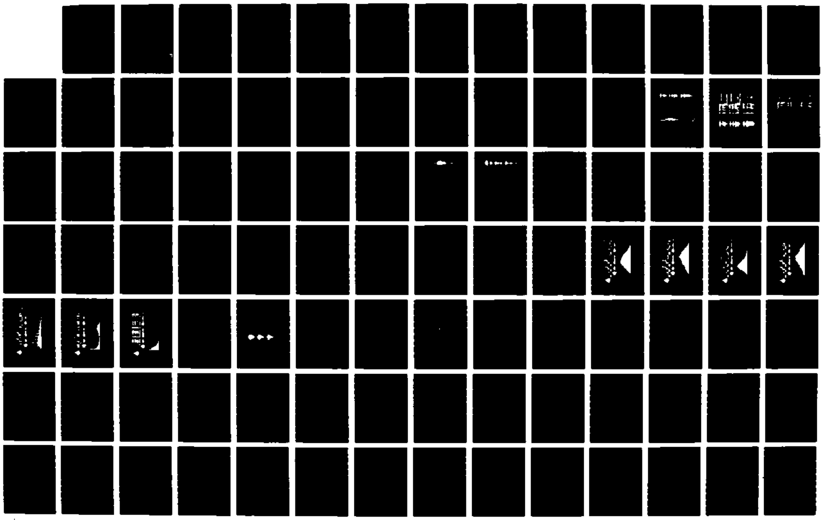
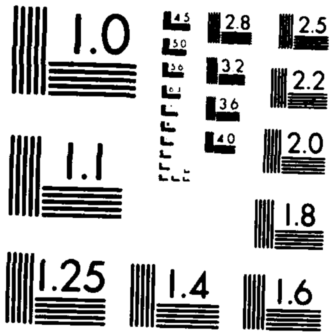
```
AD-A188 834   SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH        1/2
              RECOGNITION USING MIXED..(U) AIR FORCE INST OF TECH
              WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..  R G DAWSON
UNCLASSIFIED  DEC 87 AFIT/GE/ENG/87D-14                 F/G 12/9    NL
```




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

1

SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION USING MIXED FEATURE SETS

THESIS

Robert G. Dawson, Capt, USAF

AFIT/GE/ENG/87D-14

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

88 2 4 059

0

SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION USING MIXED FEATURE SETS

THESIS

Robert G. Dawson, Capt, USAF

AFIT/GE/ENG/87D-14

AFIT/GE/ENG/87D-14

SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION USING MIXED FEATURE SETS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Robert G. Dawson, Capt, USAF

December 1987

Acknowledgements

This work is dedicated to those I love; my parents, Lt Col and Mrs. William R. Dawson, whose example made this work possible; and my wife, Monica, whose love made it worthwhile.

Also, special thanks to my thesis advisor, Dr. Matthew Kabrisky, whose special combination of inspiration, knowledge, and humor made this work actually fun.

Table of Contents

List of Figures

Page

List of Tables

AFIT/GE/ENG/87D-14

Abstract

A system was developed to investigate continuous speech recognition. The system incorporates multiple features and dynamic programming to recognize continuous inputs of the spoken digits (zero through nine). The fundame.tal design concept extends from previous successful recognition research efforts involving both isolated and continuous speech using multiple feature sets, multiple template sets, and dynamic programming. Among the features used in the investigation are wide band spectrogram, narrow band spectrogram, linear predictive coding (LPC) coefficients, LPC spectrum, frication frequency, and formant tracks. An advanced speech research tool called SPIRE provided the computational functions needed to extract the raw features. The system is implemented in LISP on a Symbolics 3600 series LISP machine.

SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION USING MIXED FEATURE SETS

I. Introduction

The ability to communicate using language is considered one of the hallmarks of the human race. As machines attempt to do more and more of what humans do, it becomes necessary for them to also have the ability to use language. And before this can be acheived, machines must be able to accurately identify spoken words without the aid of syntax or semantics. Even simple word recognition by computer would have many potential applications, from voice controlled television sets to voice activated displays in the cockpit of an F-16 fighter aircraft. Ultimately, speech recognition is seen as essential to the total automation of the human-machine interface, including automated dictation, language translation, and artificial intelligence devices. Although word recognition has improved steadily over the past decade, general speech recognition devices still do not exist. Therefore, much research will be necessary before such general speech recognition devices can be perfected.

Background

Speech recognition is a relatively new field of study made possible only by recent advances in computer and digital signal processing technology. Today, a variety of speech recognition systems are commercially available ranging from expensive stand alone units to plug-in boards for personal computers. However, many practical problems still exist. There are many problems yet to be solved if speech recognizing devices are to find their way into common use. Most current research is focused upon finding better ways to represent speech, and once represented, better ways of handling the variations of speech patterns typical of a diverse population. Only if these problems areas are solved will the more complex problems of connected speech be solved.

Although speech recognizers have improved steadily over the past few years, they are still hampered by certain serious performance problems. The most intractable of these is inaccuracy. There are many systems available claiming as high as 95% accuracy, meaning the system can correctly identify spoken words 95% of the time. However, in real environments these claims are more hopeful than true. (14:200)

Current voice recognizers suffer from many operational deficiencies resulting from inaccuracy. Almost all systems available today require extensive user training. The user must train the system to recognize his voice and, therefore, the system is speaker-dependent. There always seems to be a certain percentage of the population for whom the system performs very poorly. Some machines work better with male voices than with female voices or vice versa. Current speech recognizers perform poorly in the presence of background noise such as office or factory noise. Further, most current speech recognizers can only recognize purposely isolated words rather than more natural continuous speech. Finally, current speech recognizers lack large enough vocabularies to be useful for many applications. (14:200)

Specific problem areas in speech recognition include selection of good feature sets and template sets, the problem of connected speech, and the problem of speaker-independent speech recognition. These terms will be discussed fully in the following paragraphs.

