

# **A Voice Controlled Window Manager for the X- Windowing System**

## **Author**

David Mel Airlie

9312617

## **Supervisor**

Dr. Colin Flanagan, University of Limerick

## **Course**

B. Eng. Computer Engineering

Submitted in part requirement for final year project course CE4907/CE4908 to

University of Limerick, Ireland

25<sup>th</sup> April 1997

## **ABSTRACT**

A Voice Controlled Window Manager for the X-Windowing System

David Mel Airlie

This report describes work carried out on designing and implementing a Voice Controlled Window Manager for the X Window system under the Linux operating system. It describes a couple of different speech recognition algorithms and implements a dynamic time warping system using an LPC front-end. It discusses X programming and the FVWM window manager which was extended using its module system to implement this project.

## Declaration

This report is presented in partial fulfilment of the requirements for the degree of  
Bachelor of Engineering ( Computer )

It is entirely my own work and has not been submitted to any other University or higher education institution, or for any other academic award in this University. Where use has been made of the work of other people, it has been fully acknowledged and referenced.

Signature:

---

David Mel Airlie

25<sup>th</sup> April 1997

## **Dedication**

To my parents, for sending me to college, and buying me the computer.

## Acknowledgements

In no particular order, Dr. Colin Flanagan, for taking this project on and giving guidance on it, Ivan Griffin, Steve Bergin and Eamonn McGuinness for giving me access to some almighty powerful hardware, my house-mates, John, John, Ed, Cathal and Niamh for making me watch TV instead of doing work and also for crashing skynet while I'm working, Caolan and Martin for giving me someone to talk to at 4 o'clock in the morning, Nicky, Dave, Darryl, Eddie, Lisa, Keith, Garret and many others for dragging me out on the beer instead of letting me work, and Nav in the stables for serving me ahead of everyone else. Everyone on the skynet admin team for dealing with my rants and to everyone I've forgotten and last but not least John Quinn and Linus Torvalds, one for giving UL and me specifically Linux and the other for giving the world it.

# Table of Contents

<b>ABSTRACT</b>	<b>ii</b>
<b>Declaration</b>	<b>iii</b>
<b>Dedication</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>Table of Contents</b>	<b>vi</b>
<b>1. Introduction</b>	<b>1</b>
<b>1.1. BACKGROUND</b>	<b>1</b>
<b>1.2. AIM</b>	<b>1</b>
<b>1.3. BRIEF DESCRIPTION</b>	<b>2</b>
<b>2. Speech Recognition System Theory</b>	<b>3</b>
<b>2.1. INTRODUCTION</b>	<b>3</b>
<b>2.2. TYPES OF RECOGNITION SYSTEMS</b>	<b>3</b>
2.2.1. Speaker Independent/Dependent Systems	3
2.2.2. Small/Medium/Large Vocabulary Systems	4
2.2.3. Isolated/Connected Word/Continuous Speech Systems	4
<b>2.3. METHODS USED IN SPEECH RECOGNITION</b>	<b>4</b>
2.3.1. Template Based	5
2.3.2. Knowledge Based	5
2.3.3. Stochastic Based	5
2.3.4. Connectionist Based	5
<b>2.4. SIGNAL PROCESSING IN SPEECH</b>	<b>6</b>
2.4.1. Linear Predictive Coding	6
2.4.1.1 The LPC Model	6
2.4.1.2. LPC Analysis Equations	8
2.4.1.3. The Autocorrelation Method	10
<b>2.5. TEMPLATE VS. STOCHASTIC</b>	<b>11</b>
2.5.1. Dynamic Time Warping	11
2.5.1.1. Time Alignment	11
2.5.1.2. Dynamic Time Warping	12
2.5.1.3. Distance Measures	13
2.5.1.4. Basic DTW approach	14
2.5.2. Hidden Markov Models	16

2.5.2.1. Discrete-Time Markov Processes	16
2.5.2.2. Hidden Markov Models	18
2.5.2.3. Coin Tossing Model	18
2.5.2.4. Urn-and-Ball Model	20
2.5.2.5. Elements of a HMM	21
2.5.2.6. The three basic problems for HMMs	22
2.5.2.7. Types of HMM	23
2.5.3 Comparison	23
<b>3. X Windowing System</b>	<b>24</b>
<b>3.1. INTRODUCTION</b>	<b>24</b>
<b>3.2. X WINDOW BACKGROUND</b>	<b>24</b>
3.2.1. What is X?	24
3.2.2. History of X	24
3.2.3. Client-Server Architecture	25
3.2.3.1. Client	25
3.2.3.2. Server	25
3.2.3.3. Connection	25
<b>3.3. PROGRAMMING FOR X</b>	<b>26</b>
<b>3.4. WINDOW MANAGEMENT</b>	<b>26</b>
<b>3.5. FVWM</b>	<b>27</b>
3.5.1. What is fvwm and what does it stand for?	27
3.5.2. The .fvwmrc file	27
3.5.3. Fvwm Module System	27
3.5.3.1. Security	28
3.5.3.2. Interface to and Initialisation of modules	28
3.5.3.3. Module-to-FVWM Communication	29
3.5.3.4. FVWM-to-Module Communication	29
3.5.3.5. Finding Current Window Information from Fvwm	30
<b>4. Analysis and Design</b>	<b>31</b>
<b>4.1. GOALS AND APPROACH</b>	<b>31</b>
4.1.1. Speech Recogniser	31
4.1.2. Window Manager Interface	32
<b>4.2. ANALYSIS OF SPEECH RECOGNISER</b>	<b>33</b>
4.2.1. Sound Device Object	33
4.2.2. LPC and Matrix Objects	33
4.2.2.1. Structures needed for LPC processor	35
4.2.3. Single Word Object	36

4.2.4. DTW object	36
4.2.5. Most Likely Word Object	37
<b>4.3. ANALYSIS OF TRAINER</b>	<b>37</b>
<b>4.4. ANALYSIS OF WINDOW MANAGER INTERFACE</b>	<b>37</b>
<b>4.5. OVERALL SYSTEM</b>	<b>39</b>
<b>5. Development</b>	<b>40</b>
5.1. PROGRAMMING LANGUAGE	40
5.2. OPERATING SYSTEM	40
5.3. DEVELOPMENT ENVIRONMENT	41
5.4. COMPILATION SYSTEM	41
<b>6 Implementation</b>	<b>43</b>
6.1. INTRODUCTION	43
6.2. SPEECH RECOGNISER	43
6.2.1. Programming the Linux Sound Device	43
6.2.1.1. /dev/dsp	43
6.2.1.2. /dev/mixer	44
6.2.1.3. OSS API	44
6.2.2. Linux Soundcard Object	44
6.2.2.1. Object Constructor	44
6.2.2.2. Object Destructor	45
6.2.2.3. Fill Block Method	45
6.2.2.4. Microphone controls	45
6.2.2.5. Reset Buffer	45
6.2.3. Matrix Object	45
6.2.3.1. Constructor	45
6.2.3.2. Destructor	46
6.2.3.3. Resize method	46
6.2.3.4. Operator Methods	46
6.2.3.5. Durbin Solving Methods	46
6.2.4. LPC object	46
6.2.4.1. Constructor	46
6.2.4.2. fill Method	46
6.2.4.3. Cepstrum Method	47
6.2.4.4. normeqn_ac Method	47
6.2.4.5. Miscellaneous Methods	47
6.2.4.6. norm_s2d Macro	47

6.2.5. sword Object	47
6.2.5.1. Constructor	47
6.2.5.2. fillword Method	48
6.2.5.3. updatedisp Method	48
6.2.5.4. createlpcs Method	48
6.2.5.5. Miscellaneous Methods	49
6.2.6. dtw Object	49
6.2.6.1. Constructor	49
6.2.6.2. Destructor	49
6.2.6.3. init Method	49
6.2.6.4. Distance Methods	49
6.2.7. mlw Object	50
<b>6.3. WINDOW MANAGER INTERFACE</b>	<b>50</b>
6.3.1. Modules under C++	50
6.3.2. Using lex for converting string to enum	50
6.3.3. Translator Object	51
6.3.3.1 Constructor	51
6.3.3.2. Destructor	51
6.3.3.3. Setappwin and GetCurWinInfo Methods	51
6.3.3.4. Action Methods	51
6.3.3.5 cancel and proceed Methods	52
6.3.3.6. direction Method	52
6.3.3.7. command Method	52
<b>6.4. COMBINED SYSTEM</b>	<b>52</b>
6.4.1. Combining System	52
6.4.2. X Microphone switch	53
6.4.2.1. Pipe Protocol	53
6.4.2.2. xsword Object	53
6.4.2.3. doNonX() Function	54
6.4.2.4. doXt() Function	54
<b>6.5. TRAINING SYSTEM</b>	<b>55</b>
6.5.1. GUI	55
6.5.1.1. File Menu	55
6.5.1.2. Buttons	56
6.5.1.3. Help Menu	56
6.5.2. Tcl/Tk to C/C++	56
6.5.3. train_individ Function	56
6.5.4. train Object	56
6.5.4.1 Constructor	57

---

6.5.4.2. createwords Method	57
6.5.5. teltrain Object	57
6.5.6. train_all Function	57
<b>6.6. OPTIMISATIONS</b>	<b>57</b>
<b>7. Future Development and Conclusions</b>	<b>58</b>
<b>7.1. INCREASED VOCABULARY</b>	<b>58</b>
<b>7.2. USE OF HIDDEN MARKOV MODELS</b>	<b>58</b>
<b>7.3. PERFORMANCE</b>	<b>58</b>
<b>7.4. MULTI-USER</b>	<b>59</b>
<b>7.5. CONCLUSIONS</b>	<b>59</b>
<b>8. References + Bibliography</b>	<b>60</b>
<b>8.1. REFERENCES</b>	<b>60</b>
<b>8.2. BIBLIOGRAPHY</b>	<b>61</b>
<b>Appendix A. Installation Guide</b>	<b>62</b>
A.1. Unpacking	62
A.2. Configuration	62
A.3. Installation	63
A..4. User specific installation	63
<b>Appendix B. FVWM 2 Module Packet Types</b>	<b>64</b>
<b>B.1. #DEFINES FOR FVWM PACKET TYPES</b>	<b>64</b>
<b>B.2 CONTENTS OF PACKETS</b>	<b>65</b>
<b>Appendix C. Tables of Words</b>	<b>70</b>
<b>Appendix D. Source Code Layout</b>	<b>71</b>

# **1. Introduction**

## **1.1. BACKGROUND**

HAL, C3PO, the computer on the USS Enterprise, just some of the more famous machines to have been able to understand human speech and carry out commands spoken by a user. Automatic speech recognition by machine has always been a part of science fiction and research into it has been on-going for over four decades. In the last few years the processing power has become available for the home and office personal computer user to utilise this technology on their own system. Voice control of Microsoft Windows systems has recently become available but at present software for the UNIX X Window System is not readily available or is very expensive.

## **1.2. AIM**

The aim of this project is to develop and implement a voice controlled window manager for the X Windowing system running on a Linux or UNIX system with sound support. Although there are voice recognition systems currently available commercially on high-end UNIX systems, these are very costly and only work on a single type of UNIX system. This project is intended to be portable across UNIX systems with only small changes for sound device support.

### **1.3. BRIEF DESCRIPTION**

This project will allow the user to use their voice to control the common window manager functions on an X system. The main functions are moving, resizing, iconising, raising, lowering, and closing of applications. The window manager will have a common look and feel to make usage easier. The system will also have an easy to use trainer program to allow training of the recogniser for the user's voice.

Due to the prohibitively large size of designing and writing a window manager from scratch, it was decided to base the project on an already existing window manager. This meant either using an extensible window manager or re-writing one with hooks for the speech input. The fvwm window manager found on most Linux systems was selected as it has support for modules that can control the window manager from another process.

## **2. Speech Recognition System Theory**

### **2.1. INTRODUCTION**

There are a number of different types of speech recognition systems each with its area of application and use. There are also many different methods of implementing the different types of system, with some methods being better suited to some systems. This chapter explores the basic categories of recognition systems and the main methods of implementing speech systems in general. It will also give a background on the main type of signal processing used in speech systems and finish up with a comparison of the two main methods of implementing recognisers.

### **2.2. TYPES OF RECOGNITION SYSTEMS**

There are three main elements that distinguish speech recognition systems.

#### ***2.2.1. Speaker Independent/Dependent Systems***

The first criterion for distinguishing between speech recognition systems is speaker dependence / independence. A speaker dependent system is trained by a single speaker to their voice and only that person can reliably utilise the system unless it is re-trained for another person's voice. Speaker independence allows the system to be used

by any number of people depending on the level of training it receives. These systems are harder to implement and also require a suitable number of people to train them.

### ***2.2.2. Small/Medium/Large Vocabulary Systems***

The second set of categories is based on the size of the vocabulary of the recogniser, i.e. the number of words it recognises. Small vocabulary systems are typically in the region of ten to one hundred words. They are typically used in control systems where the vocabulary consists of a set of order words. The number of words in a medium vocabulary system is usually in the hundreds to low thousands range. These systems are used in some telephony applications where a large range of words is not expected but many common words are used. Finally there are large vocabulary systems which may reach as many as 60,000 words. These are usually research systems running on powerful hardware, but recently PC dictaphone software has become available that resides in this category. Large systems do not recognise whole words but rather split the words up into phonemes and try to recognise these. This requires a training set of words that encompasses most of the more common phonemes found in the English language.

### ***2.2.3. Isolated/Connected Word/Continuous Speech Systems***

The final distinguishing element is what sort of speech the system can recognise. Isolated Word Recognition ( IWR ) systems can recognise distinct words or maybe some two-word phrases. They require relatively clear start and end points to compare the words. Isolated word recognisers are usually control systems or single-digit recognisers. Connected word systems can handle simple connected sequences of words such as standard replies to questions. Both isolated word and connected word systems are usually small vocabulary system and can be either speaker dependent or independent. Continuous speech systems can recognise long sentences made up from non-connected words. Typically these are large vocabulary systems and are speaker independent.

## **2.3. METHODS USED IN SPEECH RECOGNITION**

There are four main methods used to achieve automatic speech recognition by machine.

### ***2.3.1. Template Based***

This approach is probably the most basic and easiest to understand. A collection of previous speech patterns are stored as reference templates and when an unknown word is to be recognised it is matched against these templates and the best match is chosen.

### ***2.3.2. Knowledge Based***

Pure knowledge based recognition systems use experts' knowledge of speech and involve direct and explicit incorporation of this knowledge into the systems. Their main success has been in their more indirect form where they have been used to direct the development and design of other algorithms and techniques to be used in other types of speech recognition system.

### ***2.3.3. Stochastic Based***

Stochastic systems use probabilistic methods and models to deal with uncertainties or incompleteness in the information they are presented with. For this reason they have been very successful in speech recognition systems as uncertainties come from many sources in speech systems such as background noise and different microphone qualities. The main stochastic approach used in speech recognition systems is the Hidden Markov Modelling (HMM) system.

### ***2.3.4. Connectionist Based***

Connectionist approaches are based around the use of neural network technology for speech recognition. Time delayed neural networks have been most successfully applied to the speech recognition problem. These systems rely on a good training algorithm and large amounts of training data. They use parallel distributed processing system and are most attractive in hardware systems as the underlying processing element in the neural network is simple and uniform.

## 2.4. SIGNAL PROCESSING IN SPEECH

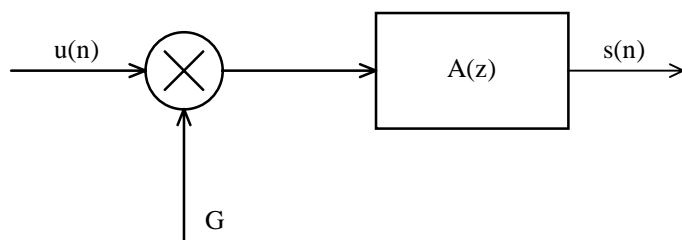
In speech recognition it is necessary to use some sort of signal processing front-end to reduce the raw speech signal to a more useful form that still contains most of the important features found in the original signal.

