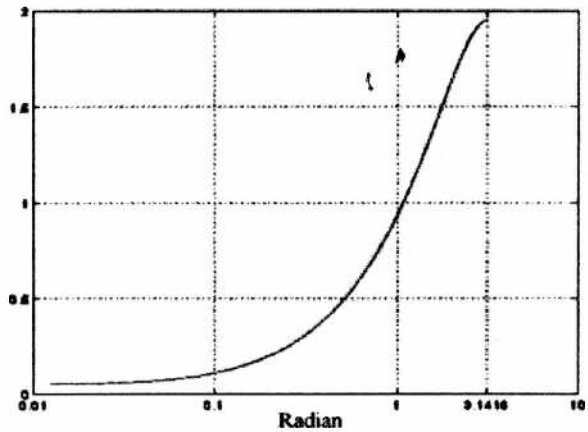

Figure 3.12 (Harvey, 2006)

3.12 is a graph showing how the higher frequencies are emphasized more than the lower. This graph is the curve used for preemphasis and is defined by equation 3.3.

$$y[i] = x[i] - (x[i-1] \times preemphasisFactor)$$

Equation 3.3

The preemphasis factor in the Sphinx front-end and the Sound Editor front-end is .97. Other ASR systems experiment with different preemphasis factors and use different preemphasis factors. Most ASR systems have their preemphasis factor stay within .90 and above range, but there are some that dip into the mid to high .80s. The Sound Editor's front-end's preemphasis is handled by the Java code depicted in Fig. 3.13

The SE processes the preemphasis algorithm within a frame. Therefore, we use the variable, *prior,* to store the value of the last sound sample from the original data stream. *Prior* then is used in the front of the next frame. It could be argued that it is easier to do the preemphasis outside the frame because we wouldn't need the prior variable. However it is more natural for Sphinx to do preemphasis within a frame because of the way the front-end is broken into a variety of data processors. It could be argued that the Sound Editor should preemphasize the entire signal as opposed to within a frame.

However, this would make it impossible to compare our results to those of Sphinx.

```
public void preEmphasizer(double prior){
    double f = SoundDefaults.getEmphasisFactor();
    //copy complex into an array to that matches complex
    //and doesn't get touched. that way we can modify the values
    //in complex with messing up the algorithm
    double [] copy = new double [complex.length];
    System.arraycopy(complex,0,copy,0,complex.length-1);
    //apply algorith y(n) = x(n)-fx(n-1)
    complex[0]=copy[0] - (f * prior);
    for(int i=2;i<complex.length;i+=2)
        complex[i]=copy[i]-(copy[i-2]*f);
}
```

Figure 3.13

Reworking the SE solution is unnecessary at this point. However this could be a useful and good contribution for a future student.

The preemphasis algorithm is a time domain filter. During our implementation, we had concerns relating to how this time domain filter alters the frequency domain. After running some experiments, we believe that the impact is minimal. This test was performed by running the front-end in the SE with and without preemphasis then examining the frequency domain. The two different approaches yielded no distinguishable change in the wave display panel. No visible change was apparent even though hand analysis of the data demonstrated some differences between the two. Hand analysis consisted of examining corresponding points of frequency domain data of a preemphasized and non-preemphasized audio signal. This test suggests preemphasis compensates for attenuation without compromising the frequency domain. Figure 3.14 shows one of the outputs of the tests we ran. In this picture we see the FFT of a 100Hz frequency. The top wave panel shows frequency domain ran with preemphasis and the bottom without. For this frequency we see no difference in the spike.

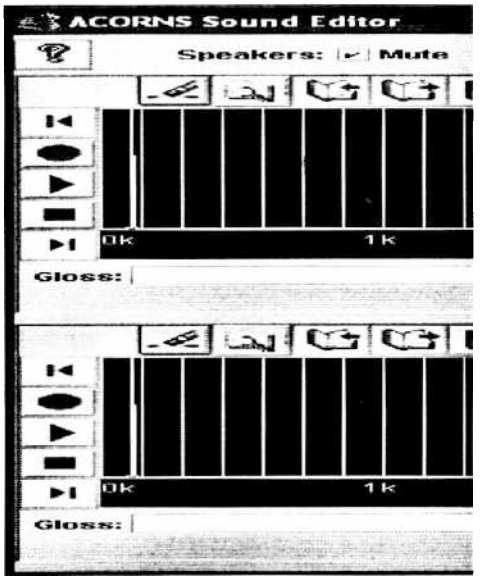

Figure 3.14

CHAPTER 4

Fourier Transform

In this section we discuss the Fourier transform. It is a crucial concept in both DSP and ASR. It is the best solution to finding the signature frequencies in a signal. This not only is important in speech recognition. It is also important to application like sonar, radar, medical imaging and more.

4.1 History and Purpose

French mathematician, John Babtiste Joseph Fourier, discovered the Fourier transform. The Fourier transform decomposes a function into a continuous spectrum of its respecting frequency components. Conversely the inverse transform synthesizes a time-domain function from its spectrum of frequency components. The Fourier transform makes it possible to decompose a signal into an infinite sum of sines and cosines used to model signal. Processing is done to these sinusoidal waves, and then recombined via the inverse Fourier transform. This is a crucial DSP concept in regards to ASR systems. Knowing the given frequencies allows us create recognizable feature vectors that map to phonemes. These recognized phonemes then can be run through a HMM state machine to recognize words. Repeating this same HMM process with words recognizes sentences.

The finite nature of computers forces us to use the Discrete Fourier Transform (DFT). The naive DFT algorithm operates at $O(n^2)$. This performance is too slow, even on modern computers. The FFT provides the same output as the DFT, but improves the complexity to $O(n \log n)$. This speed enhancement makes computing the Fourier transform a feasible in an ASR system.

4.2 Decomposition

Figure 4.1 depicts a graphical representation of DFT signal decomposition and the inverse signal recombination. The original signal on the left shows the time domain. Creating a real and a complex array decomposes the time domain into a frequency domain. The real array is an array of cosine amplitudes whereas the imaginary is an array of sine amplitudes (Smith, 2003). The IFFT takes this information and recreates the time domain. The sine and cosine waves that generate the DFT output are called basis functions (Smith, 2003). Equation 4.1 defines these basis functions. In our equations we use the electrical engineer's methodology for representing complex numbers. Mathematicians use *i* to represent the complex plane, electrical engineers use *j*.

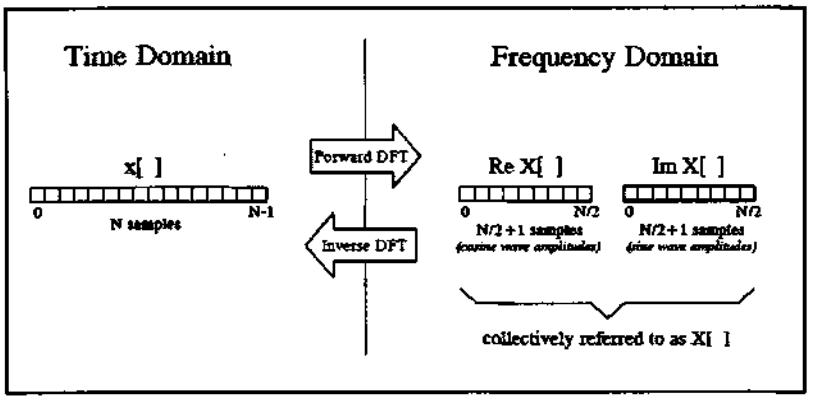

Figure 4.1(Smith, 2003)

$$c_k[j] = \cos(2\pi kj / N)$$
$$s_k[j] = \sin(2\pi kj / N)$$

where k =*frequency* & N=*length*

Equation 4.1

4.3 Complex Transform Versus Real Transform

There are two approaches to compute the DFT; the real and complex transform. The complex transform represents the entire spectrum while the real transform represents only half. While the real DFT contains real and imaginary parts, this is only done to correlate the sines or cosines in respect to the time domain. There are no complex numbers in their respective equations.

The real DFT is problematic among scientists for the following reasons. The first problem is that using substitution to map to the complex domain is mathematically unsound. Secondly, special processing is necessary at the end points. Thirdly, the real Fourier transform doesn't consider the negative frequencies. Cosine and sine wave can be described as having a positive or a negative frequency; since the two views are identical the real algorithm ignores negative frequencies (Smith, 2003). Aliasing demonstrates this problem. Aliasing occurs when two frequencies are indistinguishable. This occurs when frequencies exist that are above the Nyquist frequency. There are a number of definitions for Nyquist frequency. We define Nyquist frequency to be double the sampling rate. The Fast Fourier Transform implements the Fourier transform with O(n log n) complexity, is mathematically correct for handling complex numbers, and properly handles negative frequencies.