Definitions

Templates and Features. A template is a set of data that represents each word of the given vocabulary. The fundamental task of speech recognition is matching the spoken word, called an utterance, to a set of stored templates and deciding which template the utterance represents. There is typically a template for each word in the vocabulary. Templates are created through a process known as "training" in which the user repeats the vocabulary into the recognizer. Features are then extracted and stored as templates (12:489). Herein lies the basic challenge of speaker-independent speech recognition, that is "creating a set of templates that can be used reliably with many different speakers" (6).

A template set that can be used reliably with many different speakers must contain those features that best represent each word of the vocabulary. Exactly what features carry the essential information that distinguishes a word from the rest is not completely clear. Brusueles (2) investigated this problem by extracting 55 different features, grouped in six general categories, for each word in a 13 word vocabulary. The six general categories were:

(1) Wide-band Spectrogram. Graphic depiction of frequency content.

(2) Zero Crossing Rate. A count of the number of times the waveform passes through a region centered around zero.

(3) LPC gain and

(4) LPC coefficients. Coefficients and gain terms used in a speech coding technique called Linear Predictive Coding (LPC) in which the speech production process is modeled rather than the waveform itself. This is done by an adaptive filtering process in which the filter coefficients are calculated so as to simulate the vocal tract which itself is a filter. These coefficients are commonly referred to as linear predictive coding (LPC) coefficients. These LPC coefficients can be used to reproduce a copy of the original speech waveform. Doing so allows voice to be digitally transmitted at a very low bit rate and thus a very small bandwidth or storage capacity. Because of this data reduction property, LPC coefficients are often used for speech recognition (4:26).

(5) Formants. Resonant frequencies of the vocal tract.

(6) Time. Time over which utterance is spoken.

While certain of these 55 different features proved useful during word-template matching, others were less so. Further, their usefulness changed with different speakers and/or words. Brusueles suggests that it may be possible to use some sort of statistical weighting of features to improve the word-to-template matching algorithms (2:68-69).

"Vector quantization" is a another coding technique used in speech recognition. In this method, the different sounds that are produced by the vocal tract are represented by individual numbered codes. Then words are represented by vectors made up of these codes. For example, the word "Bill" could be represented by the vector <4, 32, 32, 32, 32, 20, 20>, where 4, 32, and 20 represent the sounds "b", "i", and "l" respectively. In their work Burton, Shore, and Buck applied "vector quantization" and achieved a recognition accuracy of 98% for speaker-independent recognition of isolated digits, "zero" through "nine" (3:837). Vector quantization is a relatively new speech encoding technique.

Speech can be also modeled as a Markov chain in which the current signal state is somewhat dependent on the previous signal state. For example, vowel sounds are more likely to follow consonant sounds. Signal modeling based on "hidden Markov models" (HMM) may be viewed as "a technique that extends conventional stationary spectral analysis principles to the analysis of time varying signals." Juang and Rabiner investigated two types of Markov models. One was based on finite mixtures of Gaussian autoregressive densities (GAM), and the other was based on nearest-neighbor partitioned finite mixtures of Gaussian autoregressive densities (PGAM). Juang and Rabiner determined that GAM and PGAM models have applicability to speaker independent digit recognizers (5:1404,1412).

A fundamental goal in speaker-independent speech recognition is that of creating a set of reference data or templates that can be used reliably with many different speakers. One way that has been used is that of template averaging. This method is accomplished by recording as many as 10 tokens (examples) of each word in the particular vocabulary and averaging them into one template set (4:32). Brusueles found that a template made by averaging two males and one female performed better than a template made by averaging three males in a test population of seven males and three females suggesting that a wide variation of reference patterns may be more effective than a narrow variation (2:28).

Multiple template sets have also been used to support speaker-independent speech recognition (14:29). In this case the tokens are stored individually and compared individually as though they were separate words in the vocabulary. One technique involved "clustering" 100 repetitions of each word in a 39 word vocabulary. The vocabulary consisted of the 26 letters of the alphabet, the 10 digits, and three command words (STOP, ERROR, and REPEAT). Average recognition accuracies of close to 97 percent were obtained on 38 of the 40 talkers (13:583). This method has the advantage of being able to represent a wider range of population, however an obvious disadvantage is the increased computation required to perform necessarily more comparisons than for a vocabulary represented by single templates. (10:263).

Dynamic Programming. Regardless of the method chosen to accomplish word-to-template matching, it is usually necessary to establish optimum time alignment between the input and reference speech data.

> Originally, advanced research in speech recognition employed relatively simple techniques to partition a speech signal into separate units, then very complex methods to classify the segments and recover from segmentation errors. It was soon realized that signals could not be reliably segmented without prior knowledge of the acoustic sound class. In the early 1970's a technique called "dynamic programming" was

> introduced. Dynamic programming improves the segmentation process by hypothesizing acoustic events and testing each hypothesis at an acoustic level (4:26).

A one-stage dynamic programming algorithm for connected speech has been proposed by Ney (10). This algorithm was actually proposed as far back as 1971 by T. K. Vintsyuk. Ney states that,

> An advantage is that the three operations of word boundary detection, nonlinear time alignment, and recognition are performed simultaneously: thus, recognition errors due to errors in word boundary detection or to time alignment errors are not possible.

Dynamic programming is considered one possible springboard for future advances in speech recognition (4:26).

Connected Speech. Connected speech, as opposed to isolated speech, presents additional unsolved problems. Although some speech recognizers can, to a limited extent, recognize words without pauses between the words, they are less accurate and more expensive. Connected speech is difficult for a number of reasons. One problem is that of detecting word boundaries. Although some techniques don't require word boundary detection, these techniques pay a penalty in terms of much more intense data comparison requirements. The real difficulty of connected speech recognition stems from the fact that acoustic variation of words spoken in connected speech is much greater than when the words are spoken in isolation. This is due to the "coarticulation" of neighboring sounds. The position of the tongue, jaw, and lips in one speech sound are affected by their previous and future positions. Further, the time variations of words is more severe for continuous speech than for isolated speech (4:27).