### 2.4.1. Linear Predictive Coding

The central idea behind Linear Predictive Coding (or LPC) is that a given speech sample may be approximated by a linear combination of previous speech samples and that by minimising the sum of the squared distances between the actual samples and the linearly predicted ones, over a finite interval, a unique set of predictor coefficients can be determined.

In a pure resonant system, linear prediction can exactly represent the system because resonant modes continue to ‘ring’ on once the excitation has ceased. This is approximately true with a speech system but not to the same extent, as formants vary slowly with time despite their strong resonance effect, the damping of a formant is modified by the opening and closing of the glottis and also the fact the resonances are always being excited to some extent by the sound sources. However the LPC system with optimisation can derive a close enough representation of the speech signal.

**Figure 2-1 LP model of speech**



#### 2.4.1.1 The LPC Model

From the basis of LPC, the fact that a given speech sample at time  $n$ ,  $s(n)$  can be approximated as a linear combination of the past  $p$  speech samples, we get

$$s(n) \approx a_1s(n-1) + a_2s(n-2) + \dots + a_p s(n-p) \dots \dots \dots (2.1)$$

where the coefficients  $a_1, a_2, \dots, a_p$  are assumed constant over the speech analysis frame. We convert the above equation to an equality by including an excitation term,  $G u(n)$  giving:

$$s(n) = \sum_{i=1}^p a_i s(n-i) + Gu(n) \dots\dots\dots (2.2)$$

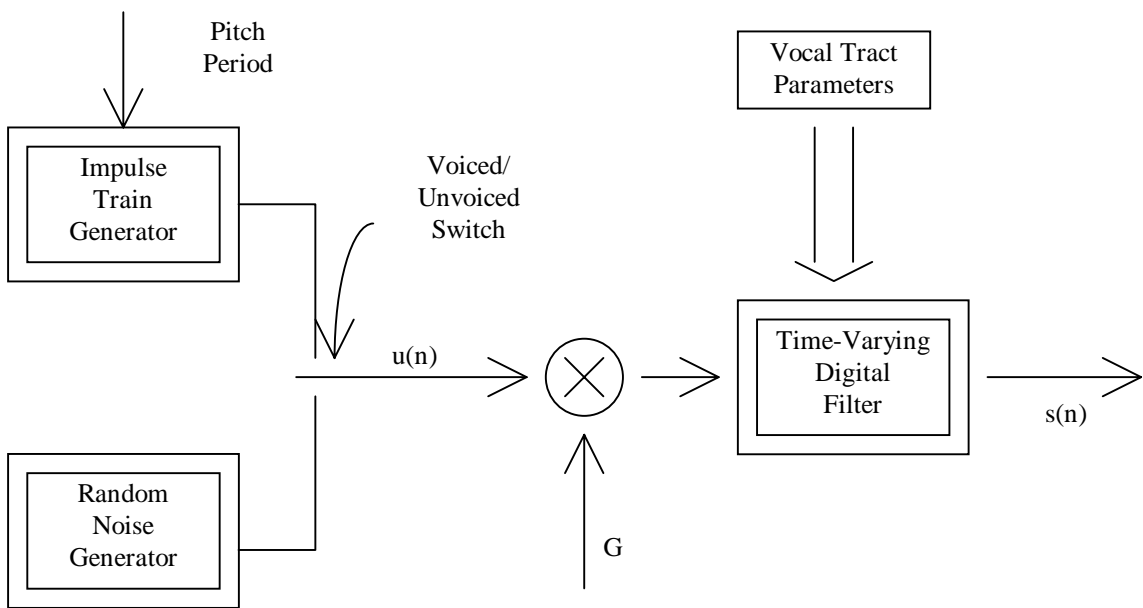
where  $u(n)$  is the normalised excitation and  $G$  is the gain of the excitation. The following relation is arrived at by expressing Eq. (2.2) in the z-domain,

$$S(z) = \sum_{i=1}^p a_i z^{-i} S(z) + GU(z) \dots\dots\dots (2.3)$$

leading to the transfer function

$$H(z) = \frac{S(z)}{GU(z)} = \frac{1}{1 - \sum_{i=1}^p a_i z^{-i}} = \frac{1}{A(z)} \dots\dots\dots (2.4)$$

This equation can be interpreted as a normalised excitation source  $u(n)$ , being scaled by the gain,  $G$ , and acting as an input to an all-pole system, to produce the speech signal  $s(n)$  as in Figure 2-1. The actual excitation function of speech is essentially either a quasiperiodic pulse or a random noise source. Figure 2-2 shows a synthesis model for speech, corresponding to the LPC analysis. Here the normalised excitation is chosen by a switch whose position is controlled by the voice/unvoiced characteristic of the speech.  $G$ , the gain of the source is estimated from the speech signal, and the scaled source is used as an input to a digital filter, which is controlled by the vocal tract parameters characteristic of the speech being produced. Thus the parameters of this model are voiced/unvoiced classification, pitch period for voiced sounds, the gain parameter, and the coefficients of the digital filter,  $\{a_k\}$ . These parameters all vary slowly with time.



**Figure 2-2 Speech synthesis model based on LPC model.**

### 2.4.1.2. LPC Analysis Equations

From the model of Figure 2-1 the exact relation between  $s(n)$  and  $u(n)$  is known. This is

$$s(n) = \sum_{k=1}^p a_k s(n-k) + Gu(n) \dots\dots\dots (2.5)$$

Consider the linear combination of speech samples as the estimate  $\bar{s}(n)$ , defined as

$$\bar{s}(n) = \sum_{k=1}^p a_k s(n-k) \dots\dots\dots (2.6)$$

The prediction error,  $e(n)$ , can be defined as

$$e(n) = s(n) - \bar{s}(n) = s(n) - \sum_{k=1}^p a_k s(n-k) \dots\dots\dots (2.7)$$

with error transfer function

$$A(z) = \frac{E(z)}{S(z)} = 1 - \sum_{k=1}^p a_k z^{-k} \dots\dots\dots (2.8)$$

When  $s(n)$  is actually generated by a linear system of the type shown in Figure 2-1 the prediction error  $e(n)$ , will equal  $G u(n)$ , the scaled excitation.

The predictor coefficients,  $\{a_k\}$ , are determined directly from the speech signal so that the spectral properties of the digital filter in Figure 2-2 match those of the speech waveform within the analysis window. The predictor coefficients at a given time,  $n$ , must be estimated from a short segment of the speech signal occurring around time  $n$ , due to the fact that the spectral characteristics of speech vary over time. The set of predictor coefficients are found such that the mean-squared prediction error is minimised over a short segment of speech.

To set-up the equations necessary to determine the predictor coefficients, the short-term speech and error segments at time  $n$  are defined as

$$s_n(m) = s(n+m) \dots\dots\dots (2.9)$$

$$e_n(m) = e(n+m) \dots\dots\dots (2.10)$$

and the goal is to minimise the mean squared error at time  $n$ ,

$$E_n = \sum_m e_n^2(m) \dots\dots\dots (2.11)$$

This can also be written using the definition of  $e_n(m)$  in terms of  $s_n(m)$  as,

$$E_n = \sum_m \left[ s_n(m) - \sum_{k=1}^p a_k s_n(m-k) \right]^2 \dots\dots\dots (2.12)$$

It is necessary to solve this equation for the predictor coefficients. Differentiating  $E_n$  with respect to each  $a_k$  and setting the result to zero,

$$\frac{\partial E_n}{\partial a_k} = 0, k=1,2,\dots,p \dots\dots\dots (2.13)$$

gives

$$\sum_m s_n(m-i)s_n(m) = \sum_{k=1}^p \hat{a}_k \sum_m s_n(m-i)s_n(m-k) \dots\dots\dots (2.14)$$

Terms of the form  $\sum_m s_n(m-i)s_n(m-k)$  are terms of the short-term covariance of  $s_n(m)$ , i.e.

$$\phi_n(i,k) = \sum_m s_n(m-i)s_n(m-k) \dots\dots\dots (2.15)$$

and using this Eq. (2-14) can be expressed in the following notation.

$$\phi_n(i,0) = \sum_{k=1}^p \hat{a}_k \phi_n(i,k) \dots\dots\dots (2.16)$$

This equation describes a set of p equations in p unknowns. It can be shown easily the mean squared error,  $\hat{E}(n)$ , can be expressed as,

$$\begin{aligned} \hat{E}_n &= \sum_m s_n^2(m) = \sum_{k=1}^p \hat{a}_k \sum_m s_n(m)s_n(m-k) \dots\dots\dots (2.17) \\ &= \phi_n(0,0) - \sum_{k=1}^p \hat{a}_k \phi_n(0,k) \end{aligned}$$

This shows that the minimum mean-squared error consists of a fixed term ( $\phi_n(0,0)$ ) and terms that depend on the predictor coefficients.

In order to solve Eq.(2-16) for the optimum predictor coefficients (the  $\hat{a}_k$ s),  $\phi_n(i,k)$  has to be computed for  $1 \leq i \leq p$  and  $0 \leq k \leq p$ , and the resulting set of p simultaneous equations solved.

Solving this set of equations is a strong function of the range m. The two principle methods used are the

1. Autocorrelation Method, and the
2. Covariance Method

Of these two the autocorrelation method has been found to be of most use in speech processing applications so the covariance method will not be discussed.

2.4.1.3. The Autocorrelation Method

A limit needs to be placed on m in the summations given above. The most straightforward method is to assume that the speech segment,  $s_n(m)$ , is identically zero outside the interval  $0 \leq m \leq N-1$ . This is equivalent to windowing the speech signal,  $s(m+n)$  with a finite length window,  $w(m)$ , which is identically zero outside the range  $0 \leq m \leq N-1$ . This can be expressed as

$$s_n(m) = \begin{cases} s(m+n).w(m), & 0 \leq m \leq N-1 \\ 0, & \text{otherwise.} \end{cases} \dots\dots\dots (2.18)$$

From this equation, for  $m < 0$ , the error signal  $e_n(m)$  is exactly zero since  $s_n(m) = 0$  for all  $m < 0$  and therefore there is no prediction error. This also applies for  $m > N-1+p$  as  $s_n(m) = 0$  for all  $m > N-1$ . A hamming window may be used for  $w(n)$  to taper the ends of the speech sample to zero which provides a smaller prediction error.

Based on Eq. (2.18) the mean-squared error is now

$$E_n = \sum_{m=0}^{N-1+p} e_n^2(m) \dots\dots\dots (2.19)$$

and  $\phi_n(i,k)$  can be expressed as

$$\phi_n(i,k) = \sum_{m=0}^{N-1+p} s_n(m-i)s_n(m-k) = \sum_{m=0}^{N-1-(i-k)} s_n(m)s_n(m+i-k), \quad 1 \leq i \leq p, \quad 0 \leq k \leq p \dots\dots\dots (2.20)$$

Since this is only a function of (i-k) rather than two independent variables i and k, the covariance function  $\phi_n(i,k)$  reduces to a simple auto-correlation function

$$\phi_n(i,k) = r_n(i-k) = \sum_{m=0}^{N-1-(i-k)} s_n(m)s_n(m+i-k) \dots\dots\dots (2.21)$$

Since that autocorrelation function is symmetric i.e.  $r_n(-k) = r_n(k)$  the LPC equations can be expressed as

$$\sum_{k=1}^p r_n(|i-k|) \hat{a}_k = r_n(i), \quad 1 \leq i \leq p \dots\dots\dots (2.22)$$

and in matrix form it looks like

$$\begin{bmatrix} \mathbf{r}_n(0) & \mathbf{r}_n(1) & \mathbf{r}_n(2) & \cdots & \mathbf{r}_n(p-1) \\ \mathbf{r}_n(1) & \mathbf{r}_n(0) & \mathbf{r}_n(1) & \cdots & \mathbf{r}_n(p-2) \\ \mathbf{r}_n(2) & \mathbf{r}_n(1) & \mathbf{r}_n(0) & \cdots & \mathbf{r}_n(p-3) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{r}_n(p-1) & \mathbf{r}_n(p-2) & \mathbf{r}_n(p-3) & \cdots & \mathbf{r}_n(0) \end{bmatrix} \begin{bmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \\ \vdots \\ \hat{a}_p \end{bmatrix} = \begin{bmatrix} \mathbf{r}_n(1) \\ \mathbf{r}_n(2) \\ \mathbf{r}_n(3) \\ \vdots \\ \mathbf{r}_n(p) \end{bmatrix} \dots\dots\dots (2.23)$$

The above  $p \times p$  matrix of autocorrelation values is called a Toeplitz matrix and can be solved efficiently through several well-known procedures, the main one being the Durbin algorithm.

## 2.5. TEMPLATE VS. STOCHASTIC

The two most successful methods of implementing speech recognition systems have been template and stochastic based systems. The main algorithms used in both methods, Dynamic Time Warping ( DTW ) and Hidden Markov Models ( HMMs ) will be discussed.

### 2.5.1. Dynamic Time Warping

#### 2.5.1.1. Time Alignment

In template based systems the pattern similarity process involves both time alignment and distance computation. The time alignment process tries to map the reference pattern onto the corresponding parts of the test pattern such that the distance between the functions is minimised by the mapping.

This can be mathematically specified in terms of the reference pattern  $R$ , test pattern  $T$ , distance between functions  $D(T,R)$  and  $w$  the mapping function:

$$D(T,R) = \min_{\{w(t)\}} \int_{t_0}^{t_1} d(t, w(t)) G(t, w(t), w'(t)) dt \dots\dots\dots (2.24)$$

where  $\{ w(t) \}$  is the set of all monotonically increasing, continuous differentiable functions;  $w'(t)$  is the derivative of  $w(t)$ ;  $d(t, w(t))$  is a metric  $d(T(t), R(w(t)))$  which is the pointwise distance from  $R$  to  $T$  and  $G$  is a weighting function.

Unfortunately, the above problem is not in general solvable so it is reduced to discrete values by letting,

$$T = \{ T(1), T(2), \dots T(NT) \} \dots\dots\dots (2.24a)$$

and

$$R = \{ R(1), R(2), \dots R(NR) \} \dots\dots\dots (2.24b)$$

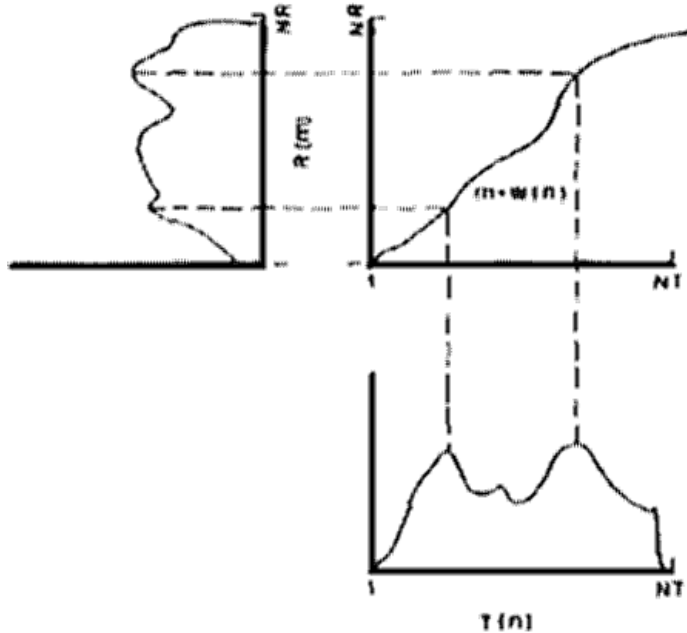
From these a number of techniques may be used. The optimum time alignment path is a curve which relates the  $m$  time axis of the reference pattern to the  $n$  time axis of the test pattern, of the form

$$m = w(n). \quad (2.25)$$

The constrained beginning and ending points shown in Figure 2-3 can be formally expressed as constraints on  $w(n)$  of the form

$$w(1)=1 \quad \dots\dots\dots (2.26a)$$

$$w(NT)=NR \quad \dots\dots\dots (2.26b)$$



**Figure 2-3 Example of time registration of a test and a reference pattern from [1]**

Several techniques have been proposed for determining the alignment path  $w$ , these include,

- Linear Time Alignment
- Time Event Matching
- Correlation Maximisation
- Dynamic Time Warping.