Euler's Formula is the tool that provides success in the complex Fourier transform. Euler's Formula is stated by Eq. 4.2. Based on Euler's Formula we arrange the formula for sine and cosine to look like Eq. 4.3. Equation 4.3 is Euler's relation that relates better to the FFT. Here we distinguish two formulas allowing us to calculate sines and cosines.

$$\boxed{e^{j\theta} = M(\cos\theta + j\sin\theta)}$$

Equation 4.2

$$\cos(\omega t) = \frac{1}{2} e^{j(-\omega)t} + \frac{1}{2} e^{j\omega t}$$

$$\sin(\omega t) = \frac{1}{2} j e^{j(-\omega)t} - \frac{1}{2} j e^{j\omega t}$$

where ω is the frequency and $-\omega$ is the negative frequency

Equation 4.3

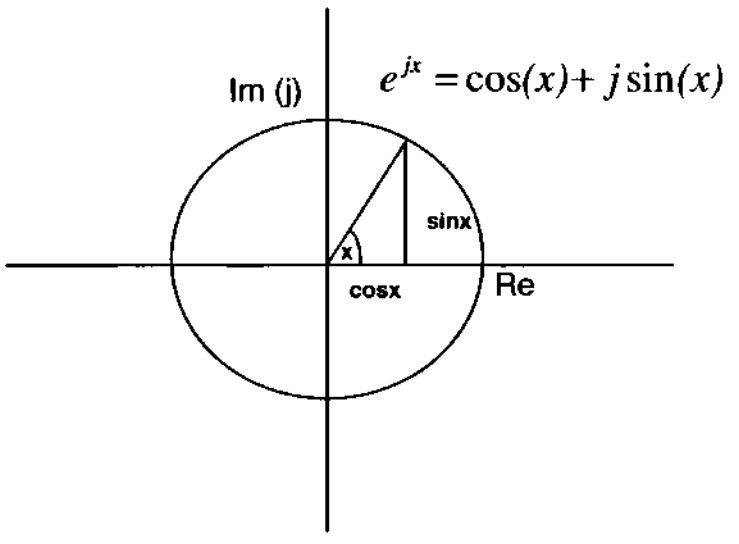

Figure 4.2

Euler's Formula, as expressed in Eq. 4.3 allows us to express sines and cosines as the sum of two exponentials. In Euler's Formula the first exponential is a positive frequency and the second is a negative frequency (Smith, 2003). It is because of these equations that we can handle the positive and the negative frequencies equally within the FFT. Also Euler's Formula is important because it makes multiplication and division easy and it saves calculations in the FFT (Harvey, 2006). For example, Eq. 4.4.

$$e^{128/64} = e^{32/64} = e^{8/16} = e^{1/2}$$
$$M_j e^{\theta 1 j} \times M_2 e^{\theta 2 j} = M_1 \times M_2 e^{j(\theta 1 + \theta 2)}$$
$$M_j e^{\theta 1 j} / M_2 e^{\theta 2 j} = M_1 / M_2 e^{j(\theta 1 - \theta 2)}$$

Equations 4.4

Euler's Formula allows an elegant solution to the deficiencies of the real Fourier transform. Euler's Formula provides a more mathematically natural way to access the complex numbers. Another advantage in using Euler's Formula in the FFT is it reduces calculations.

4.4 Implementations

We will provide various implementations for the Fourier transform in this subsection. Understanding the code is a great way to really understand the Fourier transform. Many articles, books, etc say something to the effect of, "knowing how the FFT isn't necessary if you know how to use it...so don't bother." The Fourier transform is a complicated algorithm. Understanding the coding involved in the Fourier transform however can be a great asset. For example, in our FFT we were able to make additional speed adjustments in the FFT code.

Figure 4.3 is the real DFT by correlation. It computes how the array *time[i]* affects each *frequency[k]*. As previously stated because the real DFT doesn't use the complex plane the negative half of the spectrum is lost. Also it is important because of this there are two extra points at zero and N/2.

Figure 4.4 is the method for the complex DFT by correlation. This code computes how the array *time[i]* affects each *frequency[k]*. It is also important to note that the even indices represent the real part and the odd indices represent the imaginary part. Again the complex DFT eliminates the deficiencies in the real DFT implementation. These improvements don't even require drastic alterations in the real DFT code.

```
double[] realDFT (double[] time, int N){
    double[] f[2*N+2], real, imag;
    double om, w=2 * Math.PI / time.length;
    for(k=0;k<=N/2;k++){
        for(i=0;i<N;i++){
            real=Math.cos(2*Math.PI*k*i/N);
            imag=Math.sin(2*Math.PI*k*i/N);
            f[2*k] += time[i]*real;
            f[2*k+1] += time[i]*imag;
     } }
     return freq;
 }
```

Figure 4.3

```
double[] complexDFT(double[] time, int N){
    double[] f[2*N], real, imag;
    double om, w=2*Math.PR / time.length;
    for(k=0;k<N;k++){
        for(i=0;i<N;i++){
             real=Math.cos(2*Math.PI*k*i/N);
             imag=Math.sin(2*Math.PI*k*i/N);
             f[2*k] += (time[2*i]*real-time[2*i-1]*imag);
             f[2*k+1] += (time[2*i+1]*imag+time[2*i]*real);
    }   }
    return freq;
}
```

Figure 4.4

The DFT through correlation accomplishes the tasks of breaking a time function into frequency functions. Unfortunately using correlation leads to yet another set of problems that relate to execution speed. The correlation algorithm is an $O(N^2)$ algorithm. The FFT is the optimized solution for computing the Fourier transform. By using the FFT we can complete the Fourier transform at O(n log n). It is this speed enhancement that allows scientists to compute the Fourier transform on today's processors (Harvey, 2006). Other optimization techniques exist that produce even a higher level of speed. The recursive solution to the FFT is listed under Fig 4.5.

```
Complex[] fft(Complex[] x){
    Complex[] y = new Complex[N];
    if(x.length==1) {y[0]=x[0];return y;}
    Complex []even = new Complex[N/2];
    Complex []odd  = new Complex[N/2];
    for(int i=0;i<N/2;m++){
        even[i]=x[2*i]; odd[i]=x[2*i+1];}
    Complex [] q = fft(even), r = fft(odd);
    for(int k=0; k<N/2; k++){
        double exp = -2*k*Math.PI / N;
        Complex wk = new Complex (Math.cos(exp), Math.sin(exp));
        y[k] = q[k].plus(wk.times(r[k]));
        y[k+N/2] = q[k].minus(wk.times(r[k]));
    }
    return y;
}
```

Figure 4.5

We know of three more speed enhancements that help improve upon the already efficient FFT. A simple but very effective speed enhancement is using bit shifting versus division. In Java N>>1 is ten times faster than N/2 (Harvey, 2006). Also instead of repetitively calculating sine and cosine values, using a table lookup can retrieve these values much faster. By solving an array of sine and cosines prior to its use in the FFT, clock cycles can be saved using already calculated information rather than calculating in time. The Butterfly algorithm performs the last speed enhancement. The recursive solution is an order of magnitude slower than necessary (Harvey, 2006). The Butterfly code turns two complex points into two other complex points (Smith, 2003). To accomplish this, the Butterfly algorithm flips the bits from left to right. Figure 4.6 performs the Butterfly calculation.

```
int j = N>>k, k;
for(int i=1;i<N-1;i++){
    if(i<j)     swap(x[i],x[j]);
    k=N>>1;
    while(k>=2 & j>=k){
            j -= k;
            k >>= 1;
    }j+=k;}
```

Figure 4.6

4.5 Power Spectrum

The final stage of FFT is computing the power spectrum. The power spectrum is calculated by $(\text{real}^2 + \text{complex}^2)^{1/2}$. This is the distance from the origin to the complex coordinate. It is the distance part of polar notation. The other part of polar notation is the phase, which we ignore (Jurafsky & Martich, 2000). Human hearing seems to be insensitive to phase (Smith, 2006). In Sphinx, the calculations are done within the DiscreteFourierTransform class. In the Sound Editor the FFT data is passed along to a powerSpectrum method.

Computing the power spectrum is important because we want to return real numbers not complex. So we take the square of the amplitude to eliminate the complex numbers. Remember $j = \sqrt{-1}$ and $j^2 = -1$. The power spectrum then represents the particular strength for the given frequency within in the current window. Spectral analysis and filtering then computes spectral data from the power spectrum. Figure 4.7 is the method from the Sound Editor to compute the power spectrum. In the Sound Editor the real data is stored in every even array element and the complex is stored in every odd array element. So in this implementation the length of the power spectrum is half the length of the FFT data (one power spectrum element for one real and one complex FFT element).

```
public double[] powerSpectrum(double [] fftData){
    double[] powerSpectrum  = new double[fftData.length/2];
    for(int i=0;i<fftData.length;i++){
        double real = fftData[i];
        double imag = fftData[++i];
        powerSpectrum[i/2] = (real * real) + (imag * imag);
    }
    return powerSpectrum;
}
```

Figure 4.7

CHAPTER 5

Frame Blocking and Windowing

This section describes the step prior to FFT. We described the FFT first, because it is an important pre-requisite to understanding frame blocking and windowing.

5.1 Frame Blocking

We break the captured signal into small sections for processing because applying FFT to the entire signal will yield useless data. For example, let us consider a sentence. A sentence is broken up of words and words are broken up of phonemes. By applying processing to small frames the goal is to better identify the single-most decomposed unit of speech. Without the frames this would be impossible. The frequency spectrum for human speech continually varies. This is why frames are so important. Furthermore these frames are typically sectioned off in an overlapping fashion. Slicing the frames means that we can put equal emphasis on each section of the entire signal. Without overlapping frames the boundaries would be deemphasized when applied with the windowing functions that we will describe. Figure 5.1 depicts a frame blocking sequence. Two things are important to understanding the frame blocking mechanism; these are window size and window shift. The window size is the length of the various windows. In Fig. 5.1 below *(K)* represents the window size. The window shift is the length of the shift beginning from the start of the last frame. In Fig. 5.1 the window shift is represented by *(P)*.