Speaker-Independence. It is obvious that humans can recognize the speech of a variety of speakers without the need of any training process. Somehow, the brain is able to extract the key features of speech, determining what is being said even though different speakers may say the same thing differently. Current word recognition systems simply lack this "robustness" that is necessary for most applications. One approach to improving "robustness" and thus accuracy lies in developing systems that are speaker-independent (6).

Problem

The primary purpose of this thesis was to develop a system for connected speech recognition and examine the usefulness of using multiple templates, multiple features, and dynamic programming. The system has been implemented on a Symbolics 3600 Series Lisp Machine using an advanced speech analysis tool called SPIRE (Speech Processing Interactive Research Environment).

Scope

The recognition system developed is based on recognition of continuously spoken digits, zero through nine. The small vocabulary was necessary due to limited time and disk space, however, recognition of the digits provides a sufficient challenge for the purpose of this research. This research has investigated what features are best suited for speech recognition and how best combine them. The dynamic programming algorithm used is identical to the one-stage dynamic programming algorithm proposed by Ney (10). Other programming techniques have not been directly addressed. For the most part, SPIRE is used as a library of functions called by the main LISP program, although SPIRE can be used in an interactive mode as well.

Approach

The approach is outlined as follows. First, individual feature performance will be observed by plotting "distance array contours". Next, a continuous speech dynamic time warping algorithm will be used to further study features. Finally, speaker-independent continuous speech will be studied.

Sequence of Presentation

Chapter two presents the acoustic processing environment. In particular, the chapter introduces SPIRE, a powerful speech analysis research tool, as well as the Symbolics 3600 Series Lisp Machine.

Chapter three defines the system design including the basic algorithms developed. A basic explanation of the dynamic time warping algorythm is included here.

Chapter four presents the results. First, individual feature performance is investigated to see which feature sets are best suited to speech recognition and optimal ways of combining multiple feature sets to increase performance. The best of these are then tested for speaker-independent performance. Last, multiple template sets are used in an effort to improve speaker-independent performance.

Chapter six provides conclusions and recommendations, and appendices present additional results, program description, and listings.

II. Acoustic Processing Environment

Introduction

The purpose of this chapter is to introduce the software and hardware components used to develop the recognition system. The chapter is divided into three sections. The first section describes the Lisp programming language. The next section describes SPIRE, an advanced speech analysis tool that provides many of the computational functions used for feature extraction and general speech processing. The last section describes the hardware configuration that is used as well as other optional hardware.

Lisp

Lisp is a high level programming that takes its name from List Programming. Lisp, one of the oldest active programming languages, is widely used in the field of artificial intelligence (11). Lisp is an extremely powerful language for handling large amounts of data common in artificial intelligence applications. In fact special purpose computers called Lisp Machines are designed at the circuit level especially for running Lisp. Together, these provide a powerful computing environment with a large virtual address space that make it "particularly attractive for speech and signal processing applications" (2:5).

There are many dialects of Lisp, however, one dialect called COMMON Lisp seems to be emerging as a standard. Most Lisp machines now in production have Common Lisp as a standard feature. Older machines may use different dialects. The Symbolics 3600 Series Lisp Machine used for this work uses a dialect called Zeta Lisp.

SPIRE

Overview (2:6-12). SPIRE stands for Speech and Phonetics Interactive Research Environment. [SPIRE is available by license through the MIT Patent Office.] It is a software program that allows the user to interactively examine and process speech and other audio signals. The following paragraphs provide an overview of SPIRE's design philosophy, graphical capabilities, implementation considerations, and documentation.

SPIRE was designed to be easy enough to use for tha novice, yet powerful enough for even the most advanced users. In the interactive mode, SPIRE takes full advantage of the Lisp Machine's built in graphical interface for quick and easy research. For the more advanced user, SPIRE allows relatively painless customization and modification. The interactive mode is very useful for learning about the various attributes of speech. The next paragraph describes some of SPIRE's more common capabilities. For more detailed information concerning the use of SPIRE, the reader is referred to various SPIRE documentation (7), (9), (15), and (16).

SPIRE takes full advantage of the graphical capabilities of the Symbolics Lisp Machine, providing bit-mapped display which is either 1280 pixels wide by 760 pixels high or 1216 pixels wide by 773 pixels high, depending on the model. The following figures illustrate a small sample of these capabilities for the utterance, "This is the CBS Evening News."

<u>Figure 1.1</u> Figure 1.1 shows two repetitions of the orthographic transcription and original waveform of the utterance. Note that the scale of the two displays are different to allow closer examination of waveform details.

<u>Figure 1.2</u> The second figure shows four displays of the same utterance; Orthographic Transcription, Wide-Band Spectrogram, Formants, and Original Waveform. Two of the displays are overlaid--the Wide-Band Spectrogram and the Formants. Such overlays can make it easier to track similarities among various representations of the data.

<u>Figure 1.3</u> The third figure illustrates another important feature of SPIRE: its ability to synchronize displays. For example, in the top display, there is a "cursor" located at 1.8251 seconds of the Original Waveform. The curser is automatical place at the same point in the next display, the Narrow-Band Spectrogrm. The next display shows the Narrow-Band Spectral Slice at that cursor position.

<u>Figure 1.4</u> The fourth figure identifies typical display types available from SPIRE.