Of these dynamic time warping has been the most successful and will be the only one discussed here.

2.5.1.2. Dynamic Time Warping

This method uses the following optimisation problem to get the warping curve,  $w$ :

$$D^* = \min_{w(n)} \left[ \sum_{n=1}^{NT} d(T(n), R(w(n))) \right] \dots\dots\dots (2.27)$$

where  $d(T(n), R(w(n)))$  is the “distance” between the frame  $n$  of the test pattern, and frame  $w(n)$  of the reference pattern.

### 2.5.1.3. Distance Measures

In order to use the DTW optimisation, the concept of distance between frames of features has to be defined. Common distance measures used are

1. Euclidean Distance
2. Covariance Weighting
3. Spectral Distance
4. LPC Log Likelihood.

Of these four both the covariance weighting and spectral distance measures are computationally intensive.

The Euclidean distance measure is simply

$$d(T,R)=\|T-R\|=\sum_{i=0}^p(T_i-R_i)^2 \dots\dots\dots (2.28)$$

where  $T_i$  and  $R_i$  are the  $i$ th components of the vectors  $T$  and  $R$ .

The LPC Log Likelihood Measure is of the form

$$d(T,R)=\log\left[\frac{a_R V_T a_R^t}{a_T V_T a_T^t}\right] \dots\dots\dots (2.29)$$

where  $a_R$  and  $a_T$  are the LPC coefficient vectors of the reference and test frames and  $V_T$  is the matrix of autocorrelation coefficients of the test frame. The distance  $d$  can be expressed exactly in the form,

$$d(T,R)=\log\left[\sum \tilde{V}_T(m) R_R^a(m)\right] \dots\dots\dots (2.30)$$

where

$$V_T(m)=\frac{\tilde{V}_T(m)}{E_T} \dots\dots\dots (2.31)$$

where  $E_T$  is the normalised error of the LPC analysis of the test frame, and

$$R_R^a(m)=\sum_{k=0}^{p-m} a_R(k) a_R(k+m) \dots\dots\dots (2.32)$$

i.e.  $R_R^a(m)$  is the autocorrelation of the finite vector

$$a_R = [1, a_R(1), a_R(2), \dots, a_R(p)] \dots\dots\dots (2.33)$$

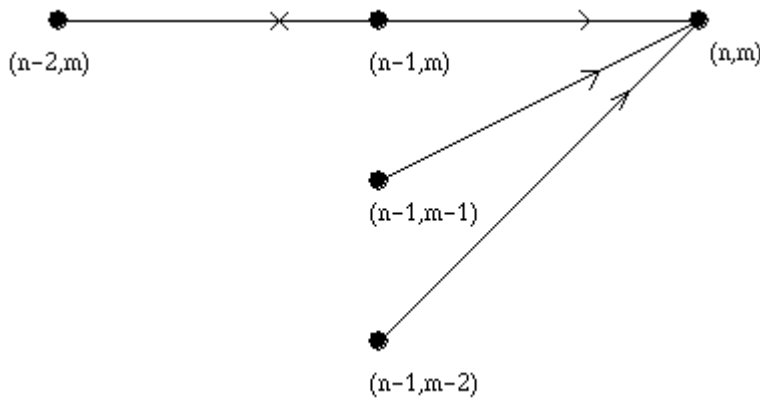
2.5.1.4. Basic DTW approach

The basis behind most DTW algorithms is the realisation that the solution to Eq. (2-24) is equivalent to finding the “best” path through a finite grid. Classical path-finding techniques then become applicable to the problem. A simple recursive technique it has been shown can find the best path in the grid.

First  $D_A(n, m)$  is defined as the minimum accumulated distance from the initial grid point  $n = 1, m = 1$  to the grid point  $(n, m)$ . If  $n$  is assumed to represent the independent grid search variable, and that all paths through the grid correspond to monotonically increasing time indices, then the following recursion can be written,

$$D_A(n, m) = d(T_n, R_m) + \min_{q \leq m} [D_A(n-1, q)] \dots\dots\dots (2.34)$$

This says that the minimum accumulated distance to the grid point  $(n, m)$  consists of the local distance  $d$  between feature sets  $T_n$  and  $R_m$  plus the minimum accumulated distance to the grid point  $(n-1, q)$  where  $q$  is the set of  $m$  values such that a path exists between  $(n-1, q)$  and  $(n, m)$ .



**Figure 2-4 Set of possible transitions to point (n,m)**

The figure above gives an example where there are only three valid paths to any point  $(n, m)$ , from  $(n-1, m)$ ,  $(n-1, m-1)$  and  $(n-1, m-2)$ . It also further constrains the model to being non-linear which states that if the best path to point  $(n-1, m)$  came through  $(n-2, m)$  then no path can lead from  $(n-1, m)$  to  $(n, m)$ . This can formally be expressed as

$$\begin{aligned}
 w(n) - w(n-1) &= 0, 1, 2 && \text{if } w(n-1) \neq w(n-2) \\
 &= 1, 2 && \text{if } w(n-1) = w(n-2) \dots\dots\dots (2.35)
 \end{aligned}$$

giving the modified form of Eq. (2-34) as

$$D_A(n, m) = d(T_n, R_m) + \min \left[ \begin{array}{l} D_A(n-1, m)g(n-1, m), \\ D_A(n-1, m-1), D_A(n-1, m-2) \end{array} \right] \dots\dots\dots (2.36)$$

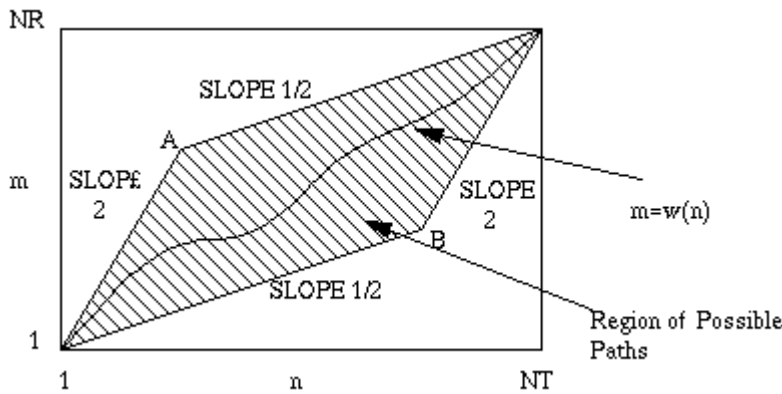
where

$$g(n-1, m) = \begin{cases} 1 & \text{if } w(n-1) \neq w(n-2) \\ \infty & \text{if } w(n-1) = w(n-2) \end{cases} \dots\dots\dots (2.37)$$

The iteration of Eq. (2-36) is carried out over all valid m, for each value of n sequentially from n=1 to n=NT, and the final desired solution is given as

$$D^* = D_A(NT, NR) \dots\dots\dots (2.38)$$

The optimum warping path w(n) is determined by backtracking from the end of the path back to the beginning. For applications in word-recognition this step does not need to be computed as only D\* is required.



**Figure 2-5 Typical range for dynamic warping path with slope constraints of 2 to 1 and 1/2 to 1.**

Figure 2-5 demonstrates a typical DTW warping region. This has the end-point constraints of Eq. (2.26) and the local path constraints of Eq. (2.35) which means the optimal warping path has to lie in the shaded region.

There are some variants on this basic DTW usually based on the endpoint constraints for situations where the start and end points of the patterns cannot easily be identified.

### 2.5.2. Hidden Markov Models

Hidden Markov Modelling is a statistical method used in speech recognition. It has been very successful in all areas of recognition and works on the assumption that the speech signal can be well characterised as a parametric random process, and that the parameters of the stochastic process can be estimated in a precise, well-defined manner.

#### 2.5.2.1. Discrete-Time Markov Processes

Discrete-Time Markov Processes form the basis of HMM theory. Consider a system whose description is that it is in one of a set of  $N$  distinct states indexed by  $\{1, 2, \dots, N\}$ . At regular spaced, discrete times, the system undergoes a change of state, possibly returning to the same state, according to a set of probabilities associated with the state. The time instants at which the state changes occur are denoted as  $t = 1, 2, \dots$ , and the state at any time  $t$  is  $q_t$ . A full probabilistic description of the system would, in general, require specification of the current state as well as all predecessor states. The probabilistic dependence function may be truncated in the case of a discrete-time, first order, Markov chain, i.e.

$$P[q_t = j | q_{t-1} = i, q_{t-2} = k, \dots] = P[q_t = j | q_{t-1} = i] \dots \dots \dots (2.39)$$

If only those processes in which the right hand side of the above is independent in time, we get a set of state-transition probabilities  $a_{ij}$  of the form

$$a_{ij} = P[q_t = j | q_{t-1} = i] \quad 1 \leq i, j \leq N \dots \dots \dots (2.40)$$

with the following properties,

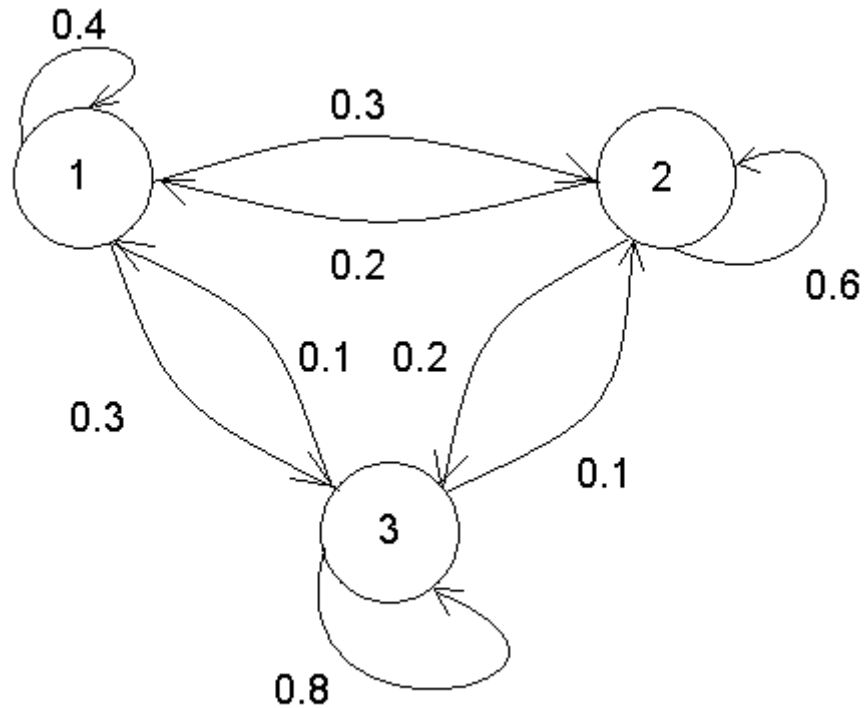
$$\begin{aligned} a_{ij} &\geq 0 \quad \forall i, j \\ \sum_{j=1}^N a_{ij} &= 1 \quad \forall i \end{aligned} \quad (2.41)$$

since they obey probabilistic constraints. This can be considered observable because the output process is a set of states at each instant of time, where each state corresponds to an observable event.

A good example of this is a three-state model of the weather. State one corresponds to rain, state two to cloudy and state three to sunny. The weather is assumed

to be observed once per day, and on a given day  $t$  is described by only a single state from the three. The matrix  $A$  of state-transition probabilities is

$$A = \{a_{ij}\} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix} \quad (2.42)$$



**Figure 2-6 Markov Model of the weather**

From this the questions like what is the probability that the weather for eight consecutive days is “sun-sun-sun-rain-rain-sun-cloudy-sun”?

$O =$	<i>sun</i>	<i>sun</i>	<i>sun</i>	<i>rain</i>	<i>rain</i>	<i>sun</i>	<i>cloudy</i>	<i>sun</i>
	3	3	3	1	1	3	2	3

$$\begin{aligned}
 P(\ddot{O} | Model) &= P[3]P[3|3]P[3|3]P[1|3]P[1|1]P[3|1]P[2|3]P[3|2] \\
 &= \pi_3 a_{33} a_{31} a_{11} a_{13} a_{32} a_{23} \\
 &= 1.536 * 10^{-4} \quad \dots\dots\dots (2.43)
 \end{aligned}$$

where the following notation is used to donate the initial state probabilities,

$$\pi_i = P[q_1 = i], \quad 1 \leq i \leq n \quad \dots\dots\dots (2.44)$$

### 2.5.2.2. Hidden Markov Models

In the previous model, each state corresponded to a deterministically observable event. Therefore, the output of such sources in any given state is not random. This model is too restrictive to be applicable to many interesting problems. It can however be extended to include the case in which the observation is a probabilistic function of the state, and the resulting model, the HMM, is a doubly stochastic process with an underlying stochastic process that is not directly observable but can be observed only through another set of stochastic processes that produce the sequence of observations.

The basic concepts of hidden Markov models can be explained with the simple coin-tossing example.

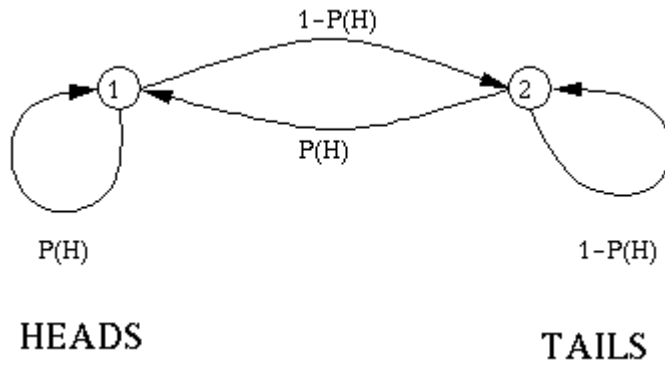
### 2.5.2.3. Coin Tossing Model

The coin toss model is based on a scenario where two people are sitting in two rooms separated by a curtain. One person is performing the coin-tossing experiment (using one or more coins). This person does not tell what coin they have flipped only what the result of the coin-flip is to the other person. Therefore a sequence of hidden coin-tossing experiments is performed, with the observation sequence consisting of a sequence of a series of heads and tails, e.g.

$$\begin{aligned} \mathbf{O} &= (\mathbf{o}_1 \mathbf{o}_2 \mathbf{o}_3 \dots \mathbf{o}_t) \\ &= (\text{H H T T T H T T H} \dots \text{H}) \end{aligned}$$

where H stands for heads and F for tails.

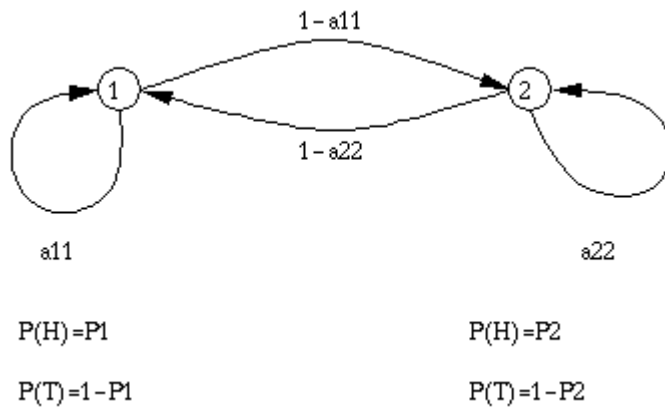
From this a HMM must be built to explain the observed sequence of heads and tails. The first decision is to decide what the states in the model stand for, and then how many states should be in it. Here one possible choice would be that a single biased coin is being tossed. This can be modelled with a two state model where each state corresponds to the outcome of the previous toss. This is depicted in Figure 2-7a and in this case the model is observable and only thing left to specify is the best value for the single parameter  $P(H)$ .