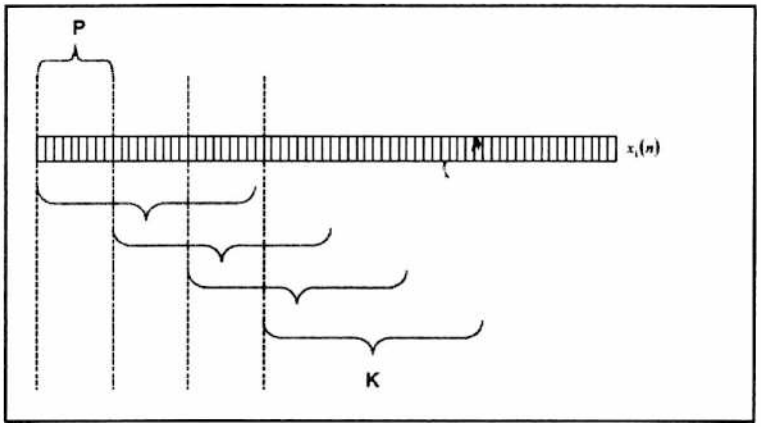

Figure 5.1 (Nilsson & Ejnarsson, 2002)

In our SE front-end, we use the same window size and shift values as Sphinx. The window size is 25.625 ms, and the window shift is 10 ms. Window shifts and sizes do vary from system to system based on experimentation, research and preference. However it is claimed that an overlap around 62% helps ensure that speech is stationary within each frame (Nilsson & Ejnarsson, 2002). A possible experiment for future SOU Computer Science students could be to verify this statement. Further verification is to determine if this statement holds up to window size and shift of Sphinx and the SE.

In the Sound Editor, the window shift and size are converted from milliseconds to number of array elements. This is done by accessing the static variables holding the shift and size values and then multiplying those values by 1000 times the sampling rate. In our version of the Sound Editor we have the same sampling rate used by Sphinx', 16000 samples per second. The original release of the SE uses a sampling rate is 22000 samples per second. In the conversion calculations we multiply the sample rate by 1000. This is done because the sampling rate is expressed in samples per second and the window shift and size are expressed in milliseconds.

Figure 5.2 shows the implementation for frame blocking. A for loop is used to iterate until the beginning of the current frame is past the end of the entire signal. The variable, *i,* is used as a counter to count the number of frames we have processed. The variable, *i,* enables the algorithm to set the value of the *start* variable. The *start* variable determines where to begin processing the signal (*start = i * wShiftInSamples.*) The

variable *wShiftInSamples* is the window shift expressed in samples in accordance with the amount of milliseconds. The variable *end* is *start* + *wSizeInSamples*. The variable *wSizeInSamples* is the window size expressed in samples in accordance with the amount of milliseconds.

```
for(int i=0, start=0, end=wSizeInSamples;
    end<timeDomain.length;
     i++,start=i*wShiftInSamples,end=start+wSizeInSamples)
```

Figure 5.2

5.2 Windowing

We apply windowing after a signal is properly framed. This process requires three steps. The first step frames the signal. The windowing function operates on these frames, and multiplies corresponding elements by a windowing equation. Hamming and Blackman are two popular windowing equations used in DSP. The Hamming window has a 20% faster roll-off than the Blackman windows (Smith, 2003). Blackman windows have better stop band attenuation than the Hamming window (Smith, 2003). Choosing the Hamming window was based purely on the fact that it is the preferred window function in Sphinx. A possible future research project could be to determine the ASR tradeoff between stop band attenuation and roll-off. Equation 5.1 defines the equations for Hamming and Blackman windows.

Hamming: $w[i] = .54 - .46\cos(2\pi i / M)$

Blackman: $w[i] = .42 - .5\cos(2\pi i / M) + .08(4\pi i / M)$

where M is the filter length

Equations 5.1

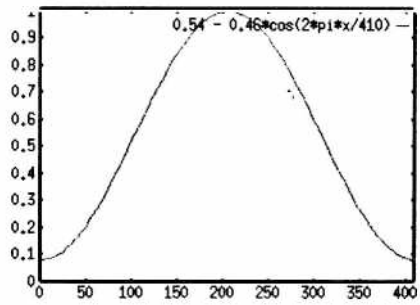

Figure 5.3 (Carnegie Mellon University, 2004)

As previously stated the Hamming window shape is a bell curve. A Hamming window is drawn in Fig. 5.3. Visually we can see, when multiplied straight across point by point the center frequencies are given a higher emphasis whereas the edge frequencies are deemphasized. Framed signal samples are more complete in the center. The samples at the end produce rough edges. Another way to think of this is one point in a frame does not say anything about the signal. However when we are given access to that one points surrounding points we can start can start to understand the signal. Boundary edges are more rough/abrupt therefore are less complete and less important to the frame. For this reason we apply the windowing function to deemphasize the boundary edges. Figure 5.4 below is a diagram showing the how frame blocking and windowing relate. First we capture a signal. After a signal is captured we frame the window for processing. Within a frame we apply preemphasis. Once we have successfully framed and preemphasized the signal data we apply a windowing function. Finally we pass this data to the Fourier transform module to decompose the signal into frequencies.

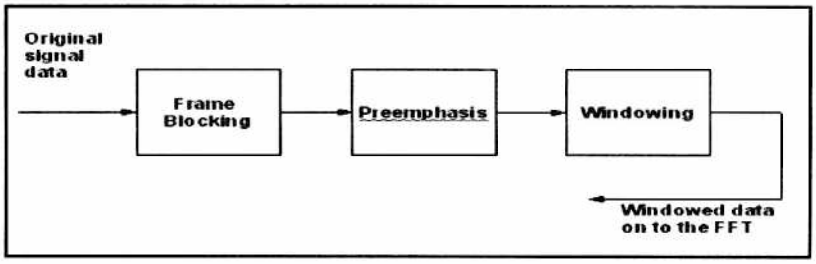

Figure 5.4

CHAPTER 6

Spectral and Cepstral Analysis

This section discusses spectral and cepstral analysis. These steps are in place to help further the likelihood of being able to determine frequencies within the frequency domain. Cepstra helps sharpen spectra and by doing this, the accuracy of recognition goes up.

6.1 Overview and Purpose

The signal that we capture from a microphone represents sound in the time domain. Unfortunately, time domain analysis does not give us enough information to accurately recognize speech. Data that is critical for ASR resides within the frequency domain. Spectral analysis helps us to examine and understand the components within frequency domain. It examines a signal by considering the amplitude, frequency and phase of its component sinusoids (Smith, 2003). Unfortunately, a raw FFT gives us frequency spectrum that is not in an easily usable form. We need additional processing to convert this data into a manageable feature vector. The purpose of Spectral analysis is to determine the dominant ranges of frequencies in a frame.

There are a number of ways to accomplish spectral analysis. The Steven Smith book tackles this approach by computing the Fourier transform over a signal via frame blocking. Next he suggests averaging adjacent frames to smooth the spectrum. A moving average filter accomplishes this step. By averaging these outputs we expect to reduce enough noise that frequency spikes are easier to distinguish.

right shows a noisy FFT signal. The box below shows a massaged frequency spectrum where key frequencies stand out.

We choose the mel scale because Sphinx uses this approach, and their application provides lots of code implementing mel filters. A Bark PLP filter bank approach could be a good project for a future SOU Computer Science student. They could easily use the same approach we describe in this paper. Sphinx already provides this modularity. The implementation would simply be a mater of researching PLP and Bark scales and using Sphinx software as a base for comparison.

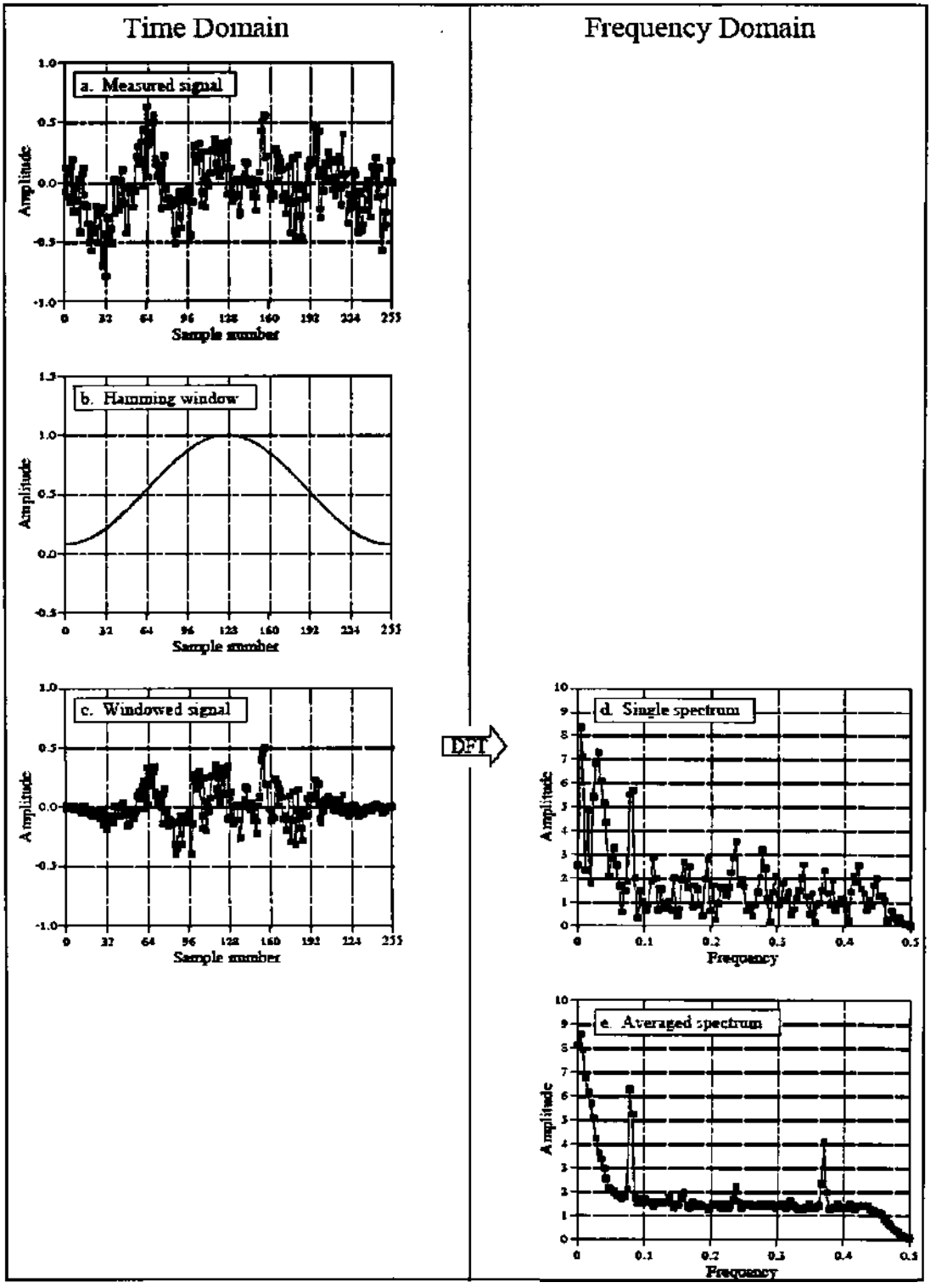

Figure 6.1 (Smith, 2003)

6.2 Spectral Analysis through the FFT

This section describes Spectral analysis as suggested in DFT (Smith, 2003). The first step in performing spectral analysis via the FFT is to record the original time domain audio signal. We next frame the signal and apply the Hamming window to each frame. This emphasizes the center of the frame and deemphasizes the boundary edges. Accomplishing this task we multiply corresponding points in the Hamming window to points in the framed audio signal. This results in a windowed signal. We arrive at a single spectrum when we take the DFT of this windowed signal.