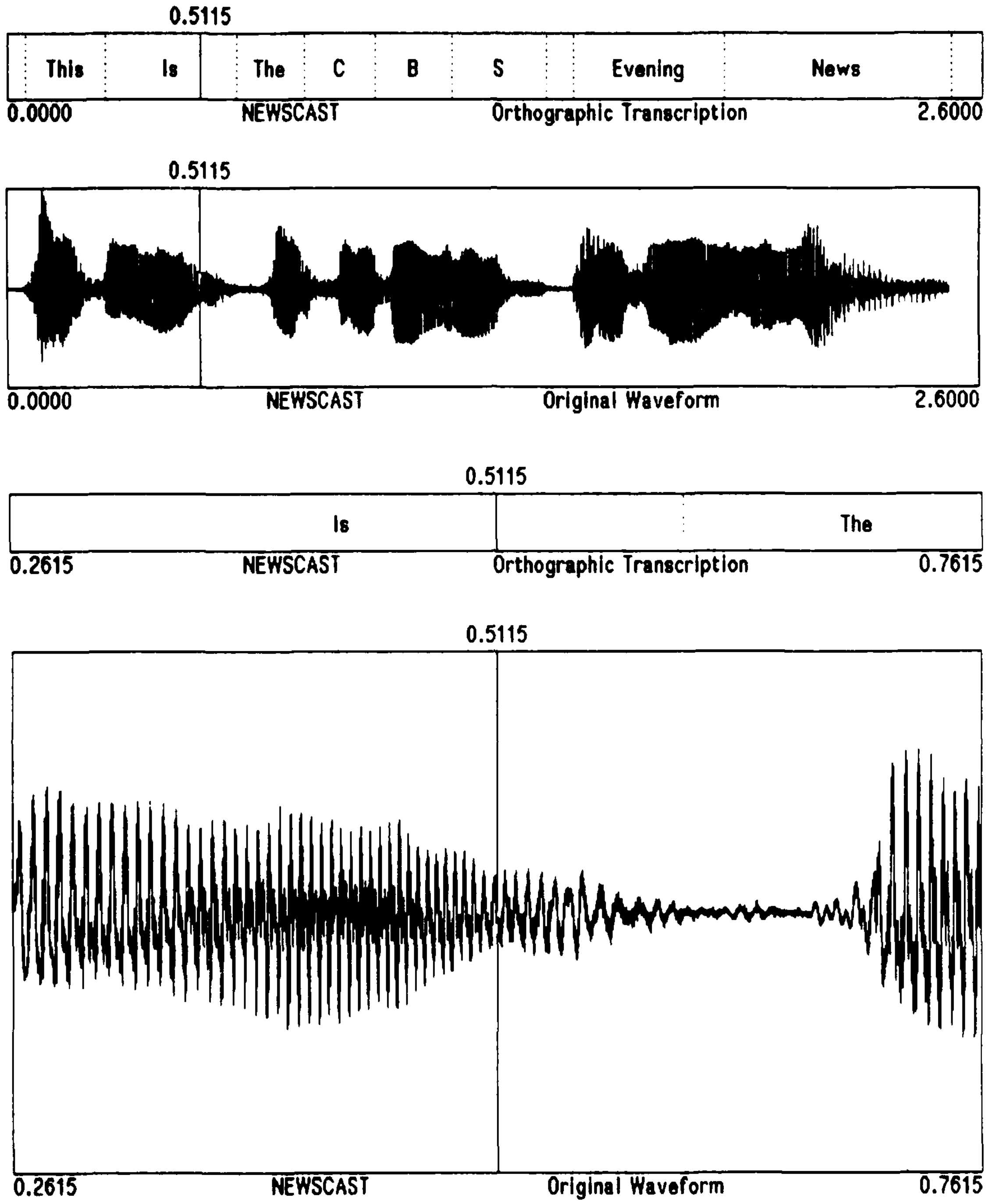


Figure 1.1 Original Waveform

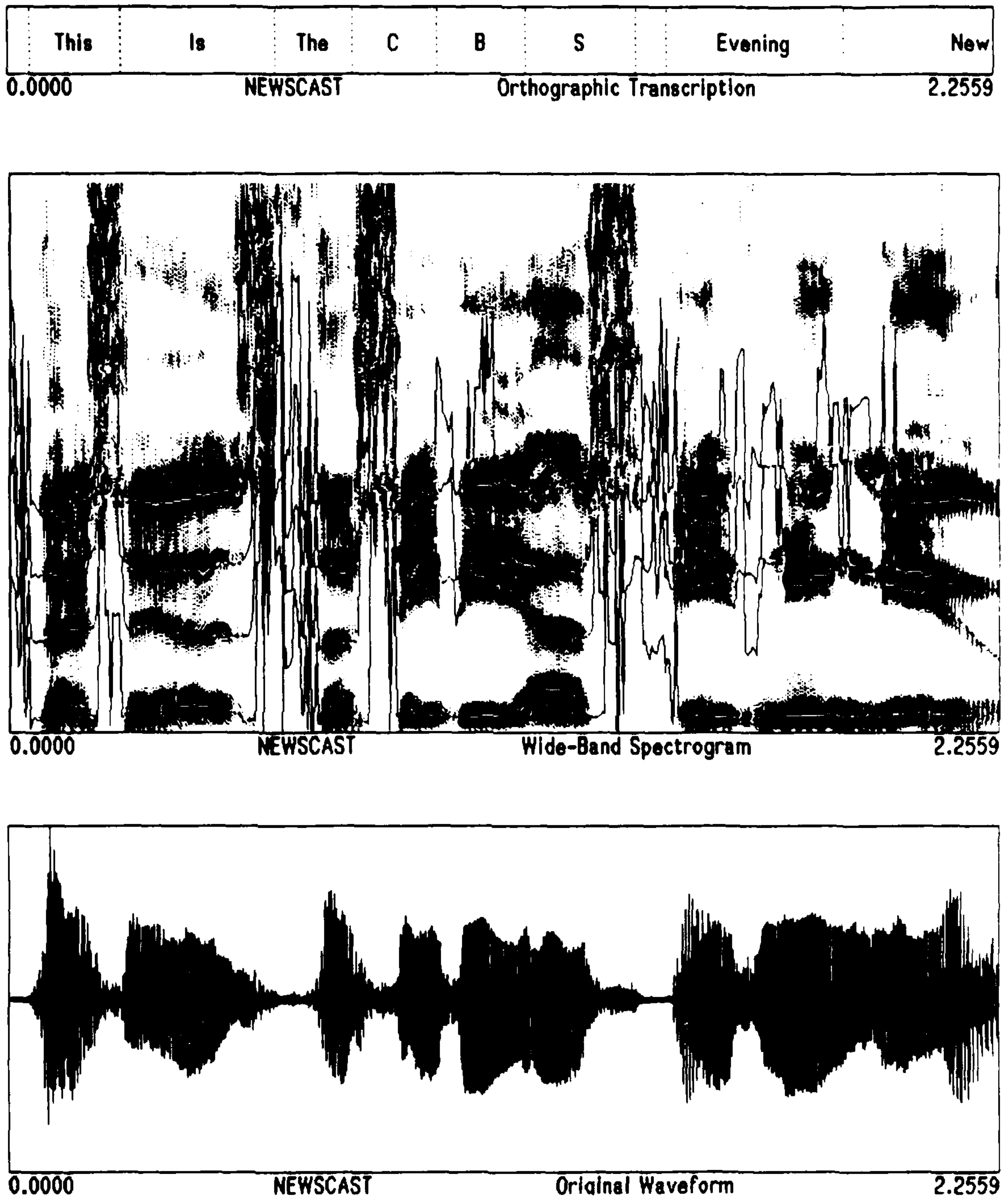


Figure 1.2 Overlaid Displays

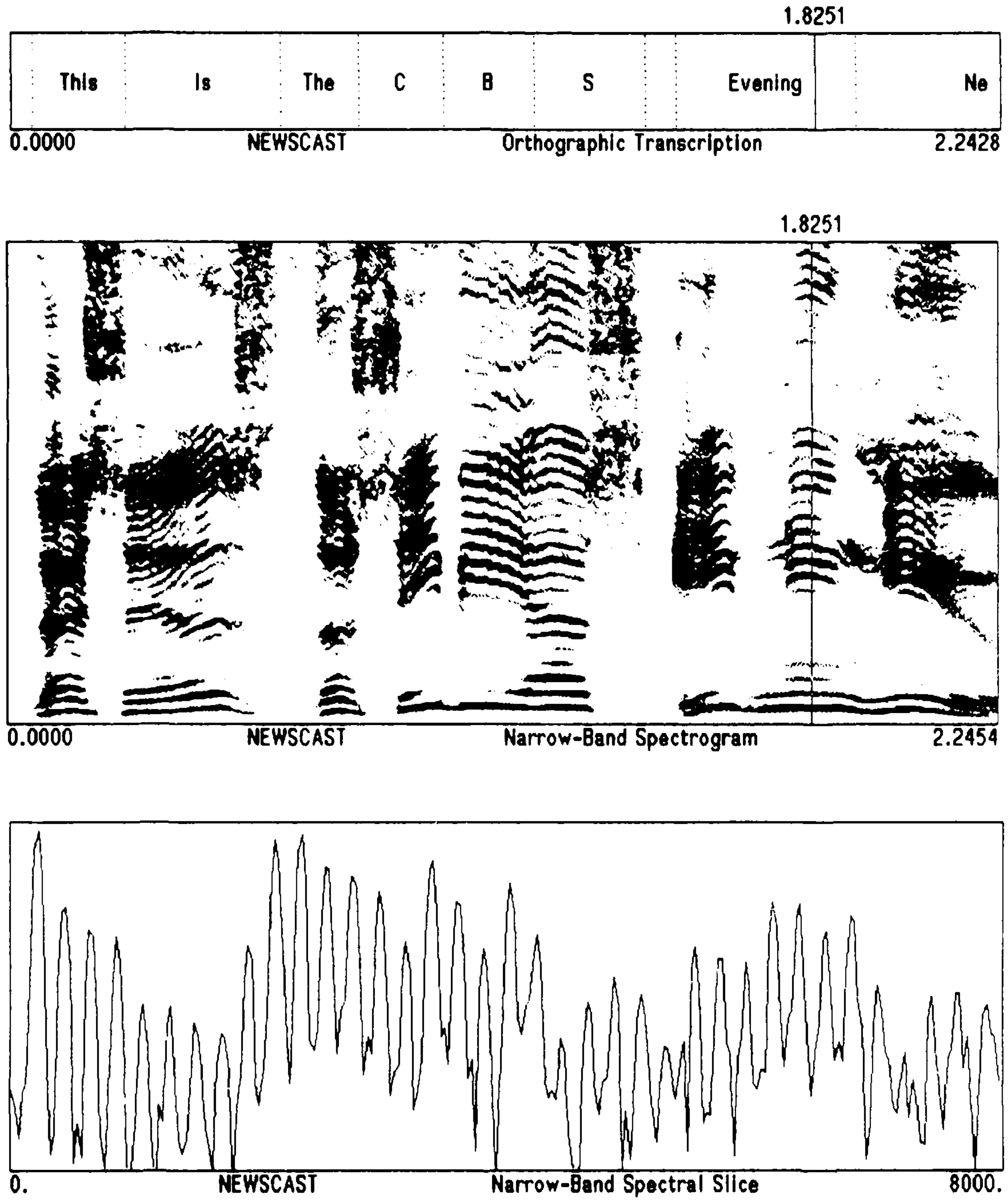


Figure 1.3 Synchronized Displays

List of SPIRE Displays

Energy, Total
Energy, 0 to 5000 Hz
Energy, 120 to 440 Hz
Energy, 3400 to 5000 Hz
Energy, 640 to 2800 Hz
Formants, All Four
Formant, First
Formant, Second
Formant, Third
Formant, Fourth
Frication Frequency
LPC Center of Gravity
LPC Gain Term
LPC Predictor Coefficients
LPC Spectrum Slice
Narrow-Band Spectrogram
Narrow-Band Spectral Slice
Narrow-Band Spectrum Slice
Original Waveform
Orthographic Transcription
Phonetic Transcription
Pitch Frequency
Waveform Envelope
Wide-Band Spectrogram
Wide-Band Spectral Slice
Wide-Band Spectrum Slice
Zero Crossing Rate