**Figure 2-7a** Single Coin Toss Models

<i>O</i> =	<i>H</i>	<i>H</i>	<i>T</i>	<i>T</i>	<i>H</i>	<i>T</i>	<i>H</i>	<i>H</i>	<i>T</i>	<i>T</i>	<i>H</i>	.....
<i>S</i> =	1	1	2	2	1	2	1	1	2	2	1	.....

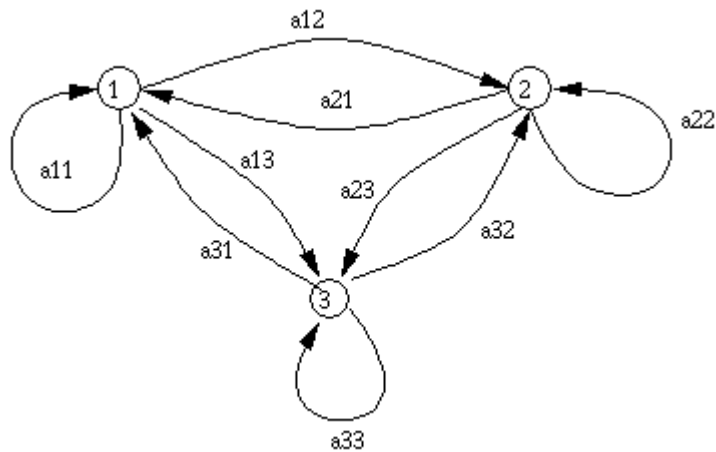
Another HMM for explaining the observation sequence is shown in diagram Figure 2-7b. This is also a two state model but each state corresponds to a different, biased coin being tossed. Each state is characterised by a probability distribution of heads and tails, and transitions between states are characterised by a state-transition matrix. The physical method that accounts for how the state transitions are selected could be itself a set of independent coin tosses or another probabilistic event.



**Figure 2-7b**

<i>O</i> =	<i>H</i>	<i>H</i>	<i>T</i>	<i>T</i>	<i>H</i>	<i>T</i>	<i>H</i>	<i>H</i>	<i>T</i>	<i>T</i>	<i>H</i>	.....
<i>S</i> =	2	1	1	2	2	2	1	2	2	1	2	.....

A third form is shown in Figure 2-7c. This corresponds to three-biased coins, and the choice between the three is based on some probabilistic event.

**Figure 2-7c**

$O=$	$H$	$H$	$T$	$T$	$H$	$T$	$H$	$H$	$T$	$T$	$H$	.....
$S=$	3	1	2	3	3	1	1	2	3	1	3	.....

The choice of which of the three models best matches the actual observations comes from the number of unknown parameters for the model. The single-coin model has one unknown, the two-coin four unknowns and the three-coin nine unknowns. In theory the models with the higher degrees of freedom provide a better model for the system, but practical considerations impose limitations.

Another example of a more complex HMM is the urn-and-ball model.

#### 2.5.2.4. Urn-and-Ball Model

The urn-and-ball model is a more complicated situation and the system consists of  $N$  glass urns in a room. Each urn contains a large quantity of balls and each ball is coloured and there are  $M$  distinct colours. The process for selecting a ball is a genie appears, randomly selects an urn and then randomly selects a ball from that urn and the colour of the ball is recorded as the observation. The ball is replaced in the urn from which it was removed. A new urn is selected by some random selection procedure associated with the current urn and it repeats. This process generates a finite observation of colours which can be modelled as the observable output of a HMM.

The simplest model for this system is one in which each state corresponds to a specific urn, and for which a colour probability is defined for each state. The urn choice is dictated by the state-transition matrix of the HMM.

### 2.5.2.5. Elements of a HMM

From the two examples above, the basic idea behind what a HMM is and how it can be applied to simple scenarios can be seen. A more formal definition of the elements of a HMM will now be given.

A HMM for discrete symbol observations such as the urn-and-ball model is characterised by the following

1.  $N$ , the number of states in the model. There is usually some physical significance attached to the states or a set of states despite their hidden nature. In the coin tossing experiment each state corresponded to a distinct biased coin. The urns made up the states in the urn-and-ball example. The states can be interconnected in a few different ways, and are labelled  $\{ 1, 2, \dots, N \}$ , and the state at time  $t$  is  $q_t$ .
2.  $M$ , the number of distinct observation symbols per state. These correspond to the physical output of the system being modelled. For the above examples, heads or tails and the colours of the balls where the observation symbols. The set of symbols  $V = \{ v_1, v_2, \dots, v_M \}$ .
3. The state-transition probability distribution  $A = \{ a_{ij} \}$  where

$$a_{ij} = P[q_{t+1} = j | q_t = i], \quad 1 \leq i, j \leq N. \dots\dots\dots (2.45)$$

For the special case in which any state can reach any other state in a single step,  $a_{ij} > 0$  for all  $i, j$ . For other types of HMMS  $a_{ij} = 0$  for one or more pairs.

4. The observation symbol probability distribution,  $B = \{ b_j(k) \}$ , in which

$$b_j(k) = P[o_t = v_k | q_t = j], \quad 1 \leq k \leq M \dots\dots\dots (2.46)$$

defines the symbol distribution in state  $j$ ,  $j = 1, 2, \dots, N$ .

5. The initial state distribution  $\pi = \{ \pi_i \}$  in which

$$\pi_i = P[q_1 = i], \quad 1 \leq i \leq N. \dots\dots\dots (2.47)$$

The complete specification of a HMM, from the above, requires specification of two model parameters,  $N$  and  $M$ , specification of observation symbols, and the specification of three sets of probability measures  $A$ ,  $B$ , and  $\pi$ . The compact notation  $\lambda = (A, B, \pi)$  is used for convenience to indicate the complete parameter set of the model. This defines a probability measure for  $\mathbf{O}$ , i.e.  $P(\mathbf{O} | \lambda)$ .

Given appropriate values of  $N$ ,  $M$ ,  $A$ ,  $B$ , and  $\pi$ , the HMM can be used to generate an observation sequence

$$\mathbf{O} = \{ o_1, o_2, \dots, o_T \}$$

as follows:

1. Choose an initial state  $q_t = i$  according to the initial state distribution  $\pi$ .
2. Set  $t = 1$
3. Choose  $o_t = v_k$  according to the symbol probability distribution in state  $i$ , i.e.  $b_j(k)$ .
4. Transit to a new state  $q_{t+1} = j$  according to the state-transition probability distribution for state  $i$ , i.e.  $a_{ij}$ .
5. Set  $t = t + 1$ . Return to step 3 if  $t < T$ ; otherwise, terminate the procedure.

This procedure can be used as both a generator of observations, and as model for how a given observation sequence was generated by an appropriate HMM.

#### 2.5.2.6. The three basic problems for HMMs

For the model to be useful in real-world application, three basic problems must be solved and they are as follows:

##### **Problem 1:**

Given the observation sequence  $\mathbf{O} = \{ o_1 o_2 \dots o_T \}$  and a model  $\lambda = (A, B, \pi)$  how can  $P(\mathbf{O}|\lambda)$ , the probability of the observation sequence, be efficiently computed given the model?

##### **Problem 2:**

Given the observation sequence  $\mathbf{O} = \{ o_1 o_2 \dots o_T \}$ , and the model  $\lambda$ , how can a corresponding state sequence  $q = \{ q_1 q_2 \dots q_T \}$  that is optimal in some sense be chosen?

##### **Problem 3:**

How can the model parameters  $\lambda = (A, B, \pi)$  be adjusted to maximise  $P(\mathbf{O}|\lambda)$ ?

Problem 1 is the evaluation problem. If given a model and a sequence of observations, how is probability that the observed sequence was produced by the model computed.

Problem 2 attempts to uncover the hidden part of the model, and find the “correct” state sequence. This cannot be done for all but the simplest models, so for practical situations, an optimality criterion is used to solve the problem as best as possible.

Problem 3 attempts to optimise the model parameters to be describe how a given observation sequence comes about. Training sequences are used to adjust the model parameters. The training problem is the crucial one as it allows best models to be created for real phenomena.

The solutions for these three problems are of a mathematical nature ([2]) and will not be shown here. The solution to problem 1 is the so called forward and backward algorithms. These algorithms provide an efficient method of calculating the probability of an observation sequence,  $\mathbf{O}$ , given the model,  $\lambda$ . These provide an exact solution for problem 1. Unlike problem 1, there is no exact solution for problem 2 which is finding the optimal state sequence associated with the observation sequence. One formal technique that exists is the Viterbi algorithm. Problem 3, to determine a method to adjust the model parameters  $\lambda = (A, B, \pi)$  to satisfy a certain optimisation criterion, is tee most difficult problem of the three. The procedure used is called the Baum-Welch method and works on re-estimation of the model after each observation vector until the optimal model is arrived at.

#### 2.5.2.7. Types of HMM

The main type of HMM considered has been the fully connected HMM or ergodic HMM in which each state has been reachable from every other state in a single step. Another model which has proved in some applications to be more useful is the Bakis or left-right model, so called because it has the property that, as time increases, the state index increases ( or remains static ) i.e. the system proceeds from left to right.

Another useful type of HMM is the continuous observation density HMM [3] The previously mentioned HMMs used discrete symbols chosen from a finite alphabet, but in some applications the observations are often continuous vectors. Although these can be converted to discrete vectors using codebooks and vector quantisation the quality can be degraded.

#### **2.5.3 Comparison**

Both techniques have been equally successful in word recognition systems, with the dynamic time warping being easier to implement. In larger vocabulary systems, HMMs have been found to be more successful.

## **3. X Windowing System**

### **3.1. INTRODUCTION**

This chapter introduces the X Windowing System and how it is structured. It introduces the window manager concept and discusses what jobs it must do. Fvwm is introduced and its module interface explained.

### **3.2. X WINDOW BACKGROUND**

#### ***3.2.1. What is X?***

X is a networked-based windowing system that is available on nearly every workstation currently in use. It provides the ability for applications running on multiple different architectures, hardware and operating systems to be displayed on one screen. It is portable across operating systems at the main API level known as Xlib.

#### ***3.2.2. History of X***

The X Window system grew out a project called Project Athena, launched in 1984 at the Laboratory for Computer Science at MIT. The goal of the project was to make programs available interactively to users at any station, anywhere on the campus. This

meant it had to work across different hardware and on different operating systems. The starting point was a Stanford University windowing system known as W.

The X Window system first became available to developers with version 10.4 in 1986. Version 11 was a much more complete windowing package than version 10, it provided better flexibility in areas of supported display features, window manager styles, and better performance. Since then major computer vendors such as DEC, IBM, AT&T, Sun and HP have funded the development of the X Window system. Currently it is being looked after by the X Consortium but control is being transferred to Open Group (a.k.a. OSF and X/Open ). The latest version is X11R6.3 with the final version being released by the X Consortium being codenamed Broadway.

### ***3.2.3. Client-Server Architecture***

X is based on a client/server architecture. Clients, or X application programs communicate with servers, or X display units over a network.

#### *3.2.3.1. Client*

The client is an application program. It does not interact directly with the user but rather receives events ( keypresses, mouse clicks etc. ) from the server. It executes X commands that request the server to draw graphics. Several clients may be attached to a single server.

#### *3.2.3.2. Server*

The server is a program running on a machine with graphics capabilities e.g. X-terminal, UNIX workstation, or PC workstation. It passes events to the clients corresponding to keypresses, mouse motion or mouse clicking. This is unscheduled and any type of event may occur in any order. It decodes client messages, such as requests for information, moving of windows or for graphics to be drawn. These messages are expressed in a formal language. It maintains complex data structures, which reduce client storage and processing and tries to minimise the amount of data transmitted over the network.

#### *3.2.3.3. Connection*

The link between client and server can either be over a physical network or may be internal in the UNIX machine. The most common protocols for running X over a

physical network are TCP/IP, DECnet and SVR4 STREAMS. Internal connection are usually done through UNIX domain sockets.

### 3.3. PROGRAMMING FOR X

Programming for X entails using one of two approaches, Xlib or Toolkits.

Xlib is the C programming interface to X11. It is the lowest level of programming interface to X. Toolkits are a higher level approach to programming X11. They provide widgets such as command buttons, menus, scrollbars and other features of the user interface. Common toolkits are the Xt and Xaw widget sets which are part of the X11 distribution, Motif from the OSF ( Open Software Foundation ) and OpenView from SUN.

The Xlib interface is more difficult to program and a lot more code can be required to program a simple application, however it allows more low-level operations to be executed.

### 3.4. WINDOW MANAGEMENT

X requires a special client to run to provide window management. This client, the window manager, is just another program written in Xlib, that has the authority to control the screen display. Some of its duties are to set window sizes, and to determine where applications place windows. Most window managers also provide decorations such as title bars, raised buttons, and root window menus.

Applications communicate with the window manager and request window sizes, placement and decorations. These requests are only suggestions or *hints* for the window manager and it is up to it to make the final decision. Although the application can override the window manager if it wishes this is bad behaviour as it may affect other applications that are running.

Window managers provide the “look and feel” element to the X window system. It means that different users can provide their own window managers and setup files and provide their own look and feel. Some of the more common look-and-feels currently

available are the CDE ( Common Desktop Environment ) a complete system, NextStep, from the NeXT system, and the Windows 95 look.

### 3.5. FVWM

#### 3.5.1. *What is fvwm and what does it stand for?*

Fvwm is a free virtual window manager originally derived from twm (one of the first window managers). It is mainly used on Linux systems due to its free nature but it can be compiled for nearly any UNIX system with X support. Currently version one has been released and version two is in beta. Version two is assumed anywhere fvwm is mentioned in this document.

Fvwm stands for (from the FAQ[4])

*"Fill\_in\_the\_blank\_with\_whatever\_f\_word\_you\_like\_at\_the\_time  
Virtual Window Manager". Rob Nation (the original Author of FVWM)  
doesn't really remember what the F stood for originally, so we  
have several potential answers:  
Feeble, Fabulous, Famous, Fast, Foobar, Fantastic, Flexible,  
F!@#%\$, FVWM (the GNU recursive approach), Free, Final, Funky,  
Fred's (who the heck is Fred?), Freakin', etc."*

#### 3.5.2. *The .fvwmrc file*

The .fvwmrc or .fvwm2rc file is the user's personal configuration and initialisation file for fvwm. It contains the user's preferences and styles. It controls the look and feel the user gets, and contains things like what icons to display on the screen for what applications and what sort of window style each application gets. It also contains the ModulePath where fvwm can look for modules, and module specific resources which always begin with a '\*'.

#### 3.5.3. *Fvwm Module System*

Fvwm has the ability to use plug-in modules to extend its basic abilities. It comes with its own selection of modules like a button-bar, window raiser, and a window listing utility. The design goals of the module system are

- Modules should be able to provide the window manager with a limited amount of instruction regarding instructions to execute
- Modules should not be able to corrupt the internal databases maintained by fvwm, nor should unauthorised modules be able to interface to fvwm.
- Modules should be able to extract nearly all the information held by the window manager, in a simple manner, to provide users with feedback not available with built-in services.
- Modules should gracefully terminate when the window manager dies, exits or restarted.

### 3.5.3.1. Security

Security is needed in the module system to stop unauthorised modules connecting to the window manager. This is done by only allowing modules that fvwm starts, which means they must be in the user's .fvwmrc file. Modules can only issue commands that can normally be issued from key or mouse bindings, so it stops malicious users connecting to other peoples' window managers and tries to stop the module corrupting the window managers internal databases.