Stopping here leaves us with a very noisy spectrum. It is virtually impossible to pinpoint the dominant frequencies within this noisy spectrum. Taking adjoining 256 point samples and creating an average for each point is one way to alleviate this problem. Frame blocking performs this step. By defining how many frames we want to average we then can compute the average of N frames. This will yield a smoother spectrum. This is true because performing the average reduces the noise and allows dominant frequencies to stand out better. This step is performed via a moving average filter. Figure 6.1 illustrates this procedure.

6.3 Mel Spectra

This section describes spectral analysis via mel spectra. This approach is popular in ASR and is the one we employed in this project. We create mel spectra by passing the power spectrum through a series of mel filters, known as the mel filter bank. Mel filters emphasize the lower frequencies, and mimics the critical bands in the human auditory system (Carnegie Mellon University, 2004). The number of mel filters in the filter bank is equal to the length of the mel spectrum. Equation 6.1 represents the mel scale of each of these triangular mel filters. Figure 6.2 shows the mel filter bank in picture form.

$$melFrequency = 2595 * \log(1 + linearFrequency / 700)$$

Equation 6.1

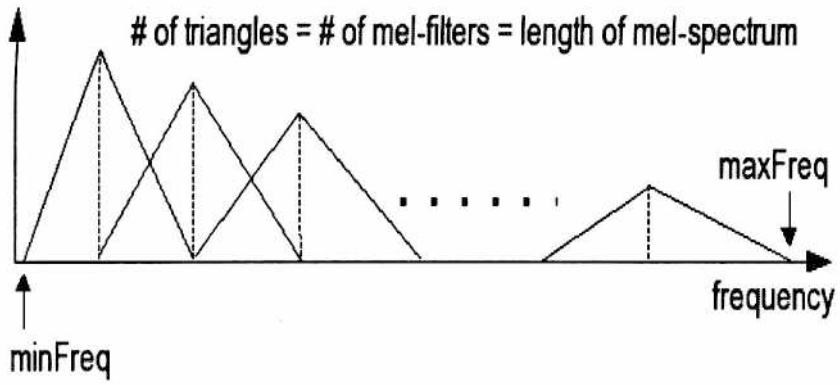

Figure 6.2 (Carnegie Mellon University, 2004)

In Fig. 6.2 we see the triangular filters are slightly skewed to right. The distance from the center frequency to the left edge is smaller than the center frequency to the right edge. These center frequencies follow the mel scale. As mentioned before, this is supposed to mimic the human critical bands. Human hearing has non-linear components. Because of this fact, so too are triangular filters in the mel filter bank (Carnegie Mellon University, 2004). Warping the frequency axis provides these non-linear triangles.

Note that the right edge always coincides with the center frequency of the following filter. This helps us implement the filter. It is easy to calculate the right and left edges of the filter, based on the center frequency. The slope is defined by these three parameters. This is because the area under any give triangle is one. First we find the triangle height. Dividing the right edge minus the left edge by 2.0 gives us the triangles height. This is how we guarantee that the area under the triangle will be one. The slope of an edge is the triangle height divided by the center frequency minus the particular edge length.

The Ben Gold book, Speech and Audio Signal Processing (Gold & Morgan, 2000) takes a different approach in building a mel filter bank. The first step is to take the log of

the frequency spectrum, called log compression. This converts the frequencies to a linear scale. The Ben Gold book mimics the non-linear human auditory system by creating filters within the first 1000 frequencies to be linear triangles. This means the first 1000 frequencies worth of triangles are equal and not changing. Figure 6.3 depicts how the mel filter bank varies as the frequency changes. Log compression is also part of the Sphinx implementation for computing mel spectra. A possible project for a future student is to implement both versions of the mel filter bank and compare and record their differences.

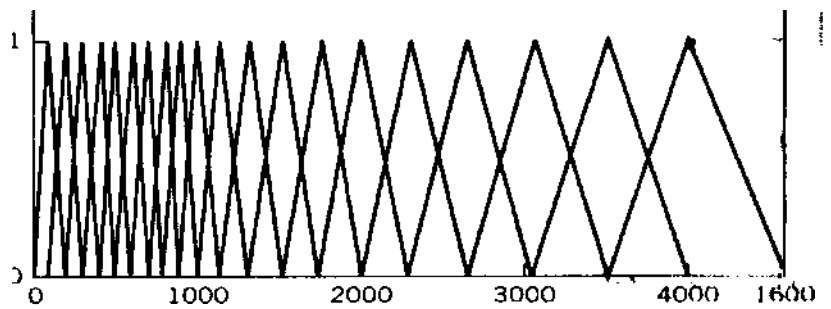

Figure 6.3 (Gold & Morgan, 2000)

6.4 Mel Cepstra and the Discrete Cosine Transform

After producing spectra that mimics the human auditory system, we need to produce cepstra. Cepstral theory states that human speech is made up of excitation frequencies and distortion (Harvey, 2006). Computing cepstra we separate the distortion and the frequencies. This makes it easier to identify formant frequencies and then map to these formant frequencies to phonemes.

To calculate mel frequency cepstrum coefficients (MFCC), we apply a logarithm then the Discrete Cosine Transform (DCT) to the mel spectrum. The log step separates the non-linear product of the speech signal from the vocal chord distortion. Now we can use standard DSP linear techniques. We can then perform an inverse Fourier transform of the logarithm of the Fourier transform.

Note that we use the DCT because our domain uses real numbers which makes DCT equivalent to the Fourier transform (Carnegie Mellon University, 2004). The output of the DCT is a MFCC vector whose length is defined by the particular ASR system. As in Sphinx, the Sound Editor produces a 13-point MFCC vector.

The first element in the feature vector contains the signal's energy. We define signal energy as the area under the squared signal. Following Sphinx' implementation we compute energy as the product of the mel cosine, the mel spectrum and a beta value, typically 0.5. The mel cosine is a two dimensional array which stores mel cosines in a filter bank. A mel cosine filter, filters spectra into cepstra. The DCT is based on Eq. 6.2 where is N the number of filter bank coefficients (Hogberg, 1997).

$$c_n = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} m_j \cos\left(\frac{\pi n}{N}(j-0.5)\right)$$

Where N is the number of filterbank Coefficients (M_j)

Equation 6.2

The following steps outline how we create spectra and cepstra:

1. Compute the mel filter bank.
2. Pass the power spectrum through the mel filters creating the mel spectrum.
3. Apply log compression to the mel spectrum. Figure 6.4 documents this process.
4. Create a mel cosine filter bank. We produce the mel cosine filter bank with the method that Fig. 6.5 outlines.
5. Run the mel spectrum through the filter bank. This creates the MFCC vector whose energy is stored in the 0th element. Figure 6.6 illustrates this process.

```
public double[] logCompression(double [] melValues){
    double [] logCompressed = new double[melValues.length];
    // in case melspectrum[i] isn't greater than 0
    // instead of trying to compute a log we just
    // assign a very small number
    for(int i=0;i<melValues.length;i++){
        if(melValues[i]>0) logCompressed[i]=Math.log(melValues[i]);
        else                logCompressed[i]=-1.0e+5;
    }
    return logCompressed;
}
```

Figure 6.4

```
private double[][] computeMelCosine() {
    double[][] melcosine = new
    double[cepstrumLength][numberOfMelFilters];
    double period = (double) 2 * numberOfMelFilters;
    for (int i = 0; i < cepstrumLength; i++) {
        double frequency = 2 * Math.PI * i / period;
        for (int j = 0; j < numberOfMelFilters; j++) {
             melcosine[i][j] = Math.cos(frequency * (j + 0.5));
     }  }
    return melcosine;}
```

Figure 6.5

```
private double[] applyMelCosine(
    double[] melspectrum, double[][] melcosine) {
        int numberOfMelFilters = SoundDefaults.getNumberOfMelFilters();
    int cepstrumLength =(int)SoundDefaults.getCepstrumLength();
    // create the cepstrum
    double[] cepstrum = new double[cepstrumLength];
    double period = (double) numberOfMelFilters;
    double beta = 0.5;
    // apply the melcosine filter
    for (int i = 0; i < cepstrum.length; i++) {
        if (numberOfMelFilters > 0) {
                double[] melcosine_i = melcosine[i];
                int j = 0;
                cepstrum[i] += (beta * melspectrum[j] *
                                        melcosine_i[j]);
                //note this loop starts at the 1st element
                //preserving the energy in the 0th element
                for (j = 1; j < numberOfMelFilters; j++) {
                        cepstrum[i] += (melspectrum[j] * melcosine_i[j]);
                }
                cepstrum[i] /= period;
        }} return cepstrum;}
```

Figure 6.6

6.5 Perceptual Linear Prediction

PLP filtering is another approach that we describe in this section. We will also describe similarities and differences between this approach and mel filtering, which we described in the previous section.