Figure 1.4 Standard SPIRE Displays

Interfacing SPIRE from Lisp. Behind each SPIRE display are the underlying computations required for computing that display, for example a Fourier transform. These underlying processes are available through Lisp as simple function calls. Figure 1.5 describes the primary functions used to make SPIRE perform computations on an utterance.

The three functions of figure 1.5 can be combined into a single Lisp expression. For example,

```
(SETQ
   RESULT-ARRAY
   (SPIRE:ATT-VAL (SEND (SPIRE:UTTERANCE PATHNAME)
                        :FIND-ATT ATT-NAME)))
```

where **RESULT-ARRAY** is the variable containing the result of the computation defined by the variable **ATT-NAME** performed on the utterance defined by the variable **PATHNAME**.

When no more computations are necessary on a particular utterance, the utterance may be "killed" or "unloaded" as follows:

```
(SEND (SPIRE:UTTERANCE PATHNAME) :KILL)
```

where the variable **PATHNAME** describes the utterance to be killed.

Note that the method used here does not alter any of SPIRE's default attributes. Appendix A shows a list of SPIRE's attribute defaults.

When SPIRE is called to perform a computation on an utterance, the result is returned in the form of an array, the dimensions of which depend on the type of computation. Figure 1.6 lists the array types returned for various SPIRE function calls.

SPIRE:UTTERANCE

Parameters:	pathname (required)
Type:	function
Returns:	utterance-flavor
Description:	The utterance in the file "pathname" becomes the current utterance in SPIRE. If needed the utterance is loaded into memory from disk. This function must be called before any computation can take place.

:FIND-ATT

Parameters:	att-name (required)
Type:	message to utterance flavor
Returns:	att
Description:	att-name is a string that identifies what attribute the user desires SPIRE to compute. For example, assume we are to compute the Wide-Band Spectrum of an utterance stored in the file "**>DAWSON>UTTS>ZERO.UTT**". First, select the utterance:

```
(SETQ TEMP1
      (SPIRE:UTTERANCE ">DAWSON>UTTS>ZERO.UTT"))
```

TEMP1 stores the utterance flavor for the next step:

```
(SETQ TEMP2
      (SEND TEMP1 : FIND-ATT "WIDE-BAND SPECTRUM"))
```

TEMP2 now holds the att from which the actual values may be extracted (see next function).

SPIRE:ATT-VAL

Parameters:	att (required)
Type:	function
Returns:	array (results of computation)
Description:	This function returns the computed value of the att we are interested in. For example, if **TEMP2** holds the att (as discussed above), extract the values:

```
(SETQ TEMP3 (SPIRE:ATT-VAL TEMP2))
```

TEMP3 now holds the "Wide-Band Spectrum" values. Similar procedures are followed for obtaining the values of any of the standard SPIRE computations.

Figure 1.5 SPIRE Interface Functions

Attribute Name	**Result Array**
Wide-Band Spectrum	2-D, 256 X N
Narrow-Band Spectrum	2-D, 256 X N
LPC Spectrum	2-D, 256 X N
LPC Coefficients	2-D, 19 X N
Formants (four)	2-D, 5 X N
LPC Gain Term	1-D, N
LPC Center of Gravity	1-D, N
Zero Crossing Rate	1-D, N
Frication Frequency	1-D, N
Total Energy	1-D, N

N = time * analysis rate

Figure 1.6 SPIRE Result Arrays

Hardware (13,15:16).

SPIRE is a software package that requires specific hardware to run. A brief description of hardware options are discussed below.

Lisp Machine. (Required) SPIRE is designed to run on a Symbolics 3600 Series Lisp Machine. The Symbolics Lisp Machine is a powerful computer specifically designed to efficiently run Lisp code. It provides an extremely efficient user interface with extensive graphics capabilities. Also available from Symbolics is a Floating Point Accelerator (FPA) card designed to speed up floating point operations by about a factor of three. The FPA is an add-on card that is generally invisible to the application software such as SPIRE.

Array Processor. Certain versions of SPIRE are designed to support a Floating Point Systems FPS 100 (or FPS 200) array processor. An array processor is a special purpose device designed to quickly handle computations on large arrays of data. The FPS 100 is connected to the

Symbolics Lisp machine through a UNIBUS interface. The array processor can drastically reduce the computation time required for certain SPIRE functions. An approximate comparison between a "bare" Symbolics Lisp Machine, one with an FPA, and one with an FPS 100.

Configuration	Ratio	Example
FPS Array Processor	10	1.0 minutes
Floating Point Accelerator	3	3.0 minutes
Bare Lisp Machine	1	10.0 minutes

Speech digitizer. SPIRE is designed to operate with a Digital Sound Corporation (DSC) analog-to-digital converter. The DSC is connected to the Symbolics Lisp machine via the UNIBUS interface. The DSC is used primarily to digitize speech and other audio signals. The audio input can be direct or prerecorded and fed through line-in jacks. The DSC can also be used for high quality playback of the digitized signals.