### 3.5.3.2. Interface to and Initialisation of modules

Fvwm must launch all modules. The initialisation procedure is as follows

1. Fvwm opens a pair of UNIX pipes, one for messages from it to the module and one for messages from the module to it.
2. Fvwm forks and executes the module.

For the above to work the module must be located in the ModulePath as specified in the users .fvwmrc file. Modules can either be initiated when fvwm starts or as a user action during the X session.

The modules initialisation procedure is quite straightforward. It must :

1. Check the number of command line arguments to make sure that it was launched by fvwm. Fvwm passes the file descriptors of the open pipes, the path to the fvwm configuration file, the application window in which the module was launched and window decoration from which it was launched. It also passes user arguments.
2. Save the file descriptors given on the command line for later use.

3. Set up a signal handler to exit cleanly when the module receives SIGPIPE which means that fvwm has exited and the pipes have been closed on the other end.

#### 3.5.3.3. Module-to-FVWM Communication

Modules communicate with fvwm through a simple text protocol. A textual command line similar to the lines contained in the .fvwmrc file is transmitted to fvwm.

First the module transmits the ID of the window that it is going to manipulate, if this is set to *none* fvwm queries the user to select a window for the operation to be carried out upon. Next the length of the command is transmitted, followed by the command. The module finally sends a one if it intends to keep running or a zero if it is going to exit immediately. This is done using the `Sendfvwm` function.

There are also two built-in functions `Send_WindowList` and `Send_ConfigInfo`. `Send_WindowList` causes fvwm to transmit everything that it currently is thinking about to the requesting module. This information contains the paging status, current desktop number, position on the desktop, current focus and, for each window, the window configuration, window, icon and class names and if the window is currently iconified, the icon location and size. `Send_ConfigInfo` causes fvwm to transmit all commands it has received which begin with a '\*', as well as the pixmap path and the icon path.

The information that fvwm passes can be controlled using the `Set_Mask` function. This is sent followed by a number which is a bit-wise OR of the packet types the module wishes to receive.

#### 3.5.3.4. FVWM-to-Module Communication

Fvwm communicates with modules in one of two modes, broadcast or module specific. Some important messages, such as important window or desktop manipulations are broadcast to all modules whether they request them or not. Modules can also request information about current windows using the `Send_WindowList` function and in this case only the requesting module receives the information. The packets are read by the module using the `ReadFvwmPacket` procedure.

Packets sent from fvwm to modules conform to a standard format, so modules which are not interested in certain messages can ignore them. The packet is made up of four header fields and a variable length data field. The four header fields are

1. This is always 0xffffffff. This is so that modules can re-synchronise to the data stream.
2. The packet type. This field contains a packet type as defined in the fvwm module.h file.
3. Total length of the packet, in unsigned long integers, including the header.
4. The last time stamp received from the X server in milliseconds.

The body information depends on the packet type and is described in Appendix B.

#### 3.5.3.5. Finding Current Window Information from Fvwm

There is no simple way to get information on a single window from fvwm using the module system. The module must send the `Send_WindowList` command to fvwm and then extract the information for the current window from the packets that fvwm replies with. The method used in the sample `FvwmIdent` module is used. This sets the packet mask to the types of packet which contain window information, sends the `Send_WindowList` command and loops reading the packets, taking packets pertaining to the window of interest. The information gathered is the window name, icon name, and the target window structure described in Appendix B.

## 4. Analysis and Design

### 4.1. GOALS AND APPROACH

The goal of this project was to design a simple voice control window management system for the X Window system. The resulting speech system should be easy to re-train for different users, and should be easy to use in practice.

The approach to the project was in an object-oriented fashion but not following any strict object-oriented methodologies. The project was firstly split into two major parts

- Speech Recogniser.
- Window Manager Interface

Each part was approached separately with the intention that one part would pass the command to the other allowing the speech recogniser to be re-used.

#### *4.1.1. Speech Recogniser*

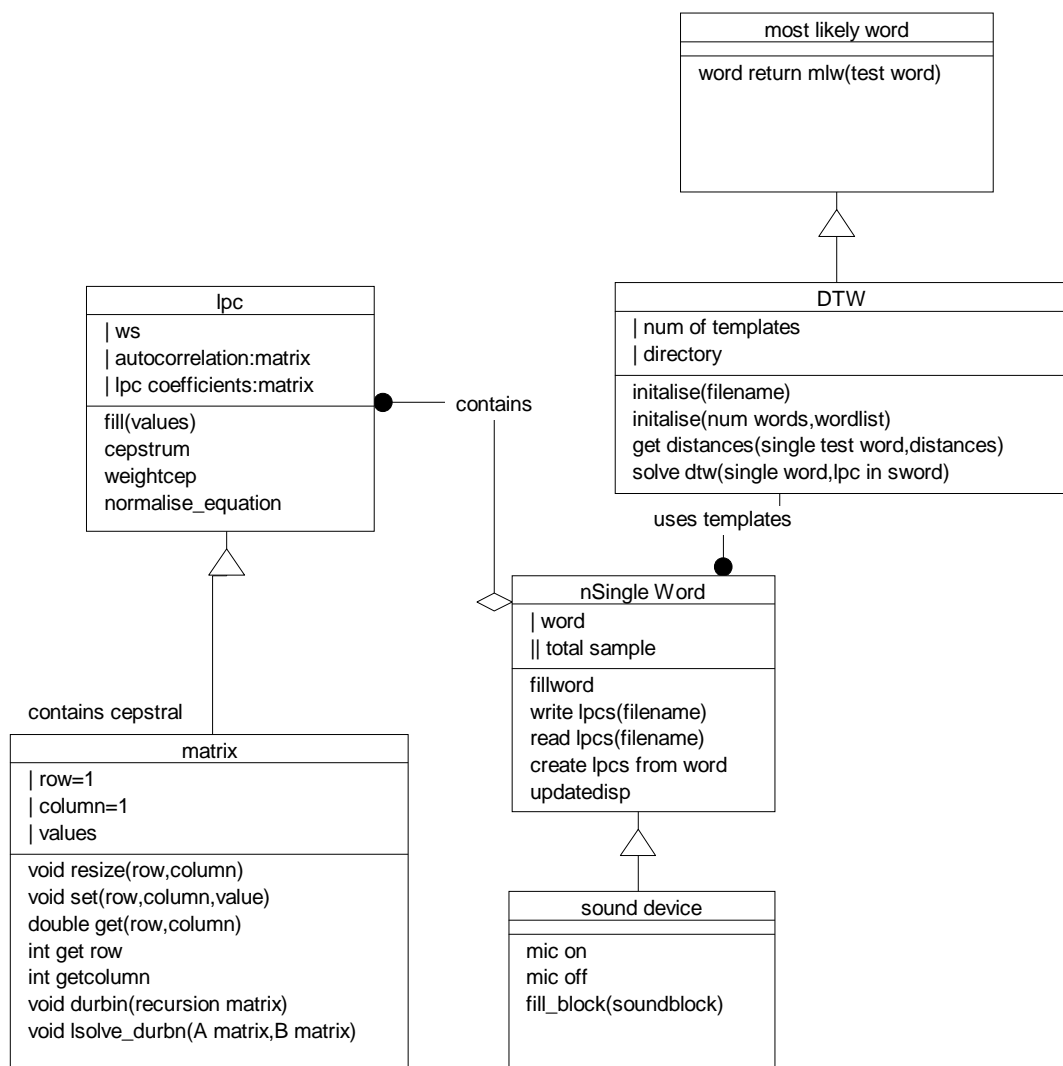
The speech recogniser portion sets up the sound device with the parameters for the project. It listens to the microphone for a word to recognise and detects when a word is spoken. The samples for the complete word are stored into an array, the LPC processing is performed on this array to produce the LPC vectors for the frame which it then stores. These vectors are fed into the dynamic time warping system where the unknown word is

compared to the known reference patterns, and outputs the most likely word to have been spoken which it passes to the second part of the project.

This part of the project also includes building a trainer, which gets the user to say each word a number of times and stores the LPC vector reference patterns on disk for usage in the recogniser DTW system. The trainer has a GUI front-end for ease-of-use.

#### 4.1.2. Window Manager Interface

This section of the project initialises the module interface with fvwm, displays a button to enable the microphone to be switched on and off. It waits to receive a word from the first section of the project, which it translates into an fvwm command or Xlib function call on the current window, transmits the command to fvwm or executes the Xlib function call, and continues.



**Figure 4-1 OMT Object Model of the System**

## 4.2. ANALYSIS OF SPEECH RECOGNISER

The speech recogniser contains six major objects from figure 2-8.

### 4.2.1. Sound Device Object

The program has to initialise the sound device to set it up properly for the recogniser. The values used for the project were

**Sampling Rate:** 8000 Hz

**Bits per sample:** 8

**Stereo or mono:** mono

These were selected to minimise calculations required and also for cross-platform compatibility. The sound-card object has to make sure that the sound device is only opened and initialised once and all other references use the same file descriptor.

It also needs to be able to fill out a sound buffer object. This is a simple structure which contains the values sampled from the sound device along with the energy of the sample. The sound device object devised also controls the microphone status and can turn the microphone on and off.

This object should be the only which communicates directly with the sound device. This level of abstraction allows the project to be more easily ported to other systems with different sound hardware as only this object needs to be re-written.

### 4.2.2. LPC and Matrix Objects

A typical LPC processor for speech recognition applications was used. This consisted of some basic steps.

- 1. Preemphasis :** The digitised speech signal is put through a low-order digital system, to spectrally flatten the signal and to make it less susceptible to finite precision effects later in the processing. The most widely used preemphasis network is a fixed first-order system

$$H(z)=1-\tilde{a}z^{-1}, \quad 0.9 \leq a \leq 1.0 \dots\dots\dots (4.1)$$

In this case the output of the preemphasis network,  $\tilde{s}(n)$ , is related to the input of the network  $s(n)$ , by the difference equation

$$\tilde{s}(n)=s(n)-\tilde{a}s(n-1). \dots\dots\dots (4.2)$$

A common value for  $\tilde{a}$  is 0.95.

**2. Frame Blocking :** The preemphasised speech signal  $\tilde{s}(n)$  is blocked into frames of N samples, with adjacent frames being separated by M samples.

If the  $\ell^{th}$  frame of speech is denoted by  $x_\ell(n)$  and there are L frames within the entire speech signal then,

$$x_\ell(n) = \tilde{s}(M\ell + n), \quad n=0,1,\dots,N-1, \ell=0,1,\dots,L-1. \dots\dots\dots(4.3)$$

Typical values for N and M are 240 and 80 when the sampling rate is 8kHz. This corresponds to 30 ms frames separated by 10 ms.

**3. Windowing :** Each individual frame needs to be windowed to minimise the signal discontinuities at the beginning and end of the frame. The window is used to taper the beginning and end of the frame to zero. If such a window is defined as  $w(n)$ ,  $0 \leq n \leq N-1$ , then the result of the windowing on the signal is

$$\tilde{x}_\ell(n) = x_\ell(n)w(n), \quad 0 \leq n \leq N-1. \dots\dots\dots(4.4)$$

A hamming window is normally used for the LPC autocorrelation ..... method which has the form

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2n\pi}{N-1}\right), \quad 0 \leq n \leq N-1. \dots\dots\dots(4.5)$$

**4. Autocorrelation Analysis :** Each frame of windowed signal is autocorrelated to give

$$r_\ell(m) = \sum_{n=0}^{N-1-m} \tilde{x}_\ell(n)\tilde{x}_\ell(n+m), \quad m=0,1,\dots,p. \dots\dots\dots(4.6)$$

where p is the order of the LPC analysis. This is typically set to 8 for most applications.  $R_\ell(0)$ , is as a side effect of the autocorrelation the energy of the  $\ell$ th frame.

**5. LPC Analysis :** LPC Analysis converts each frame of p+1 autocorrelations into an ‘‘LPC parameter set’’. The set may be one of LPC coefficients, reflection or PARCOR coefficients, the log area ratio coefficients, cepstral coefficients, or any desired transformation of the above sets. The method for converting from autocorrelation coefficients to LPC parameter set is known as Durbin’s method. This is formally given as

$$E^{(0)} = r(0) \dots\dots\dots(4.7)$$

$$k_i = \left\{ r(i) - \sum_{j=1}^{L-1} \alpha_j^{i-1} r(|i-j|) \right\} / E^{(i-1)}, \quad 1 \leq i \leq p \dots\dots\dots (4.8)$$

$$\alpha_i^{(i)} = k_i \dots\dots\dots (4.9)$$

$$\alpha_j^{(i)} = \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)} \dots\dots\dots (4.10)$$

$$E^{(i)} = (1 - k_i^2) E^{(i-1)}, \dots\dots\dots (4.11)$$

where the summation in Eq. (4.8) is omitted for i=1. This set of equations are solved recursively for i=1,2,...,p, and the final solution is given as

$$a_m = \text{LPC coefficients} = \alpha_m^{(p)}, \quad 1 \leq m \leq p \dots\dots\dots (4.12)$$

$$k_m = \text{PARCOR coefficients} \dots\dots\dots (4.13)$$

$$g_m = \text{log area ratio coefficients} = \log \left( \frac{1 - k_m}{1 + k_m} \right) \dots\dots\dots (4.14)$$

**6. LPC Parameter Conversion to Cepstral Coefficients :** An important LPC parameter set is the LPC cepstral coefficients, c(m), which can be derived directly from the LPC coefficient set. The recursion involved is

$$c_0 = \ln(\sigma^2) \dots\dots\dots (4.15)$$

$$c_m = a_m + \sum_{k=1}^{m-1} \left( \frac{k}{m} \right) c_k a_{m-k}, \quad 1 \leq m \leq p \dots\dots\dots (4.16)$$

$$c_m = \sum_{k=1}^{m-1} \left( \frac{k}{m} \right) c_k a_{m-k}, \quad m > p, \dots\dots\dots (4.17)$$

where  $\sigma^2$  is the gain term of the LPC model. This set of coefficients have been shown to be more robust and reliable for speech recognition applications than the other sets. Generally Q > p coefficients is used, with typical value of Q being 12.

4.2.2.1. Structures needed for LPC processor

The most basic object required for the LPC processor is a matrix object. A generic matrix object can be used to store the auto-correlation analysis, the LPC coefficients and the cepstral coefficients. The auto-correlation analysis matrix needs to be p x p where p is the order of the LPC, the LPC coefficient matrix is p x 1 and the cepstral coefficients are stored in a Q+1 x 1 matrix where Q is the number of coefficients.

A generic matrix class was implemented to store these, with the ability to use the Durbin algorithm to solve the Toeplitz matrix produced in the auto-correlation/LPC analysis.

In the above object model, the LPC object inherits from the matrix object. The cepstral coefficients are stored in the objects inherited storage, while two more matrix objects must be instantiated to store the LPC coefficients and the auto-correlation analysis. The `fill` function takes a frame of speech, i.e. a fixed number of values depending on the LPC values selected, and creates the LPC coefficients and cepstral coefficients for that frame. The `weightcep` method calculates the weight cepstral coefficients. The `normalise_equation` method is used internally to fill out the Topelitz matrix.

#### ***4.2.3. Single Word Object***

The single word object is mainly a storage object. It stores a complete single word as LPC vectors or samples from the sound device. It is used to store the reference templates for the DTW system and for creating words from the sound device.

The `fillword` method waits for a word to be spoken on the sound device and stores the individual samples into the `total_sample` buffer. The `createlpcs` method converts the currently stored individual samples into a collection LPC objects. This involves splitting the samples into frames, normalising the samples and passing the frames to the LPC objects' `fill` methods. This object also contains the method to write the LPCs to a file on disk for use in the trainer and for reading them back in the recogniser.