PLP, like mel filtering, has the goal to help improve likelihood of pinpointing dominant frequencies. PLP and MFCC computation are both very popular in today's ASR systems. As we already said, PLP filtering was not used in the creation of the Sound Editor front-end. However we outline the approach because of its popularity. The ACORNS project will, for the foreseeable future, focus on recognizing vowels, then consonants, and then words. We will consider PLP when our application matures to the

point where we begin to deal with larger data sets. In this section we will discuss the similarities and the differences of the two approaches.

As stated above, PLP uses the Bark scale for its filter banks using trapezoidal filters at roughly 1-Bark intervals. Equation 6.3 shows the warping function that implements the Bark scale. Transformation of frequency in hertz to the bark scale and vice versa is outlined in Fig. 6.7 and Fig. 6.8. ω is the frequency in radians per second (Gold & Morgan, 2000). Figure 6.7 shows how a linear frequency in hertz converts to the Bark frequency. The parameter, hertz, is a parameter that is passed to the method (Carnegie Mellon University, 2004). Converting Bark frequency to a linear frequency in hertz is performed by Fig. 6.8 where the parameter, bark, is passed to the method (Carnegie Mellon University, 2004).

$$\Omega(\omega) = 6\ln\left\{\frac{\omega}{1200\pi} + \left[\left(\frac{w}{1200}\right)^2 + 1\right]^{.05}\right\}$$

where ω is the frequency

Equation 6.3

double x = hertz/600

$$bark\ frequency = 6 \times \log(x + \sqrt{x^2 + 1})$$

Figure 6.7

double x = bark/6

$$hertz\ frequency = 300 \times (e^x - e^{-x})$$

Figure 6.8

With these equations we can now look into the steps of PLP filtering. Below are the steps to compute PLP coefficients (Gold & Morgan, 2000). We will note the differences between this method and mel filtering method. Figure 6.10 visually depicts these steps and how they relate.

1. Compute a power spectrum. This is done by frame blocking a signal, applying a Windowing function to the frame, performing the Fourier transform, and computing the squared magnitude.
2. Critical band integration. We place create the trapezoidal filters at 1-Bark intervals This is implemented with Eq. 6.3. This step mimics the critical bands of the human ear.
3. Pre-emphasis. Unlike the mel approach pre-emphasis is not done as the first step but is performed on the spectra. The preemphasis is placed here to place emphasis in the higher frequencies in the spectrum. This is done to help better define the dominant frequencies. We can see where the preemphasis is placed in Fig. 6.10.
4. Cube root compression. Unlike the log compression of the mel spectrum, a cube root compression is used to better fit the trapezoidal filters. This compression is useful in reducing the amplitude variance for the spectral resonances (Gold & Morgan, 2000).
5. IDFT. The inverse Fourier transform is done here just like is done in the mel approach. As previously mentioned this is the last step in producing cepstra from an input signal. As with the mel approach IFFT can be replaced with the DCT.
6. Spectral smoothing. This helps smooth out the dominant frequencies as to where no false frequencies are found. A false frequency determination illustrated in Fig. 6.9. In the first box we see 3 peaks. The second peak is a false frequency. Its removal in shown in the second box.

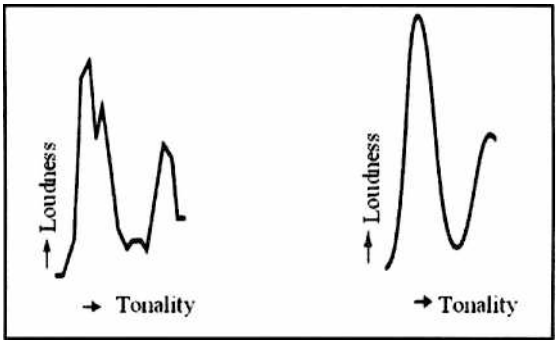

Figure 6.9 (Gold & Morgan, 2000)

In the mel approach the reduction to N amount of cepstrum coefficient, 13 in our case, is the PLP equivalent. The Ben Gold book calls this cepstral truncation; meaning higher Fourier components are not taken into consideration (Gold & Morgan, 2000). The PLP approach uses an autoregressive model, which smoothes the compressed critical bands (Gold & Morgan, 2000). Many researchers believe the autoregressive model performs better in noise robustness and speaker independence (Gold & Morgan, 2000). Linear Prediction coefficients (LPCC) are defined in Eq. 6.4.

$$c_n = -a_n + \frac{1}{n}\sum_{i=1}^{n-1}(n-1)a_i c_{n-1}$$

where a_i are the coefficients

Equation 6.4

7. Orthogonal representation. There is no mel equivalent, in fact after spectral smoothing there is no mel steps to be done. However, the PLP solution computes cepstral variables from the autoregressive coefficients. This is often performed by Durbin recursion. The Levinson-Durbin algorithm is defined by Eq. 6.4 (Carnegie Mellon University, 2004). The Sphinx Linear Predictor class uses the Durbin Algorithm.

```
x[i] = a[0] + a[1] * x[i - 1] + a[2] * x[i - 2] + ...
```

Equation 6.5

8. Liftering. Liftering is filtering in the cepstral domain (Gold & Morgan, 2000). This is the last stage of further helping our likelihood of determining the frequencies.

Its purpose is to perform a rough equivalence of variances within the different cepstrum coefficients (Gold & Morgan, 2000). Liftering implementation uses cepstral recursion.

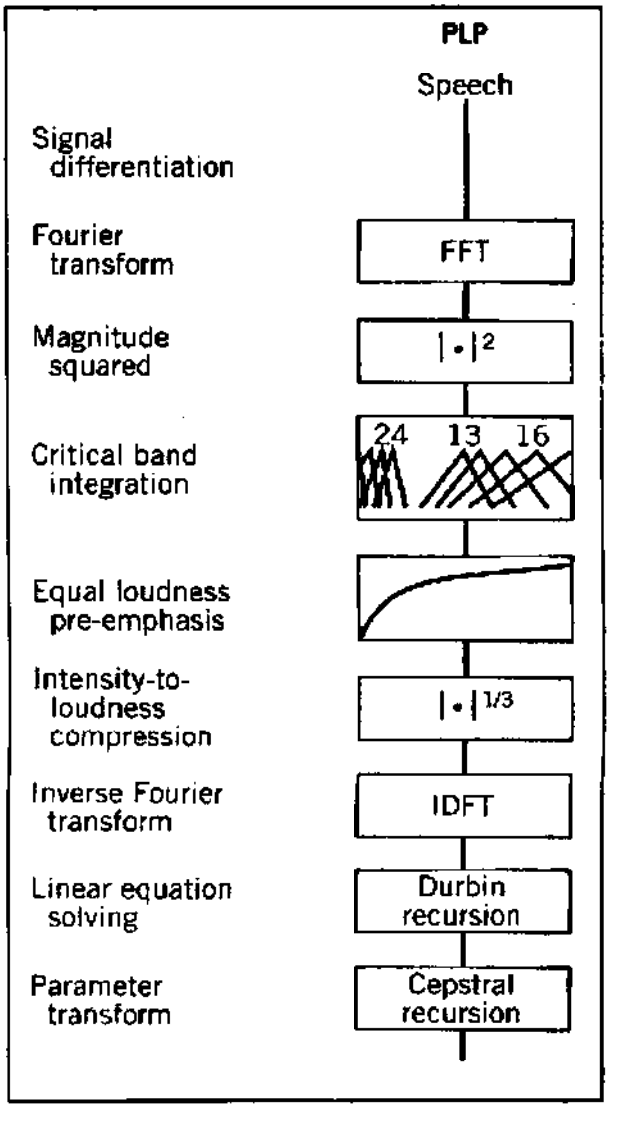

Figure 6.10 (Gold & Morgan, 2000)

CHAPTER 7

Setting up the SE Front-end

This section uses the concepts of the previous chapters to describe how to use the front end. We will discuss static variables and all the methods we need. Section 7.1 outlines these steps.

7.1 Methods Involved

Refer to Fig. 2.3 to follow the method interaction described in the section. Pressing the front-end driver button in the sound panel of the SE calls the front-end driver. This initializes the framing variables to set up our window size and window shift. Following this step we create a frame from the original signal. Processing is then done to this frame. Preemphasis is the first module executed in the front-end. The PreEmphasizer is a self-contained method in the SignalAnalysis class. It uses as input the variable *prior,* which contains the value of the last element from the last frame. It modifies the time domain sound data; stored in the *complex* array. It returns nothing.

The ApplyWindow method in SignalAnalysis executes next. This method retrieves the filter type from SoundDefaults.getFilterType(). Hamming and Blackman windows are the filter types available; Hamming is the default. Next we execute the CreateHammingWindow or the CreateBlackmanWindow method; depending on which filter type is chosen. These methods multiply sample by sample the window to the frame and stores this windowed data in a complex array. In this complex array, real numbers are stored in even indices whereas complex numbers are stored in the odd indices.