Summary of Equipment

The Symbolics Lisp Machine actually used for this speech recognition research was an older model Symbolics 3600 with one mega-word of RAM operating under version 6.0 of the operating system. The Lisp Machine was equipped with a Floating Point Accelerator to reduce computation time. An FPS 100 array processor was not connected. Speech samples were digitized on a using a noise reducing microphone fed directly into a DSC A/D converter. Version 17.2 of SPIRE was used.

III. System Design

Introduction

The purpose of this chapter is to describe the system design. This chapter will provide details about the major processing functions and how they are used. It will also describe generally how major groups of data are handled. Finally, an description of the dynamic programming algorithm used is given.

Utterance Processing

As mentioned earlier, the continuous speech recognition system is designed around ZetaLisp and SPIRE. SPIRE is used as a function library that is called by the main Lisp routines. A discussion of how this is done is given in chapter two. Processing of an utterance consists of specific computations done by SPIRE on the original digitized waveform, plus any additional processing done by the main Lisp routines. Several Lisp functions are defined for this purpose depending on the desired features. (See Appendix B).

Feature Extraction. Feature extraction consists of calling SPIRE, with the filename of the utterance and the name of the feature, to perform the necessary computations and thus return the desired feature. This is done by a function called COMPUTE-ATT. (See Appendix B). A discussion of methods used by SPIRE to compute the desired features follows.

Wide-Band and Narrow-Band Spectrum. Spectrum calculations are returned by SPIRE as two dimensional arrays, 256 X N, where N is proportional to the length of the utterance. In both cases, the wide-band spectrum and the narrow-band spectrum, the original waveform is pre-emphasized and then run through a 256 point Fast Fourier Transform (FFT) routine incorporating a Hamming window. The wide-band spectrum is calculated using a filter bandwidth of 300.0 Hz, while the narrow-band spectrum uses a filter bandwidth of 78.0 Hz. Accordingly, the narrow-band spectrum provides more frequency resolution than does the wide-band spectrum. The results are returned in 256 discrete frequency components representing 0 to 8000 Hz in log-magnitude form. Figure 3.1 shows an example of wide-band and narrow-band spectral slices.

<u>LPC Spectrum.</u> The LPC spectrum result is similar to wide-band spectrum above, except the LPC coefficients are used to calculated the spectrum. The LPC spectrum generally resembles a smoothed version of the wide-band spectrum. Figure 3.1 shows an example of LPC spectrum slice as well as wide-band and narrow-band spectral slices.

<u>Formants.</u> Formants are returned by SPIRE as a two dimensional array, 5 X N, where N is proportional to the length of the utterance. Rows one through four of this array represent the first four formant frequencies, respectively. Row zero is not used. Formant values are computed from the LPC Spectrum. The formant peaks are found by fitting a polynomial to each LPC spectral slice. The polynomial is then differentiated and solved for zeros. Formant tracts are somewhat erratic. The formant tracking algorithm usually loses track during fricative sounds. Figure 3.2 shows an example of formants along with original waveform and frication frequency.

<u>Frication Frequency.</u> Frication frequency is returned as a one dimensional array of length N, where N is proportional to the length of