#### ***4.2.4. DTW object***

The dynamic time warping system needs to be able to compare a template given to it with its list of reference templates and calculate the distance measure after time alignment between the templates. The lowest of these can be selected as the most likely word for passing to the next part of the system. The reference templates are stored on disk by the trainer and read in by the dynamic time warping object during initialisation.

The DTW object uses single word objects as its templates. It has a directory on the local drive which stores the word templates created by the trainer. It receives a wordlist upon initialisation and reads in the words to check against from this. It then reads in the templates from the templates directory into single word objects. The `get_distances`

method takes two arguments, one is the test template to check against and the other is used to return the distances to each reference template from the comparison.

#### **4.2.5. Most Likely Word Object**

This is the highest level object of the word recogniser. It inherits the DTW object and is initialised with a filename containing the words it is to recognise. The single method `returnmlw` is passed two parameters, one is the unknown template which is a single word object and a string in which to return the most likely word to have been contained in the test template.

### **4.3. ANALYSIS OF TRAINER**

The trainer opens a wordlist and gets the user to speak each word a set number of times. It then runs the LPC processor over the word and stores the resulting vectors onto disk for reading by the recogniser. It re-uses most of the structures used in the recogniser to reduce implementation effort. The main part of the trainer is the GUI, which provides the ability to select a wordlist file and the number of templates per word and provides the user with the ability to train all the words or just one individual word template. The single training mode just instantiates a single word object, the train all mode instantiates a new train object which trains all the words given the word-file.

The trainer also contains the directory on the disk which stores the reference patterns.

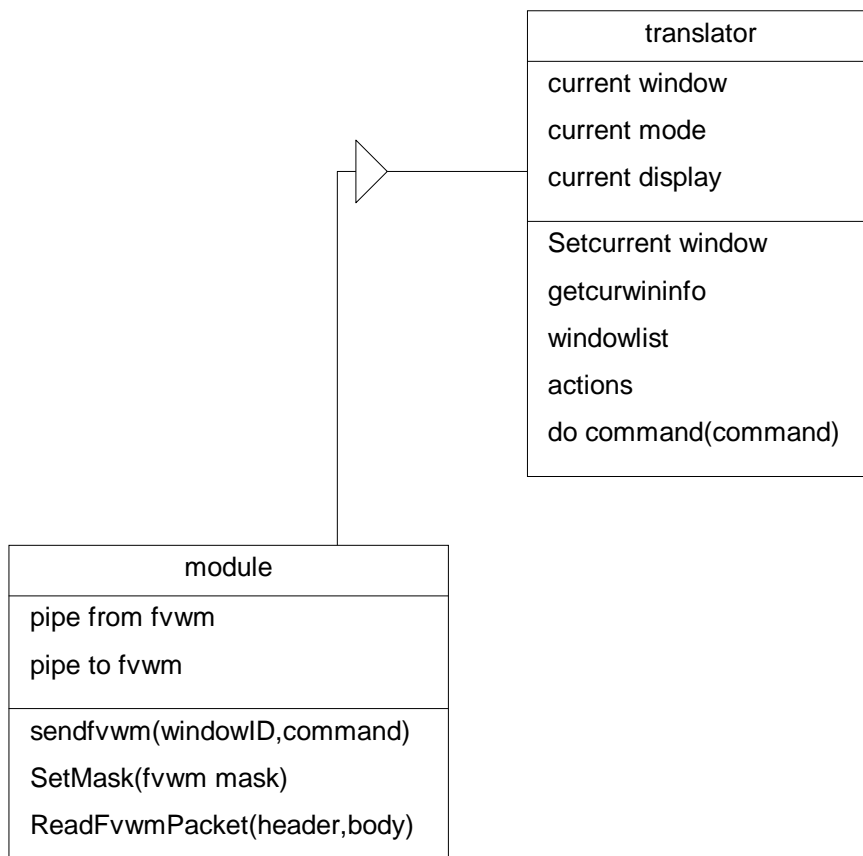
### **4.4. ANALYSIS OF WINDOW MANAGER INTERFACE**

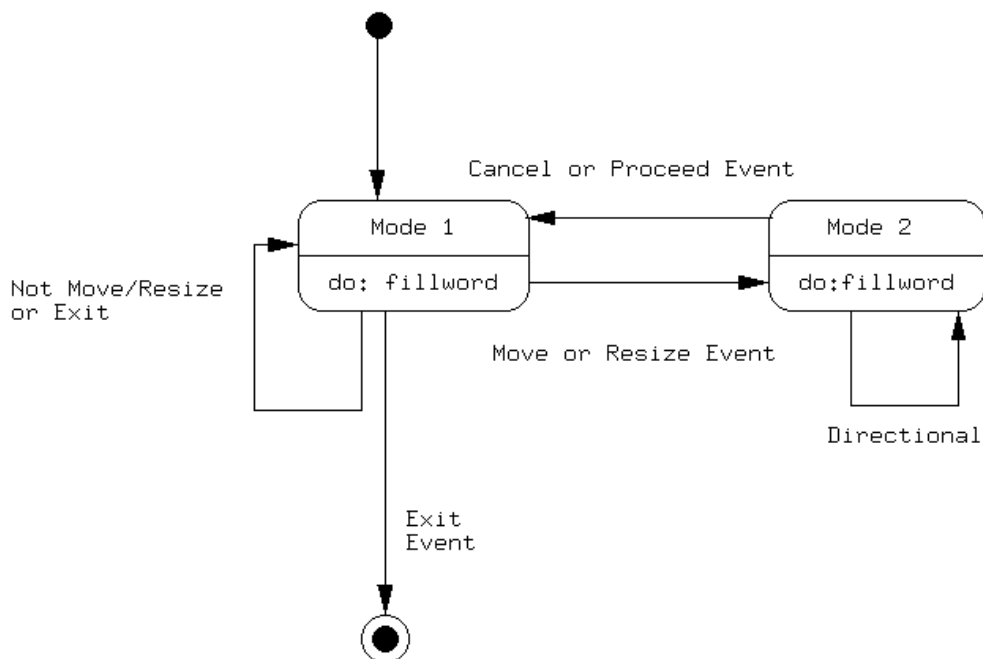
The window manager interface portion of the code is made up of two objects as shown below. The module object contains all the necessary methods to handle the interface to `fvwm`. It is initialised with the two file descriptors for the open pipes which it then uses internally to handle the low-level `fvwm` interface. The three methods

`Sendfvwm`, `SetMask` and `ReadFvwmPacket` are implementations of the three functions mentioned in Chapter 3.5.3.

The translator object which inherits the module object is the main object on the interface side. It contains methods listed in the object diagram as a single method actions which when called carry out an action on the current window. These are actions such as `resize` and `destroy`. The methods described as `windowlist` are used to get information on the current window. The `setcurrent_window` invokes the `windowlist` methods to set the `current_window` attribute. The `do_command` method is invoked by the main program loop with the command received from the recogniser. The `do_command` method translates this command to one of the actions and this action is carried out on the current window.

The `current_mode` attribute stores which of the two operating modes the translator is currently in. The translator has two levels of operation; In mode one operation, normal mode, it accepts the commands from Appendix C, Table 1. These are the top level command mode commands. If it receives either `move` or `resize` in this mode it switches to the second mode and accepts the commands from Appendix C, Table 2. These are the directional commands along with escape routes back to mode one. A state diagram showing this structure is shown in Figure 4-3.



**Figure 4-2 OMT Object diagram of window manager interface****Figure 4-3 State Diagram for operating modes.**

## 4.5. OVERALL SYSTEM

The overall system uses a translator object and two most likely word objects. The two most likely words correspond to the two modes of operation of the translator. This is done in order to reduce computation in the DTW module. If the user says “left” at level one it is not recognised as a command word at that level. The two most likely words are initialised with separate wordlists but share the same template directory.

The main program also has to place a button on the screen to switch the microphone on and off. This is done using a separate process due to the problems with polling the microphone and allowing X refresh the window. The processes communicate using pipes and pass simple messages to and from each other.

## **5. Development**

### **5.1. PROGRAMMING LANGUAGE**

C++ was selected as the main language to be used in this project. Its object-oriented architecture allows for code re-usability and abstraction, e.g. the object to access the sound device can be replaced to port the system to another platform.

The GUI for the training system was written in Tcl/Tk. Tcl is an interpreted scripting language and Tk adds X programming ability to the scripting language. It can be easily integrated with C/C++ through its C API. The main program button was written in C using the Xaw widget set. LEX was used to parse the command from the voice system to the module code.

### **5.2. OPERATING SYSTEM**

The Linux operating system was the main development environment. Linux is a free UNIX clone available originally on the x86 family of processors but has recently been ported to many other architectures. The distribution used were from Redhat versions 3.0.3 and 4.1. Some development work was also carried out on a Sun UltraSparc 1 running Solaris 2.5, a SUN SparcClassic running SunOS 4.1.3 and an IBM RS/6000

running AIX. These were mainly used to debug some of the slower parts of the project and profile them.

### 5.3. DEVELOPMENT ENVIRONMENT

The main editor used during the implementation of the system was the Xemacs editor, an extended version of the popular GNU Emacs, which has better support for the X Window system. Standard Emacs provides a multi-buffer editing system and programming language modes. The Xemacs program also provides syntax colouring for many languages but also better multi-frame support than the original. Compilation under emacs occurs in a sub-process and emacs parses the errors from the compiler in its own buffer. The `vi` and `vim` (Vi IMproved) editors were also used for smaller code changes and the writing of test code.

### 5.4. COMPILATION SYSTEM

The GNU C (GCC) version 2.7 compiler was used to compile most of the program source code. This is the standard compiler available on Linux systems and is also available on nearly every operating system available. It can compile programs written in C, C++ and Objective C. The C++ mode of the compiler was used (`g++`) for compilation of the C++ code. Some code was also compiled with SunPro SparcWorks C/C++ compilers for the Sun Sparc systems.

GNU `lex` (`flex`) was used to create the `yy.lex.c` file from the lex code.

The main debugger was the GNU debugger (`gdb`). This allows full C/C++ debugging and also using the core dump facilities on UNIX can be used to backtrace a crashed out program to see where it crashed. It also allows itself to be attached to a running process which is useful for checking the difference between unending loops and slow code.

The GNU profiler (`gprof`) was used to profile the code to check where it was spending most of its time. This was important in finding what parts of the code needed to be optimised. It reports back how often each function was called, how long each call

took, how long was spent in total in the function, the number of functions called by a function and also the number of times a function was called by other ones.

GNU make was used to facilitate the compilation of the separate parts of the code into executables. Special features of GNU make to allow compilation of sub-directories of code provide for a cleaner toplevel source directory.

## 6 Implementation

### 6.1. INTRODUCTION

This chapter discusses the implementation of the project. It describes how individual parts of the code were implemented and also how the overall system integrates as well as problems and optimisations in the code.

### 6.2. SPEECH RECOGNISER

#### *6.2.1. Programming the Linux Sound Device*

The Linux sound device is part of the Linux kernel. It was written originally by Hannu Savolainen. The driver uses device files in `/dev` to access the soundcard once support has been compiled into the kernel for the particular card. The two main device files the concerned this project were `/dev/dsp` and `/dev/mixer`.

##### 6.2.1.1. /dev/dsp

The dsp device is the main device file for digitised voice applications. Data written the device is fed directly to the DAC/PCM/DSP hardware on the soundcard. Reading from the device returns the audio data recorded from the current input source. The device reads initially in 8-bit unsigned linear format. It has a counterpart device `/dev/audio` which reads 8-bit logarithmic mu-law encoding like Sun machines standard `.au` files.

The device is opened using the standard `open()` system call and be opened for read (`O_RDONLY`), write (`O_WRONLY`), or both (`O_RDWR`). Once opened the system calls `read()`, `write()` and `close()` may be used on the device. The `ioctl()` system call is used on the device to set the format, sampling rate, stereo or mono modes, and also to re-synchronise or reset the device.

#### 6.2.1.2. /dev/mixer

The mixer device controls the soundcards onboard mixer device. A mixer is used to adjust playback and recording levels of the various sources connected to the card. The device file is used to control the volumes of the devices and also which devices to record from.

The mixer is also a device file which can be opened with `open()`. The `ioctl()` call is again used to set the various options on the mixer device.

#### 6.2.1.3. OSS API

The Linux sound system was the basis for the Open Sound System which is a generic API for sound devices available for most UNIX systems [5]. There is a programmers guide available for the OSS API [6].

### **6.2.2. Linux Soundcard Object**

On Linux systems the sound device can only be opened once by any application and only one application may be using the device. In order to allow each single word object to use the sound device, the first initialised sound card object, stores an internal reference to itself in a static pointer. This static pointer is checked whenever a new sound card object is initialised and if it is not NULL the sound object does not attempt to open the devices again. The single word object initialises a new sound card object by having the new sound card object point to the internal reference pointer thus giving it access to the original sound card object.

#### 6.2.2.1. Object Constructor

The constructor for the sound device object begins by checking the static internal reference to see if it is the first sound card object to be initialised. If not it returns, if so, it opens the sound device through the `/dev/dsp` device, sets the format to 8 bits, sets

mono mode, opens the mixer device, sets the speaker volume to zero, and turns off the microphone. It finally sets the internal reference to itself.

#### 6.2.2.2. Object Destructor

The destructor closes the dsp and mixer devices.

#### 6.2.2.3. Fill Block Method

The `fillblock` method takes one parameter which is a reference to a `sndblk` structure which is simply a store of the 8-bit samples taken from the sound device along with number of samples and the energy of the total sample. This method takes in the number of samples, using the `read()` system call, specified in `size_` attribute of the `sndblk` structure from the sound device and places them into the `sndblk` buffer. It also calculates the total energy of the sample and stores that in the structure.

#### 6.2.2.4. Microphone controls

The microphone control methods `deaf_mic` and `full_mic` simply set the microphone recording level using the mixer device to either zero or full.

#### 6.2.2.5. Reset Buffer

The `empty_buffer` method simply resets the sound devices on-board PCM buffer.

### **6.2.3. Matrix Object**

A generic matrix object which could store an  $n \times n$  matrix of doubles was implemented. This used two integers, `row` and `col` for storing the dimensions and a pointer, `first`, to point to the first entry in the matrix.

#### 6.2.3.1. Constructor

The matrix object constructor takes three parameters, the number of rows, number of columns and initial value of the matrix entries. These values default to one row, one column and 0.0 as the initial value. The matrix constructor allocates the dynamic memory necessary to store the total number of values and points `first` to it.

### 6.2.3.2. Destructor

Frees the memory allocated in the constructor.

### 6.2.3.3. Resize method

This method is used to resize and already allocated matrix. It takes rows and columns as parameters and simply destroys the old matrix and allocates a new one.

### 6.2.3.4. Operator Methods

The = operator is overloaded to allow assignment of one matrix to another. This is necessary due to the dynamic memory allocation. The ( ) operator is overloaded to allow assignment to and reading from individual matrix entries. This also bounds checks the requested values to stop illegal memory accesses. The `set` and `get` methods provide functionality similar to the ( ) overloading.

### 6.2.3.5. Durbin Solving Methods

The `lsolve_durbn` method takes the two recursion matrices from Eq. (2.23) and solves them for the  $a_n$  coefficients. It uses the `durbn` method as part of this which is an implementation of the Durbin method described in Eqs. (4.7-4.11).

## **6.2.4. LPC object**

The `lpc` object inherits the `Matrix` object. It stores the final cepstral coefficients in the inherited object and also requires three more matrix objects which are the three matrix objects from eq. (2-23) in the form  $AC=B$ . Matrix `C` stores the LPC coefficients. The `numq` attribute stores the number of cepstral coefficients which is set to 12 in this project.

### 6.2.4.1. Constructor

The constructor takes two values `p` and `q`, `p` is the order of the LPC analysis and `q` is the number of cepstral coefficients. The constructor resizes the `A`, `B` and `C` matrices to `p x p`, `p x 1` and `p x 1` respectively. The default values are 10 and 12 respectively for `p` and `q` from Section 4.4.2. The `lpc` object has no explicit destructor.