After the windowing function executes, we execute the FFT method. In the SignalAnalysis class there are two FFT methods. The first comes from the original

```
/**Emphasis factor, used by the preEmphasizer method*/
public static final float     EMPHASISFACTOR = .97f;
/**the maximum frequency for the mel filter bank 6800 is optimal
/** for a 16000Hz sampling rate*/
public static final double MELMAXFREQ =  6800;
/**the minimum frequency for the mel filter bank 130 is optimal
 * for a 16000Hz sampling rate*/
public static final double MELMINFREQ =  130;
/**the number of filters for the mel filter bank 40 is optimal
 * for a 16000Hz sampling rate*/
public static final int NUMBEROFMELFILTERS = 40;
/** The length of the MFCC vector.  */
public static final int     CEPSTRUMLENGTH = 13;
/**The length of the window in used for hamming function*/
public static final float   HAMMINGWINDOWSIZE = 25.625f;
/**The length of the offset used for hamming function*/
public static final float   HAMMINGWINDOWSHIFT = 10f;
```

Figure 7.1

release of the SE. The second method is very similar but contains modifications required by the front end driver. The difference between the two is the new versions takes in a boolean *performWindowResize* parameter whereas the original does not. This modified FFT resizes the window based on the closest power of 2 greater than the window length and pads the remaining indices with zeros. This functionality can easily be put within one method to eliminate the need for have two methods. In the current solution, the method will resize the frame no matter the value of the *performWindowResize* parameter. Our FFT algorithm utilizes the standard FFT approach and incorporates the butterfly algorithm for additional speed. It returns the FFT data. The PowerSpectrum method uses this FFT data as a parameter. It performs the power spectrum and returns the FFT power spectrum.

The remaining modules perform spectral and cepstral analysis. The first step executes the BuildMelFilterBank method. This method requires no parameters and

returns a two dimensional array containing 13 mel filters. This is done by calling the produceMelFilter which initializes variables for the left edge, right edge, center edge and Delta Frequency based on the previous mel filter. After this, the applyMelFilter method uses the mel filter bank and the power spectrum to produce the mel spectrum.

The next step is to execute the DCT. This involves calling the getMFCC method with the mel spectrum as input and then we invoke DCT processing. To accomplish DCT processing we compute a two dimensional array of mel cosines. The cosine transform consists of these mel cosines. Once we construct a mel cosine filter bank, we apply log compression to the mel spectrum. This step corresponds to the logarithm of the Fourier transform defined in section 6.4. The log compressed mel spectrum along with the mel cosine filter bank are then parameters to the applyMelCosine method. This method runs the spectra through the mel cosine filter bank. The resulting data is mel ceptra in the form of a MFCC vector. The MFCC vector is the final output of our front-end

7.2 Static Variables

Figure 7.1 outlines variables added to the SoundDefault class. These variables are specific to the front-end driver. The EMPHASISFACTOR outlined by Sphinx and incorporated into the SE front is a floating-point number of .97. We discuss this in section 3.5. We store the Hamming window and size values in HAMMINGWINDOWSIZE and HAMMINGWINDOWSHIFT. We discuss the purpose of these variables in section 5.2. The MELMAXFREQ and MELMINFREQ determine the frequencies spanned by the filter bank (Carnegie Mellon University, 2004). The sampling rate determines their values. In our front-end we set the default sampling rate to be 16000Hz. Figure 7.1 illustrates how the maximum and minimum mel frequencies relate to the sampling rate. The number of mel filters depends on the sampling rate. Our implementation requires us have 40 mel filters. This is the value stored in NUMBEROFMELFILTERS. CEPSTRUMLENGTH defines the length of the MFCC vector. We discuss mel spectra and cepstrum in chapter 6.

CHAPTER 8

Testing and Results

This section discusses the experiments we executed to verify our front-end. Our goal was to produce results that closely match those from similar Sphinx modules with Sphinx, and ultimately compare their MFCC vector with ours.

8.1 Testing Setup

We ran three different tests to check the accuracy of our front-end. Our first test performed the best. This test used prerecorded IPA vowels to produce a 13-point MFCC vector using Sphinx and in the SE. The 0th element containing the energy is particularly important. The reasons are two fold. First the 0th element is the largest in the vector. Therefore, a difference percentage is more meaningful in this element than the data with smaller magnitudes that fallow. This means the following 12 elements are more sensitive to variation. Furthermore, we compute the energy with a formula that relies on the remaining 12 elements. We don't restrict our comparison to the energy. However, we do feel that energy is a fair judge of how closely our vectors relate.

The second test we ran compares audio input from a user. In this case we recorded data from the user. These sounds were then used as prerecorded data in the two different front-ends. We compared the two resulting MFCC vectors.

The final test compares the MFCC vector of an IPA vowel to the MFCC vector of the same vowel produced by a human speaker. The SE front-end produces the vectors in both instances. This is the most interesting test because it demonstrates if human speakers can consistently produce the sound associated with an IPA vowel. This consistency is important because we plan to eventually create a dictionary based on IPA sounds. If

humans cannot consistently produce accurate IPA sounds we will need to consider alternate approaches.

8.2 Test 1 Results

Test 1 looks for differences in the MFCC vector of prerecorded IPA vowel produced by Sphinx and the SE when processing a prerecorded vowel. Table 8.1 charts a selection of these differences point by point follows by the averaged distance for each column. For example, point (0,0) from SE is subtracted from point (0,0) in Sphinx. Point (0,0) corresponds to the first element of the MFCC vector. This value is then divided by point (0,0) in Sphinx. This gives us the percentage of error between the two systems and corresponds to the point (0,0) in Table 8.1. .

These results are encouraging. We see only a 3% difference between the energy values. Some of the following values in the vector don't perform well, with 18%, 23% and 51% differences. Other values show differences of 1%, 2%, and 3%.

	Energy [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
MFCC Vectors (0 - N)	0.01	-0.04	0.16	-0.16	0.05	-0.25	3.07	13.5	0.31	-0.58	-0.78	-0.01	0.85
	0.04	0.03	-0.15	-0.48	0.24	0.47	-3.28	5.35	0.83	-0.55	0.29	1.44	0.77
	0.05	-0.01	0.36	0.73	-10.98	-0.26	1.79	-0.58	-1.3	-0.74	2.71	0.09	-0.43
	0.05	0.03	0.43	0.03	-0.55	-0.06	0.03	-0.09	-1.14	-0.7	-0.57	11.05	0.46
	0.08	0.02	-0.06	-0.29	-0.24	0.08	2.25	36.78	0.14	-2.62	2.31	-2.84	-0.16
	0.08	0.08	1.04	-0.6	0.18	-8.32	15.32	-0.69	0.1	2.58	1.61	###	1.11
	0.05	-0.09	0.1	0.07	0.64	-0.08	-78	0.48	-8.39	11.48	-3.89	0.32	-6.82
	0.06	-0.01	0.23	0.13	-0.79	-0.53	-0.15	1.34	-0.53	2.56	-9.8	1.12	3.73
AVERAGE	0.03	0.03	0.18	-0.1	-0.05	-0.13	0.23	-0.02	0.07	0.39	-0.51	1.47	0.11

Table 8.1

We hoped these numbers would match straight across. With some work we pinpointed the disparity that caused these differences. The Sphinx FFT produces slightly

different results than the SE version. There are slight rounding differences that we believe to be the cause of these discrepancies. The mel filtering compounds this difference; as does the power spectrum. As stated in section 4.5 the power spectrum is the sum of the squared imaginary and squared real values, which further increases the disparity between the Sphinx and SE values.

Nevertheless the results are promising. We believe this test demonstrates our front-end is working properly. Further analysis will lead to even better results.

8.3 Test 2 Results

The theory behind test 1 and test 2 are the same. In test 1 we used a prerecorded IPA for our audio signal and compared the MFCC vector of Sphinx and of the SE. Test 2 also uses prerecorded data. This time the data is a user recorded sound clip. We conducted this experiment using one male and one female supplying the IPA sound for the *e* sound. The user saved their best effort recording an IPA *e* sound. We used this saved vowel and ran it through the front-end of Sphinx and the SE. We expected these results to be similar to the results in test 1. Both tests compare differences between recorded data processed by the two systems. This step is important though because it will lead us into our final test.

	Energy [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
	0.01	0.36	-0.34	0.02	0.33	0.41	-0.35	-0.25	0.5	0.08	0.13	0.5	-0.09
	0.01	0.12	-0.26	-0.03	-0.98	-0.02	0.3	0.13	0.25	0.01	0.15	0.31	0.86
	0.02	0.11	-0.35	1.86	-2.43	0.21	-1.46	-0.63	-0.1	0.02	1.34	0.39	0.32
	0.03	0.09	-0.25	0.86	-2.39	-0.11	-0.86	0.18	0.7	-1.03	-0.66	0.69	-0.75
	0.01	-0.19	-0.12	0.25	-0.34	-0.27	-0.5	0.15	0.43	0.31	-0.79	-0.46	-0.14
	0.01	0	0.06	-0.02	0.25	-0.35	-0.49	0.17	-0.76	-0.4	0.17	0.1	0.52
	0.02	-0.1	-0.65	-0.54	0.12	-0.18	-0.05	-0.22	0.06	0.29	0.32	0.58	1.28
	0.04	-0.1	0	-0.09	0.18	0.18	-0.23	-0.01	-0.2	-0.51	-0.25	-0.21	0.04
AVERAGE	0.02	0.05	0.01	-0.2	0.29	-0.11	0.04	0.24	-0.04	0.01	-0.02	0.1	-0.16

Table 8.2

As expected this experiment produced results equivalent to test 1. Table 8.2 tabulates the results for a male recording of the *e* sound. Table 8.3 is a similar chart but reflects a female speaker. Both show the results of Sphinx and SE front-end processing. Again the average difference is shown in the last row. The tests results in experiment 1 are very similar to the experiment 2; as we expected.