### 6.2.4.2. fill Method

The `fill` method is the main external interface to the `lpc` object. It accepts a

pointer to an array of double values which make up the normalised values for the current frame. It sets up the A and B matrices from these values and uses the `lsolve_durbn` method to solve for C. It then constructs the cepstral coefficients from the LPC coefficients using the `cepstrum` method.

#### 6.2.4.3. Cepstrum Method

The `cepstrum` method construct the cepstral coefficients from the LPC coefficients stored in the C matrix. It implements Eqs (4.15-4.17).

#### 6.2.4.4. *normeqn\_ac* Method

This method creates the A and B matrices from the normalised samples. It implements eq. (4.6).

#### 6.2.4.5. *Miscellaneous Methods*

The `setws` method sets the length of the frame, N for use in the `normeqn_ac` method. The `fin` and `fout` methods accept file streams, input and output respectively, as parameters and read or write the cepstral values to or from the stream in a space delimited format.

#### 6.2.4.6. *norm\_s2d* Macro

This macro is included in the `lpc` object header file for convenience. It converts an unsigned char in the range 0->255 and normalises it to the range -1 to 1 and returns the result in a double.

### **6.2.5. *sword* Object**

The `sword` object is the most complex object used in the speech recogniser. It inherits from the `soundcard` object so that it can use the internal reference to gain access to the sound device that has previously been opened. It contains a pointer to an unsigned char array in which it stores the word straight from the soundcard. It has a pointer to an `lpc` array where the `lpcs` for the individual frames of speech are stored.

#### 6.2.5.1. *Constructor*

The constructor simply sets the pointers to NULL. There is no explicit destructor.

#### 6.2.5.2. fillword Method

The `fillword` method is the most complex method in this object. Its main function is to read a word in from the sound device and store the samples to its internal buffer. It begins by switching on the microphone, and de-allocating any previous allocations. It begins by deciding when a word has been spoken. This start point detection routine is a simple routine. It creates a `sndblk` structure and over a 200ms time period, passes the block to the `soundcard` object to fill, and stores the energy into a three element buffer which holds the last three energies. After the 200ms period the final three energies are averaged and this value is taken to be the background noise level.

The program creates an array of `sndblks` to store the word. It loops reading in a block from the device and calling the `updatedisp` method described below, until the block energy is greater than the background noise level. If the call to `updatedisp` returns -1 the `fillword` method exits, this is to allow a derived class signal an error. When this occurs it moves the current block into the array and gets the next one. Final point detection occurs when the background noise energy level is reached for a fixed number of sound blocks. Finally the buffers from the `sndblks` are copied to the one large internal buffer in the `sword` object and the number of samples is set.

#### 6.2.5.3. updatedisp Method

This is a virtual method which is used by a derived object to allow the program do something else like update the display while it is polling the microphone. This method does nothing in the base class. A derived class must return either -1 or 0 from this method, -1 forces `fillword` to exit and 0 allows `fillword` to continue.

#### 6.2.5.4. createlpcs Method

This method dynamically allocates the number of `lpc` objects needed to store all the `lpc` vectors for the word. It then frames the samples from the `fillword` method into blocks of `M` samples, where `M` is the width of a frame. It normalises each 8-bit sample using the `norm_s2d` macro and places them into a temporary double pointer. It then calls the `lpc` objects `fill` method with the double pointer. It repeats this for each frame.

#### 6.2.5.5. Miscellaneous Methods

The `readlpcs` and `writelpcs` take a filename and call the `lpc::fin` and `lpc::fout` methods to read and store the vectors for a word to disk. Trivial methods, `getword` and `setword` are used to set an internal string to the word that the `sword` object contains. The `getnum` method returns the number of LPC vectors used to store the word.

#### **6.2.6. dtw Object**

The `dtw` object contains an array of strings to store the word list for itself, `wlist` and an array of `sword` objects, `templates`, containing the reference templates. It also contains the template storage directory in a constant variable `t_dir`, the number of templates per word and the number of words.

##### 6.2.6.1. Constructor

There are two constructors for the `dtw` object. The first takes a filename. The file has the number of words contained in it followed by the words. It reads the words in and stores them into a temporary array and calls the `init` method with the list of words and the number. The second constructor takes an array of strings containing a list of words and the number of words and passes both straight to the `init` method.

##### 6.2.6.2. Destructor

Frees up all the dynamically allocated memory.

##### 6.2.6.3. init Method

The `init` method takes a list of words as an array of strings and the total number of words in the list. For every word in the list, it copies the word to internal `wordlist`, `wlist`, allocates memory for an `sword` object in the `templates` array. It passes the filename created from the template directory, current word and current template for the word to the `sword::fillword` method.

##### 6.2.6.4. Distance Methods

The `getdists` method simply takes the unknown `sword` object and a pre-allocated array of doubles. It calls for every reference pattern the `sdtw` method with the unknown pattern and stores the returned distance into the array.

The `sdtw` method takes the unknown pattern and an index into the reference patterns. It implements eqs (2.35-2.38) and returns  $D^*$ .

The distance measure used is the Euclidean distance measure and the `norm` method calculates this for two `lpc` objects.

### **6.2.7. *mlw* Object**

This is a simple object that inherits from the `dtw` object. It has the same constructors as the `dtw` object. It has one method which simply takes the unknown pattern and a string and returns in the string the most likely word to have been spoken. It does this by getting the minimum distance measure and taking the word that corresponds to it from the `wlist` array.

## **6.3. WINDOW MANAGER INTERFACE**

This section describes the design and implementation of the interface to `fvwm` and the translator from the window manager to the module system.

### **6.3.1. *Modules under C++***

The `fvwm` module system was designed so that modules could be written in any language and although complete support for this is not yet implemented, interfacing the module system with either C or C++ is a trivial task. This is done with `module` object which implements all the `fvwm` interface functions as methods. The object's constructor takes the file descriptors and stores them to the internal `fd` array.

### **6.3.2. *Using lex for converting string to enum***

The command string received from the speech recogniser portion of the project needed to be converted to something that could be used in a `switch/case` statement like an enumerated type. This was done using a method in the translator object `lexit` which accepts a string and return a token enumerated type. This used the `lex` ability to scan strings using `lex_yy_string` to set up the string for scanning and `yylex` to scan it.

### ***6.3.3. Translator Object***

The translator object contains, two `target_struct` window structures, `origwin` and `target` to store the original window and current window states, the original one is used during move and resize operations. The current display, `disp`, is stored as a `Display *`, which is defined in Xlib, and the current window, `app_win`, as a `Window` object.

#### *6.3.3.1 Constructor*

This opens the current X display, with the Xlib `XOpenDisplay()` function and stores it internally for use with later Xlib calls. It sets the `fvwm` packet mask to nothing, and the current mode to the basic first level mode.

#### *6.3.3.2. Destructor*

Closes the X display.

#### *6.3.3.3. Setappwin and GetCurWinInfo Methods*

The `Setappwin` sets the internal `app_win` to the window that currently has the input focus using the Xlib, `XGetInputFocus()` function.

The `GetCurWinInfo` method fills out the internal `target` structure using the methods described in Section 3.5.3.5. This needs six other internal functions, `Loop`, for looping through the packets, `proc_msg` for processing the packets and `list_window_name` for dealing with window names, `list_icon_name` for dealing with the icon name, `list_configure` for dealing with the `target_struct` information and `list_end` for setting the `fvwm` mask back to 0.

#### *6.3.3.4. Action Methods*

These methods are all similar with only `move` and `resize` having one difference. The action methods all call `Setappwin` and then invoke `sendfvwm` with the current window and a command for `fvwm` to act on, e.g. the `delete` method calls `Setappwin` and then sends “delete” using `sendfvwm` and `fvwm` deletes the window from the display.

The `move` and `resize` methods, send the “stick” command to `fvwm` which makes `fvwm` draw lines across the title bar. They then set the current mode variable to `T_MOVE` or `T_RESIZE` modes which are the level two modes.

#### 6.3.3.5 *cancel and proceed Methods*

These methods are invoked when the level two commands `cancel` and `proceed` are encountered. They both send the “stick” command to `fvwm` to remove the bars from the title bar and they both reset the mode to `T_BASIC`, but “cancel” returns the window to its original state and “proceed” leaves it in its new state.

#### 6.3.3.6. *direction Method*

This method accepts an integer which corresponds to a level two command. It gets the current window information and builds up the command to send to `fvwm`. Depending on whether it is in `T_MOVE` or `T_RESIZE` mode the first word of the command strings is set to “move” or “resize”. It then using the geometry information stored in the `target_struct` works out the new geometry for the window and sends the command to `fvwm`. If it receives the `cancel` command it puts back the original window geometry stored in the `origwin` structure.

#### 6.3.3.7. *command Method*

The `command` method is the main interface to this object. It takes a string as a command. This is passed to the lex parser described above and a `switch/case` statement is used to decide the action to take depending on what mode the system is currently in. It calls the action methods or `directcom` methods depending on the current mode of the translator object.

## **6.4. COMBINED SYSTEM**

### ***6.4.1. Combining System***

The two pieces of the system were quite easily combined due to their modular design. The main program instantiates a new translator object with the two pipes passed to the program by `fvwm`. It creates a new `soundcard` object which initialises the

sound device for all the other `soundcard` objects. It then creates two `mlw` objects, one for each command level and passes them the wordlists for the levels.

#### ***6.4.2. X Microphone switch***

Implementation of the on screen microphone switch was difficult due to the problem of having both Xt having its own main loop and the recogniser wanting to poll the sound device as well. This was resolved by using two processes, the parent containing the speech recogniser and the child containing the X portion of the code.

The main program checks for `fvwm` starting it and then creates two UNIX pipes using the `pipe()` system call. Pipe A is for messages from the child to the parent and Pipe B is for messages from the parent to the child. It sets up signal handlers for the `SIGPIPE` and `SIGCHLD` signals. The `SIGPIPE` error handler throws up an error message and exits immediately. The `SIGCHLD` handler calls the UNIX `wait()` system call to allow child processes exit code to be caught and then exits. After setting up these handlers, the main program calls `fork()`.

The parent process closes the write end of pipe A and the read end of pipe B. It uses the `fcntl()` system call to set pipe A to do non-blocking I/O. This means that any reads done by the process on an empty pipe will not make the process wait for something to appear in the pipe, it will just continue processing. It then enters the `doNonX` function.

The child process closes the read end of pipe A and the write end of pipe B, and calls the `doXt` function.

##### ***6.4.2.1. Pipe Protocol***

Pipe A from the child to the parent is used to tell the parent that the user has clicked on the button to change the microphone on/off toggle. If the microphone is being turned on, the string “ONN”<sup>1</sup> is sent, if off the “OFF” string is sent.

Pipe B is not currently used but could be used to send the word that was spoken to the X portion of the program for displaying in a text box.

##### ***6.4.2.2. xsword Object***

---

<sup>1</sup> This is ONN and not ON, this was just to have the messages both the same length.

The `xsword` object is inherited from the `sword` object. It overloads the `updatedisp` and constructor of `sword` and creates storage for the pipe. The constructor takes one argument which is the file descriptor of the pipe to send the messages to which it stores in the `readpipe` internal storage variable.

The overloaded `updatedisp` reads data from the pipe and checks for ONN and OFF messages. It keeps reading until the pipe is empty as the button may have been pressed multiple times and the program is only interested in the current state. If the current state is on, or the pipe is completely empty, it returns 1 which causes the `fillword` method to continue. If the status is off, then it returns 0 which causes `fillword` to exit.

#### 6.4.2.3. *doNonX()* Function

This function contains the main recogniser routines. It initialises a translator object with the pipes from `fvwm`, a soundcard object, two `mlw` objects, one for each level of commands and an `xsword` object to store the unknown pattern in.

It then enters an infinite loop for which the following is the pseudo code,

```
call xsword::fillword to fill out xsword with unknown word
did user turn off microphone?
```

```
Yes?
```

```
    Wait till user turns back on microphone
```

```
No?
```

```
    Create lpcs with xsword::createlpcs
```

```
    Level 1?
```

```
        Call Level 1 recogniser mlw object
```

```
    Level 2?
```

```
        Call Level 2 recogniser mlw object
```

```
    Send command to translator object command method
```

```
    Command method returns new level.
```

```
Loop in current level.
```

#### 6.4.2.4. *doXt()* Function

This is the main function of the child process. It declares two X Intrinsic widgets called `toplevel` and `button` and an X Application Context structure, `app`. It calls `XtAppInitialize` to create the new `toplevel` window and creates a managed X

widget using the Xaw Command widget to make the button. It adds a callback for the button, i.e. the function to call when the button is clicked, and enters the `XtAppMainLoop()` procedure which just processes the events from the X server.

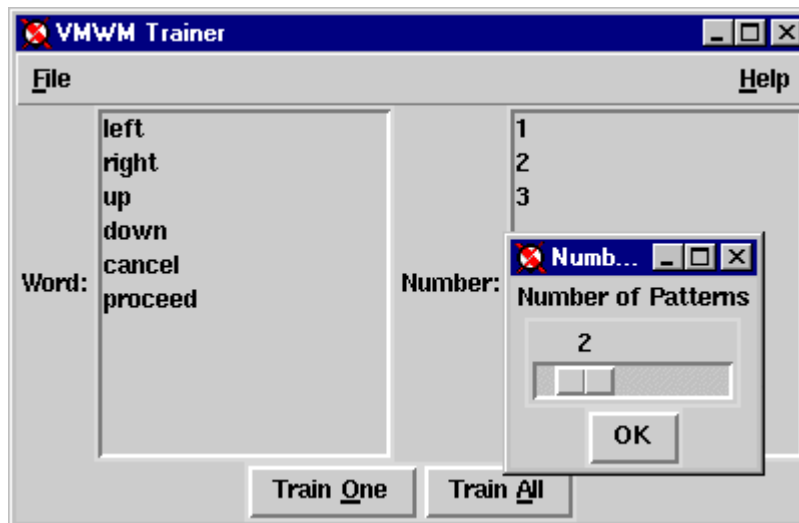
The button callback procedure `mic_switch` is a simple toggle using a static variable to store the current status. It changes the string displayed on the button and sends the "ONN" or "OFF" messages into the pipe.

If the second pipe was being used, the `XtAddInput` procedure would be called to setup a procedure to be called whenever anything arrived on the pipe.

## 6.5. TRAINING SYSTEM

### 6.5.1. GUI

The GUI for the trainer system uses Tcl/Tk. Figure 6-1 shows a screen shot of it running.



**Figure 6-1 Main Trainer Window with number of patterns window open**

#### 6.5.1.1. File Menu

The file menu contains three menu options one of which exits. The other two are:

- Open Wordlist..

This gives a file selection dialog for the user to select a new word list to train. A wordlist is a text file beginning with the number of words followed by the words.

- Patterns per Word

This displays the dialog box as shown in Figure 6-1. This allows the user to set the number of patterns per word.



**Figure 6-2 Recording Dialog Box**

#### 6.5.1.2. Buttons

The TRAIN ALL button pops up the dialog box shown in Figure 2. As each word is spoken the slider moves across one.

The TRAIN ONE button requires the user to select a word and to select a pattern number before being pushed. It pops up a dialog box with the word to be spoken.

#### 6.5.1.3. Help Menu

This contains only the about box at present.

#### **6.5.2. Tcl/Tk to C/C++**

The “train all” and “train single” word functions are written in C and interface to Tcl using the Tcl C API. The main program is written in C/C++ and initialises the Tcl/Tk using the Tcl\_Init and Tk\_Init procedures and gives the Tcl script containing the commands to draw the GUI to the interpreter. It then creates two Tcl commands to correspond to the train functions and hands control to Tcl. From within the Tcl script the C functions are called at the appropriate times.

#### **6.5.3. train\_individ Function**

The train\_individ function is responsible for training a single word. Tcl passes it the word and the template number and an sword object is created, filled and written to the file corresponding to the combination of word and template number.

#### **6.5.4. train Object**

The train object is an object which contains a word list in an array of strings, and the number of words and number of patterns in integers.

#### 6.5.4.1 Constructor

The constructor takes a wordfile as a string and a number of patterns as an integer. It reads in the word file similar to the `dtw` object in the main recogniser and stores them into the word list array.

#### 6.5.4.2. *createwords* Method

This method loops through all the words and all the patterns and creates, fills and saves the `word` object for each one.

#### **6.5.5. *tcetrain* Object**

This object inherits from the `train` object and overrides the `createwords` method for use with Tcl. It displays the dialog box and changes the word and slider on it for every word that is spoken.

#### **6.5.6. *train\_all* Function**

The `train_all` function is responsible for training all the words in the list. Tcl passes it the wordfile and the number of patterns. It instantiates a `tcetrain` object with the wordfile and number of patterns and invokes the `createwords` method.

## **6.6. OPTIMISATIONS**

The code was profiled and the slower parts were identified. A problem with passing large object by value was uncovered and this gave a large speedup to the code when replaced with passing by reference. Removal of the bounds check in the matrix class also improved speed of the code as the overloaded access operator was one of the most called functions. The matrix copy constructor originally used a for loop to copy all the matrix elements, but this was replaced with a `memcpy()` to increase performance.

## **7. Future Development and Conclusions**

### **7.1. INCREASED VOCABULARY**

The system could benefit from a higher vocabulary level for starting up often used programs or the alphabet and digits for insertion into terminals.

### **7.2. USE OF HIDDEN MARKOV MODELS**

Hidden Markov Models could have been used in the recogniser instead of the DTW module. These can provide similar recognition rates and are more extensible into larger systems.

### **7.3. PERFORMANCE**

The performance level of the system could be further increased. It takes about 2 seconds for it to compare 30 templates with an unknown. This could further improved by either using C or maybe assembly language to optimise these pieces of the code.

## **7.4. MULTI-USER**

The project as it stands when installed will run and can be trained by each user individually storing their patterns in their own home directory. This could further be enhanced to give an option button to select which users patterns to use so that users could switch patterns while the program was running.

## **7.5. CONCLUSIONS**

This project shows that designing and implementing a voice control window manager was possible using the Linux operating system. The ability to build onto a pre-existing window manager without having to change any of the code of the original was ideal for such an implementation. The final system is not perfect, but through further fine tuning or usage of a different recogniser the performance level may be further improved.

In conclusion, the project has given the author in-depth knowledge of a wide range of areas, from UNIX programming, X Window and Tcl/Tk programming and the applications of speech recognition algorithms used today.

## 8. References + Bibliography

### 8.1. REFERENCES

- [1] Rabiner L.R. / Levinson S.E.; 1981. "Isolated and Connected Word Recognition - Theory and Selected Applications."; Readings in Speech Recognition Ch 4 P120 Fig 7.
- [2] Rabiner L / Juang BH.; 1993. "Theory and Implementation of Hidden Markov Models"; Fundamentals of Speech Recognition; Ch 6, pg334-348
- [3] Rabiner L. R.; 1989 "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition"; Readings in Speech Recognition Ch 6 P277
- [4] <http://www.hpc.uh.edu/fvwm/FAQ.html> : The official FVWM FAQ. This file changes often and this web reference is the main copy.
- [5] <http://www.4front-tech.com/oss.html>; the Open Sound System for UNIX
- [6] <http://www.4front-tech.com/pguide/index.html>; The preliminary draft of the OSS/Free Linux sound driver programmer's guide. This is work in progress with some sections missing at the moment.

## 8.2. BIBLIOGRAPHY

Readings in Speech Recognition, 1990, Edited by Alex Waibel & Kai-Fu Lee, Morgan Kaufmann Publishers Inc. ISBN 1-55860-124-4

Fundamentals of Speech Recognition, 1993, Lawrence Rabiner and Biing-Hwang Juang, Prentice Hall Signal Processing Series, ISBN 0-13-015157-2

UNIX for Programmers and Users: A Complete Guide, 1993, Graham Glass, Prentice Hall, ISBN0-13-061771-7

X Window Inside & Out, 1992, Levi Reiss and Joesph Radin, Osborne McGraw-Hill, ISBN 0-07-881796-X

Object-Oriented Modeling and Design, 1991, Rumbaugh et al., Prentice Hall, ISBN 0-13-630054-5

C++ Nuts and Bolts for Experienced Programmers, 1995, Herbert Schildt, Osborne McGraw-Hill, ISBN 0-07-882140-1

The usenet Comp.speech FAQ available from

<http://www-svr.eng.ac.uk/comp.speech>.

## Appendix A. Installation Guide

Get fvwm-vcwm.tgz from <ftp://ftp.csn.ul.ie/pub/linux/X/utils/fvwm-vcwm.tgz> or from <http://www.csn.ul.ie/~airlied/fyp/fvwm-vcwm.tgz>.

### *A.1. Unpacking*

Un-tar it using

```
tar -zxvf fvwm-vcwm.tgz if using GNU tar  
or  
gzip -dc fvwm-vcwm.tgz | tar xvf -
```

This should create a subdir called vcwm.

### *A.2. Configuration*

Change directory to the vcwm subdir and edit the Makefile to suit the local system.

The main variables to be changed will be

USER_TEMPLATE_DIR	storage for users templates ~/.snds is recommended
PREFIX	prefix for installation /usr/local is usual
OS	currently only LINUX or OSS may also be SOLARIS or AIX.

Once happy with changes,

```
make
```

### ***A.3. Installation***

```
make install
```

The program by default installs itself into /usr/local/lib/vcwm.

It creates the vcwm executable fvwm module and also the vcwm-train executable containing the trainer and also stores

```
prim.list - Level One Words
sec.list  - Level Two Words
train.tcl - The tcl front end to the trainer.
```

In this directory.

### ***A..4. User specific installation***

The user must edit their .fvwm2rc file and add /usr/local/lib/vcwm to the module path or wherever they have installed it \$PREFIX/lib/vcwm.

They must also add a line to fvwm2 InitFunction section of the file to start the module, similar to the following,

```
+ Module vcwm
```

This will cause vcwm to start up when fvwm2 is started.

## Appendix B. FVWM 2 Module Packet Types

All information in this appendix is from the modules.tex file distributed with fvwm2.

### B.1. #DEFINES FOR FVWM PACKET TYPES

These are taken from the modules.tex file distributed with fvwm2.

```
#define M_NEW_PAGE (1)
#define M_NEW_DESK (1<<1)
#define M_ADD_WINDOW (1<<2)
#define M_RAISE_WINDOW (1<<3)
#define M_LOWER_WINDOW (1<<4)
#define M_CONFIGURE_WINDOW (1<<5)
#define M_FOCUS_CHANGE (1<<6)
#define M_DESTROY_WINDOW (1<<7)
#define M_ICONIFY (1<<8)
#define M_DEICONIFY (1<<9)
#define M_WINDOW_NAME (1<<10)
#define M_ICON_NAME (1<<11)
#define M_RES_CLASS (1<<12)
#define M_RES_NAME (1<<13)
```

```
#define M_END_WINDOWLIST (1<<14)
#define M_ICON_LOCATION (1<<15)
#define M_MAP (1<<16)
#define M_ERROR (1<<17)
#define M_CONFIG_INFO (1<<18)
#define M_END_CONFIG_INFO (1<<19)
```

## **B.2 CONTENTS OF PACKETS**

### **M NEW PAGE**

These packets contain 5 integers. The first two are the x and y co-ordinates of the upper left corner of the current viewport on the virtual desktop. The third value is the number of the current desktop. The fourth and fifth values are the maximum allowed values of the co-ordinates of the upper-left hand corner of the viewport.

### **M NEW DESK**

The body of this packet consists of a single long integer, whose value is the number of the currently active desktop. This packet is transmitted whenever the desktop number is changed.

### **M ADD WINDOW, and M CONFIGURE WINDOW**

These packets contain 24 values. The first 3 identify the window, and the next twelve identify the location and size, as described in the table below. Configure packets will be generated when the viewport on the current desktop changes, or when the size or location of the window is changed. The flags field is an bitwise OR of the flags defined in fvwm.h.

#### **Format for Add and Configure Window Packets and target\_struct structure:**

Byte	Significance
0	0xffffffff - Start of packet
1	packet type
2	length of packet, including header, expressed in long integers
3	ID of the application's top level window

4	ID of the Fvwm frame window
5	Pointer to the Fvwm database entry
6	X location of the window's frame
7	Y location of the window's frame
8	Width of the window's frame (pixels)
9	Height of the window's frame (pixels)
10	Desktop number
11	Windows flags field
12	Window Title Height (pixels)
13	Window Border Width (pixels)
14	Window Base Width (pixels)
15	Window Base Height (pixels)
16	Window Resize Width Increment(pixels)
17	Window Resize Height Increment (pixels)
18	Window Minimum Width (pixels)
19	Window Minimum Height (pixels)
20	Window Maximum Width Increment(pixels)
21	Window Maximum Height Increment (pixels)
22	Icon Label Window ID, or 0
23	Icon Pixmap Window ID, or 0
24	Window Gravity
25	Pixel value of the text colour
26	Pixel value of the window border colour

### **M LOWER WINDOW, M RAISE WINDOW, and M DESTROY**

These packets contain 3 values, all of the same size as an unsigned long. The first value is the ID of the affected application's top level window, the next is the ID of the Fvwm frame window, and the final value is the pointer to Fvwm's internal database entry for that window. Although the pointer itself is of no use to a module, it can be used as a reference number when referring to the window.

### **M FOCUS CHANGE**

These packets contain 5 values, all of the same size as an unsigned long. The first value is the ID of the affected application's (the application which now has the input

focus) top level window, the next is the ID of the Fvwm frame window, and the final value is the pointer to Fvwm's internal database entry for that window. Although the pointer itself is of no use to a module, it can be used as a reference number when referring to the window. The fourth and fifth values are the text focus colour's pixel value and the window border's focus colour's pixel value. In the event that the window which now has the focus is not a window which fvwm recognises, only the ID of the affected application's top level window is passed. Zeros are passed for the other values.

### **Format for Lower, Raise, and Focus Change Packets**

Byte	Significance
0	0xffffffff - Start of packet
1	packet type
2	length of packet, including header, expressed in long integers
3	ID of the application's top level window
4	ID of the Fvwm frame window
5	Pointer to the Fvwm database entry

### **M ICONIFY and M ICON LOCATION**

These packets contain 7 values. The first 3 are the usual identifiers, and the next four describe the location and size of the icon window, as described in the table. Note that M ICONIFY packets will be sent whenever a window is first iconified, or when the icon window is changed via the XA WM HINTS in a property notify event. An M ICON LOCATION packet will be sent when the icon is moved. In addition, if a window which has transients is iconified, then an M ICONIFY packet is sent for each transient window, with the x, y, width, and height fields set to 0. This packet will be sent even if the transients were already iconified. Note that no icons are actually generated for the transients in this case.

### **Format for Iconify and Icon Location Packets**

Byte	Significance
0	0xffffffff - Start of packet
1	packet type
2	length of packet, including header, expressed in long integers
3	ID of the application's top level window

---

<b>4</b>	ID of the Fvwm frame window
<b>5</b>	Pointer to the Fvwm database entry
<b>6</b>	X location of the icon's frame
<b>7</b>	Y location of the icon's frame
<b>8</b>	Width of the icon's frame
<b>9</b>	Height of the icon's frame

### **M DEICONIFY**

These packets contain 3 values, which are the usual window identifiers. The packet is sent when a window is de-iconified.

### **M MAP**

These packets contain 3 values, which are the usual window identifiers. The packets are sent when a window is mapped, if it is not being deiconified. This is useful to determine when a window is finally mapped, after being added.

### **M WINDOW NAME, M ICON NAME, M RES CLASS, M RES NAME**

These packets contain 3 values, which are the usual window identifiers, followed by a variable length character string. The packet size field in the header is expressed in units of unsigned longs, and the packet is zero-padded until it is the size of an unsigned long. The RES\_CLASS and RES\_NAME fields are fields in the XClass structure for the window. Icon and Window name packets will be sent upon window creation or whenever the name is changed. The RES\_CLASS and RES\_NAME packets are sent on window creation. All packets are sent in response to a Send\_WindowList request from a module.

### **M END WINDOWLIST**

These packets contain no values. This packet is sent to mark the end of transmission in response to a Send\_WindowList request. A module which request Send\_WindowList, and processes all packets received between the request and the M END WINDOWLIST will have a snapshot of the status of the desktop.

**M ERROR**

When fvwm has an error message to report, it is echoed to the modules in a packet of this type. This packet has 3 values, all zero, followed by a variable length string which contains the error message.

**M CONFIG INFO**

Fvwm records all configuration commands that it encounters which begins with the character \". When the built-in command \"Send ConfigInfo\" is invoked by a module, this entire list is transmitted to the module in packets (one line per packet) of this type. The packet consists of three zeros, followed by a variable length character string. In addition, the PixmapPath, IconPath, and ClickTime are sent to the module.

**M END CONFIG INFO**

After fvwm sends all of its M CONFIG INFO packets to a module, it sends a packet of this type to indicate the end of the configuration information. This packet contains no values.

## Appendix C. Tables of Words

resize	move
minimise	maximise
restore	destroy
delete	raise
lower	exit

**Table 1 Mode One Command Words**

left	right
up	down
cancel	proceed

**Table 2 Mode Two Command Words**

## Appendix D. Source Code Layout

/	- Source Toplevel Directory
Makefile	- Toplevel Makefile
mainprog.cc	- Main Program Loop, X and Non X.
prim.list	- Level One Words
sec.list	- Level Two Words
/trainer/	- Trainer Subdirectory
Makefile	- Trainer Makefile
train.cc	- Train All Words Base Class
train.h	- Train All Words Header File
tcltrain.cc	- Derived train for Tcl Trainer
tcltrain.h	- Header file for tcltrain.cc
traintcl.cc	- Main trainer program / C++/Tcl Interface
train.tcl	- Tcl Source Code
libtrain.a	- Library Archive for trainer subdirectory
/sb/	- Soundcard Subdirectory
Makefile	- Sound Makefile
soundcard.cc	- Sound Device Implementation
soundcard.h	- Sound Device header file
sword.cc	- Single Word Class Implementation
sword.h	- Single Word Class Header File
sndblk.h	- Sound Block structure header file
libsb.a	- Library Archive for sb directory
/dtw/	- Dynamic Time Warping Subdirectory
Makefile	- DTW Makefile
dtw.cc	- Dynamic Time Warper (dtw) Implementation
dtw.h	- Dynamic Time Warper (dtw) Header File
mlw.cc	- Most Likely Word (mlw) Implementation

---

mlw.h	- Most Likely Word (mlw) Header File
libdtw.a	- Library Archive for dtw directory
/fvwmmod/	- Fvwm Module Interface Subdirectory
Makefile	- Fvwm Module Makefile
module.h	- Module Interface Header File
module.cc	- Module Interface Implementation
translator.h	- Translator Header File
translator.cc	- Translator Implementation
xsword.h	- Derived Single Word for X Header File
xsword.cc	- Derived Single Word for X Implementation
lex.yy.cc	- Generated Lex Parser
texttocom.lex	- Original Lex Parser
modfvwm.h	- Module Include file from Fvwm distribution
libmod.a	- Library Archive for fvwmmod directory
/lpc/	- LPC subdirectory
Makefile	- LPC Makefile
lpc.h	- LPC Class Header File
lpc.cc	- LPC Class Implementation
module.h	- Module Class Header File
module.cc	- Module Class Implementation
liblpc.a	- Library Archive for LPC