	Energy [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
MFCC Vectors (0 - N)	0.02	0.2	-0.95	0.15	-0.22	0.15	-0.53	-0.62	0.75	0.33	1.09	0.09	-1.58
	0.02	0.09	-0.08	-0.26	-0.09	0.68	0.85	-0.18	-0.13	0.61	-0.11	-0.52	-0.2
	0.02	0.18	-0.1	0.28	-9.58	-1.19	-0.14	0.12	0.23	0.07	0.45	-0.55	0.69
	0.02	-0.13	8.33	0.16	0.42	-0.55	0.96	-0.69	-0.43	-1.02	0.18	0.46	0.44
	0.04	0.29	-0.67	0.16	-0.39	0.41	3	-0.09	-0.07	0.44	-0.15	-0.03	-0.3
	0.02	-0.08	-0.71	-0.13	-0.24	0.35	-0.17	0	-0.11	-0.11	0.75	0.19	-0.22
	0.03	0.2	-1.13	-0.5	-0.45	0.36	0.65	0.3	-0.52	0.71	1.19	-1.35	-0.87
AVERAGE	0.02	0.04	0.15	-0.03	0.92	-0.1	-0.11	-0.1	0.09	-0.01	-0.05	0.15	-0.02

Table 8.3

8.4 Test 3 Results

estimates how well humans can consistently recreate distinct IPA vowels. Originally we believed an accepted prerecorded IPA library of sounds could serve as a dictionary for the back-end processing. Test 3 demonstrates that this strategy might have to be reworked. This experiment compares a male's recording of a vowel to that of an IPA recording. Our prerecorded IPA vowels are vowels were produced from a credible IPA website. IPA sounds are downloadable from http://www.phonetics.ucla.edu.

We averaged three samples for the male speaker and compared it to the prerecorded IPA sound. This experiment indicates our originally strategy for dictionary creation is not as feasible we would have hoped. The alternative to this approach is to create a dictionary using user-produced sounds. This is current approach of many ASR systems including Sphinx.

The male submitted data didn't correlate well to the downloaded IPA sounds. It is important to note that our downloaded IPA sounds also reflect a male speaker. The average row at the end of Table 8.4 shows 10 of 13 columns had difference rates of 17% or higher. Simply looking at the data we can see significant differences between data our users recorded and the IPA prerecorded data. The energy of the first frame in the prerecorded MFCC vector is 12.87. Nowhere within the user-supplied data does the energy drop so low. The remaining frames of the MFCC vector of the prerecorded data are within a range of 7-9 roughly. Again these numbers are very low compared to the average the MFCC vector from a male submitted vowel.

	Energy [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
MFCC Vectors (0 - N)	-0.28	0.83	0.13	0.11	0.17	0.06	-6.11	17.66	0.71	3.02	-0.11	0.58	1
	-1.1	0.68	-1.35	0.54	0.43	-1.88	-6.33	-7.7	4.34	-0.02	-0.19	-2.65	0.77
	-1.07	0.6	-0.11	1.27	5.49	-0.26	-2.06	-12.03	-5.02	-5.4	1.73	0.09	-0.07
	-0.99	0.59	-1.05	0.71	-1.44	-1.46	0.2	0.14	21.9	-2.09	-2.69	-1.79	6.82
	-1.16	0.56	-0.19	0.23	-2.1	-1.13	-5.41	-36.77	2.08	-3.65	4.68	-2.22	-0.06
	-1.14	0.69	-0.04	0.3	0.44	9.48	###	-0.13	-2.52	-3.75	-4.06	###	-0.52
	-1.09	0.56	-0.64	-0.39	-0.88	-1.34	-1.41	-0.63	-24.4	-19	7.29	3.94	2.65
	-1.02	0.69	0.15	0.51	0.03	-0.16	-0.69	0.32	-0.53	-6.02	-9.31	-0.27	-3
AVERAGE	-0.12	0.25	-0.72	0.3	-0.32	-0.17	0.68	-0.22	-1.56	-0.32	-1.8	0.74	0.14

Table 8.4

Comparing this data is very difficult to do. It is very difficult to produce two sounds that will have the same amount of frames created from it. For example, suppose we choose to compare the *e* sound from different samples; this is similar to our experiment. Now let's say sample *a* is 75 frames longer than sample *b*. A difference as large as 75 is not unusual. Seventy-five frames is only a fraction of one second. It could simply be a drawl when the vowel is spoken that produces the 75 extra frames. We can only compare data as long as sample b has data to compare. When we run out of data in sample b it is natural to get somewhat conflicting data. The center frames are more likely

to match similar parts of the utterance as compared to the end or the beginning. The end and the beginning is where we see the most change in the utterance's signal. This is why we believe that our results actually might be closer than what the experiments show.

This experiment does have positive results. If we ignore the lower energies at the beginning and end of the utterance, we see that most energies within the middle frames have energies of approximately 19 whereas the average of the 3 samples provided by the male user was 18.78. This is promising. Perhaps the results would be better if recordings were done in less noisy environments with trained professionals. In this case, we would expect that pauses before and after the utterance of the IPA vectors would reflect low energy because there would be less background noise. A comparison algorithm could then ignore noise and on the middle frames, which compare better in our experiment.

8.5 Test Conclusions

We believe that these tests are very promising. Test 1 and test 2 shows how our front-end produces very similar results when compared with Sphinx. We also discussed with making adjustments to our FFT method we perhaps could mimic the Sphinx results outright. We believe test 1 and test 2 show how ready this front-end is ready to be utilized by a proper back-end. With our current results and knowledge of current disparities using this front-end within an ASR system seems very promising.

Test 3 enlightens us about future considerations we may have been overlooking as well as gives us concerns for our previous strategy of dictionary creation. We know that a user supplying sound introduced to the system will have a very difficult time producing data that would fit within the bounds of the prerecorded IPA data. However we know that our center frames, the frames more important to the recognition, match better. We can draw two conclusions from this. The first being that using user submitted IPA sounds might be ideal to use for dictionary creation. We say this because our test sample, even though were female and male, were closer when comparing MFCC vectors than did the prerecorded IPA vowel. Using these sounds will reduce the amount of variance between users submitting to the dictionary. Variance will happen but if we allow a certain amount

of variance centered on a known sound we believe we can make better recognitions. Also if we can devise a system of better pinpointing the center frames within a spoken sound we believe that this IPA produced dictionary could still work. This being said user created dictionaries are popular and done be Sphinx.

Making user-independent ASR systems are very difficult because of such considerations. Things such as drawls, the clarity at utterance beginnings and ends, and clarity between sounds all play factor in producing a MFCC vector that closely matches another. These considerations will be looked into heavily upon research and development of the Sound Editor back-end.

CHAPTER 9

Conclusion

We have produced a correctly functioning front-end. This paper is a self-contained work allowing anyone to reproduce our results. Future Southern Oregon University Computer Science students will be able to use this works as a foundation for our back-end research. We are confident in this statement because the test results in chapter 8 prove our front-end that our front-end is comparable to Sphinx.

In this paper we describe the concepts needed to fully understand our front-end. For example, we showed how digital filters drive front-end processes. Illustrated in chapter 4 we see how to turn the time domain into the frequency domain and why do this is important. Chapter 5, the windowing and framing chapter, shows how we prime the time domain data in a way that will be useful to the FFT. Chapter 6 shows how the power spectrum of the FFT is used in spectral analysis as well as explaining how and why we use spectra and cepstra to help further pinpoint dominant frequencies within the frequency domain. Chapter 7 puts these concepts and describes how all these modules work as a unit.

This paper's intent is for future students to use as a guide to help further their knowledge about DSP and speech recognition. Our hope is that computer science students at SOU stay interested in this topic and continue the work started by this project. We are confident that this paper lays a good foundation for front-end principles. We are also confident that a lot more work can be done. We point out throughout the paper future enhancements we consider to be important. This paper, Sphinx and the ACORNS documentation will help future students stay inspired to improve the ACORNS project.

CHAPTER 10

Continuing the Project

This section describes possible enhancements and areas of consideration that future students can use to further improve front-end performance. We also discuss how Hidden Markov Models will drive back-end functionality.

10.1 Front End Enhancements

Most of the front-end enhancements we describe in this section have been touched upon throughout this paper. In this section we provide a more complete discussion for future work on this project. In section 3.5 we discuss cleaning up the preemphasis. We use a *prior* variable to store the last array element of the previous array, which will be used in the current arrays analysis. As previously mentioned, the reason to overlook this minor inefficiency was conform to Sphinx's design and to not fix a broken thing. Here though we believe this architecture does not fit the architecture of the SE and therefore this could be an opportunity to contribute a simple yet important enhancement to this project.

Section 5.1 presents an interesting question concerning the optimal percentage overlap in windowing. The claim states a wrap of 62% helps ensure that speech is stationary throughout the signal even though framing and windowing have taken place. It would be very interesting if this statement could be verified. If this in fact were true, altering our window size and window shift would not be very difficult. It too would be interesting to see what the affects will have throughout the front-end modules and on the resulting MFCC coefficients. This is an interesting study because its change would happen in the time domain and prior to any other front-end modification. Understanding its affects could help us frame our windows better.

The Hamming window is the implemented window in Sphinx. Whereas the Hamming window has a 20% faster roll-off than the Blackman window the Blackman window does have better stop band attenuation (Smith, 2003). In chapter 5 we suggest that this statement be analyzed. Is the Hamming window used in Sphinx and other ASR systems because of the faster roll-off, even though the stop band attenuation is worse? In windowing is roll-off more important than stop band attenuation? Answering this question can help us understand why the Hamming window is used in Sphinx or any other ASR as opposed to the Blackman window.

Chapter 6 we discuss spectral and cepstral analysis. We also site a reason for choosing the mel scale approach for the mel filter bank as opposed to the Bark scale in the PLP filter bank. Sphinx plug-in implementation in both and having PLP filter bank. Our framework could easily provide capabilities for both approaches. This would not be much work, yet would add flexibility. This added capability would allow us to verify our decision to use the mel scale to produce spectra and cepstra. Once a back-end is in place, testing can be done to see if one performs better for certain data sets or data sizes. More importantly experimentation can be done in creating a hybrid model with both mel and PLP filtering. Ben Gold and Nelson Morgan discuss this approach. They state that research indicates that particular hybrid approaches yield better results than when mel or PLP filtering are used by themselves.

We can also perform spectral experiments using different types of mel filter banks. Figure 6.2 and Fig. 6.3 show two possibilities. The first mel filter bank in Fig. 6.2 uses the Sphinx design. This implementation uses a non-linear scale throughout the entire filter bank. Each filter is based on the previous with the right slope of each triangular filter warped along the frequency axis. Looking at figure 6.2 we can see how the right edge of any triangular filter is farther from the center than its left edge. It is this warping that gives us non-linearity, which is supposed to mock the human auditory system. Figure 6.3 creates a linear filter bank up to 1000Hz and the following filters are created logarithmically. We wonder how much these two implementations will affect the outcome in the mel spectrum. Also, will these differences in the mel spectrum ultimately

affect our MFCC vector. As we discussed section 8.3 about test 3 we know that slight changes in the output of one module can create larger changes down the front-end chain. For example, slight changes in the output of our FFT in the SE versus Sphinx escalate as the data passes through the remaining front-end modules. Our original implementation closely follows Sphinx design. Now that we have a functioning front-end we can ask these critical questions to make our system unique and possibly suggest better solutions and a better front-end

10.2 Post Processing

Post-processing is any process that is done to the MFCC. These steps finalize our MFCC and primes the data for use by the back-end (Nilsson & Ejnarsson, 2002). Post-processing usually involves weighting the data, normalizing the data. In the case of Sphinx, post-processing computes the delta and double deltas from the original 13-element vector. Weighting the data may be important if we know that the first few elements of a feature vector will contain low frequency noise (Nilsson & Ejnarsson, 2002). Weighting down the first few elements might be necessary so these elements do not have an adverse affect on the remaining elements. Weighting can be achieved with Eq. 10.1.

$$\boxed{\begin{aligned} &x_3'(n;m) = x_3(n;m) \cdot w(n) \\ &n = 0,2,\ldots,N-1 \end{aligned}}$$

Equation 10.1

We have found no evidence that this step is a part of Sphinx. Perhaps adding a weighting function to the data may not be necessary, but this could be a useful feature. Further research could be done on this topic to see if it would applicable to the SE front-end.

Cepstral mean normalization is a process done by Sphinx to reduce any transmission channel distortion (Carnegie Mellon University, 2004). This is done by calculating the cepstral mean and subtracting the mean from the cepstrum. The mean is where the distortion resides. The cepstrum is the Fourier transform of the log spectrum. Here the logarithm turns the non-linear multiplication of signals into a summation. Averaging over time the mean represents the channel; which is constant (Carnegie Mellon University, 2004). Hence by removing the mean we remove the channel distortion. Cepstral mean normalization is a module that has been implemented in our project. It however is not tested and therefore not called in the front-end driver. We feel confident that this suffices in our release because the back-end might dictate how we should implement this step.

Sphinx and many other ASR systems compute delta and delta delta coefficients, also call double deltas. These are the first and second order derivatives of the original 13 point MFCC vector. This post processing procedure is done after normalization. The first and second order derivatives help model the speech signal (Carnegie Mellon University, 2004). In doing this we turn a 13 element MFCC vector into a 39 point feature vector. Figure 10.1 demonstrates this.

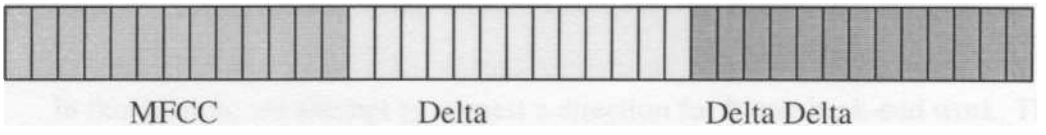

Figure 10.1

Calculating the delta and double delta can be done with the equations in Equation 10.2. Here we see that the delta is equal to the current position plus two minus the current position minus two. Calculating the double delta is done by subtracting the current position, plus one, minus the current plus three, by the current position plus three minus the current position plus one. We show this pictorially in Fig. 10.2.

$$delta = (current + 2) - (current - 2)$$

$$double\ delta = [(current + 3) - (current - 1)] - [(current + 1) - (current - 3)]$$

Equation 10.2

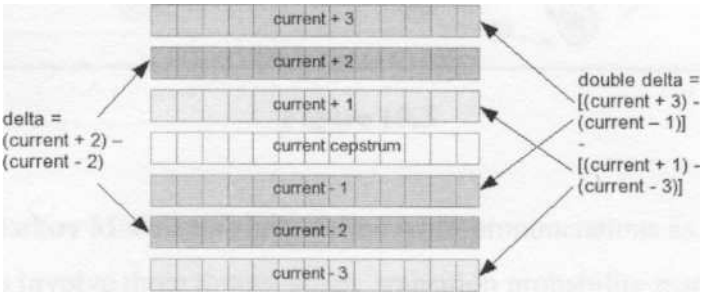

Figure 10.2 (Carnegie Mellon University, 2004)

ACORNS research should consider this procedure in the future. As Sphinx states, this procedure better models a speech signal. Research also should investigates how this relates to our goal to create an IPA based dictionary.

10.3 A Quick Look Into the Back-end

In this section, we attempt to suggest a direction for future back-end work. The engines that drive the back-end are Hidden Markov Models (HMM), which were briefly discussed in section 1.1. A HMM is a finite state machine that based on an output, finds the highest probable corresponding input (Jurafsky & Martich, 2000). Figure 10.3 shows a pronunciation HMM for the word "about." In this example we observe the pronounced phonemes. Based on the probabilities defined in the model, we hope to determine if the given speaker intended to pronounce the word "about."

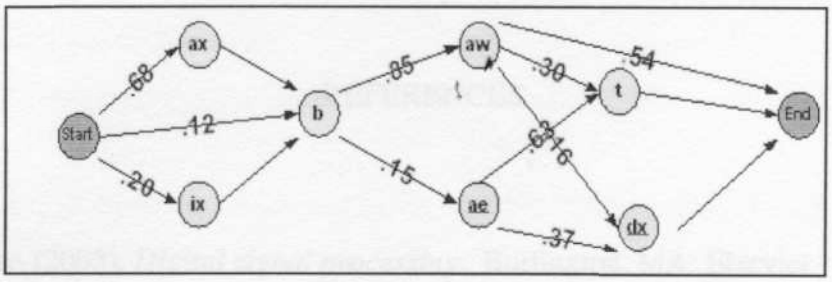

Figure 10.3

Hidden Markov Models can help define word pronunciations as well as sentence grammar. HMM's involve three things: states, transition probability matrix and observation likelihoods (Jurafsky & Martich, 2000). The observed outputs determine the initial state. The probability matrix defines the probabilities of the possible next states based on the current state. A particular language dictionary contains the probability matrix. These probabilities are updated as the dictionary receives new words and becomes more complex. Dictionaries with more words lead to more complex HMM's. Observation likelihoods are similar to state transition probabilities. They express the observation likelihoods based on the occurrence of a different state (Jurafsky & Martich, 2000).

HMMs are a very useful aide in back-end creation because of their versatility. Sphinx and many other ASR systems use the HMM approach. ACORNS back-end research will investigate HMM post processing functions implementations so we can develop of our own HMMs. After we create HMM classes for pronunciation and grammar, we will have a fully functionally ASR system.

REFERENCES

Smith, Steven (2003). *Digital signal processing*. Burlington, MA: Elsevier

Gold, Ben and Morgan, Nelson (2000). *Speech and audio signal processing*. New York: Lehigh Press

Carnegie Mellon University (1999-2004). *Sphinx-4 A speech recognizer written entirely in the Java programming language*. Retrieved from the web September 2006: http://cmusphinx.sourceforge.net/sphinx4/

Jurafsky, Daniel and Martin, James H. (2000). *Speech and language processing*. Upper Saddle River, NJ: Prentice Hall

Cheng-Hsueh, A., Contie, M., Johnson, T., Gyllenhaal, J., & Wen-mei, H. (1997). *"Optimizing NET Compilers for Improved Java Performance"*. IEEE Vol 30, Issue 6, (1997), pp. 67-75.

Harvey, Dan (2006). *Audio Processing*. Presented in CS582 lectures at Southern Oregon University.

Seltzer, Michael (1999). *"Sphinx III Signal Processing Front-end Specification"* Technical report Carnegie Mellon University. Available at: http://www.cs.cmu.edu/~mseltzer/sphinxman/s3_fe_spec.pdf

Binu, Michael (2004). Sphinx3 (). Front-end. In *The Perception Processor* (Section 4.1). PhD Thesis. Available at: http://www.cs.utah.edu/~mbinu/perception/node22.html

Nilsson, Mikael & Ejnarsson, Marcus (2002).. In *Speech Recognition Using Hidden Markov Models.* From Speech to Feature Vectors (Chapter3). Masters Thesis, Blekinge Institute of Technology. Ronneby, Sweden. Available at: http://www2.hh.se/staff/maej/publications/MSc%20Thesis%20-%20MiMa.pdf

Hogberg, Jesper (1997) *"Prediction of format frequencies from linear combinations of filterbank and cepstral coefficients"* KTH, Stockholm, Sweden. Speech Transmission Lab., Quarterly Progress and Status Report.

Nuance NaturalSpeaking9 (2006) Viewed on the web September 2006: http://www.nuance.com/naturallyspeaking/home

Voice XML (2006) Viewed on the web September 2006: http://www.voicexml.org