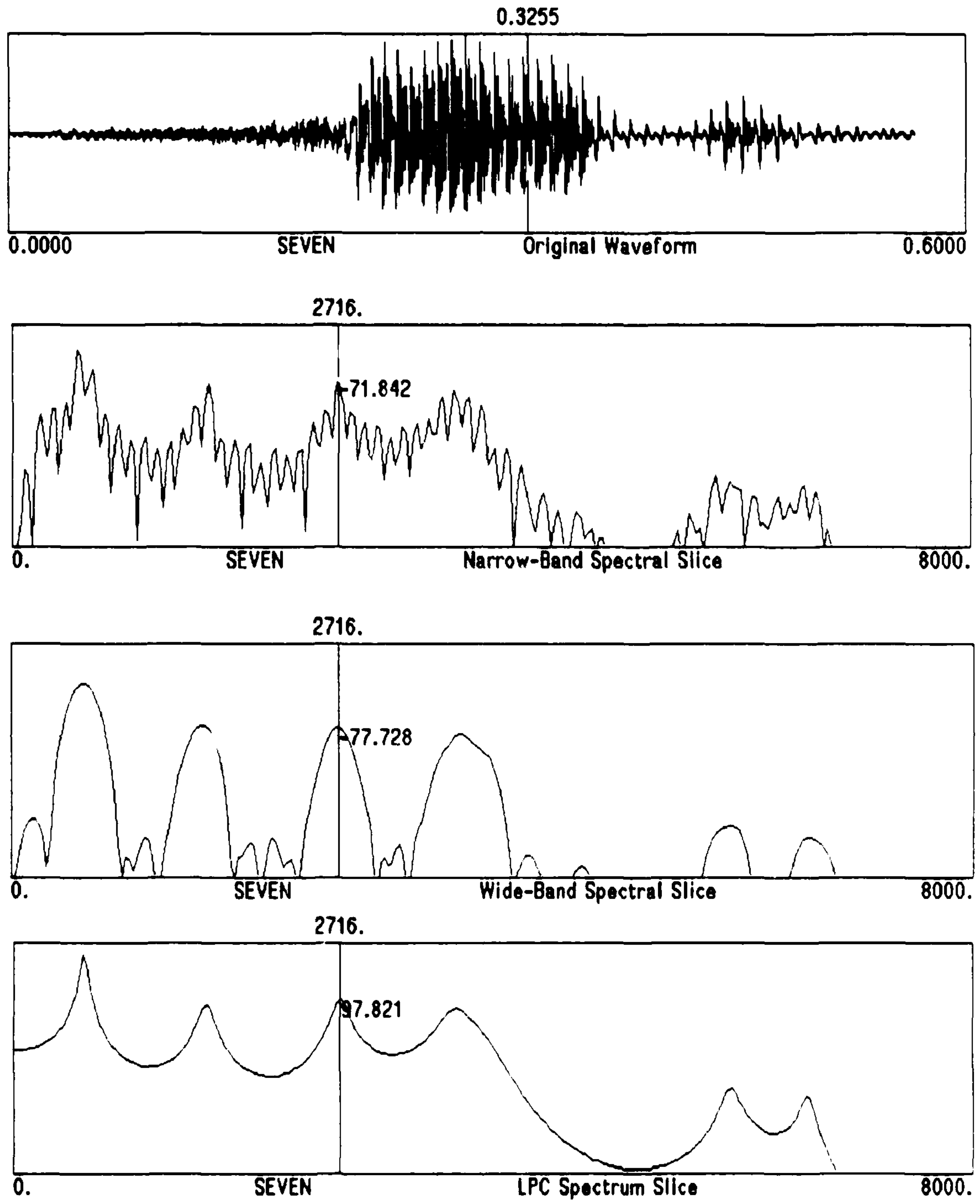


Figure 3.1 Spectral Slices

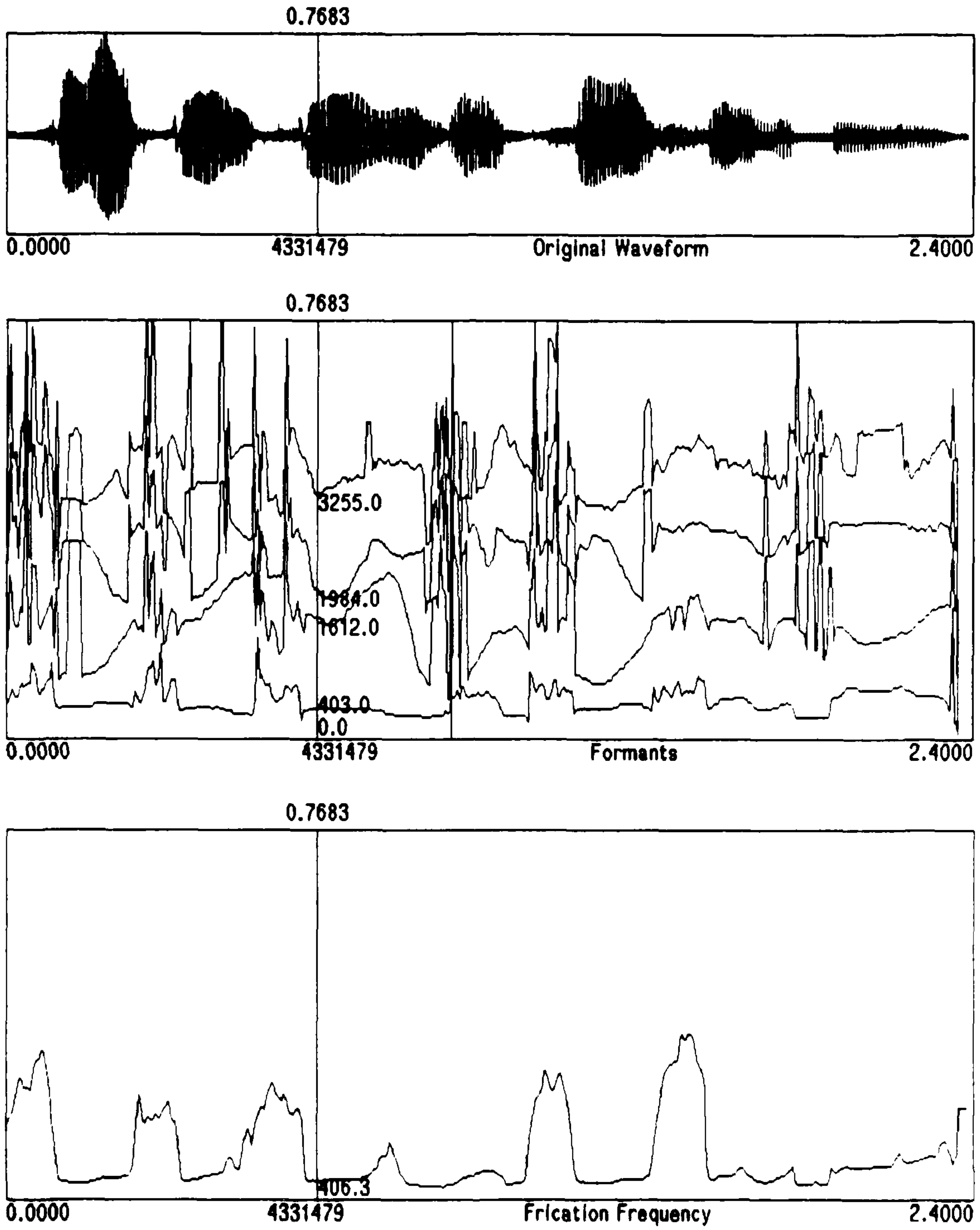


Figure 3.2 Formants, Frication Frequency

the utterance. It attempts to track the frequency of fricative sounds in an utterance. During non-fricative sounds the value is below 500, and during fricative sounds the values are above 1000. Frication frequency is fair indicator of whether a fricative or vowel sound is occurring. Figure 3.2 shows an example of Frication Frequency.

Additional Processing. It is necessary to perform additional processing on SPIRE results. This additional processing is discussed below.

Clipping. The last five time slices of all SPIRE results are clipped or ignored. Due to the predictive nature of the LPC coefficients computations, the last five time slices can't be calculated and are returned by SPIRE as zero values. As a result of this, any feature which is built upon LPC coefficients, such as formants, also has zero values in the last five time slices. To maintain uniformity, that is, so that any feature extracted from an given utterance will have the same meaningful length, the last five time slices are ignored for all SPIRE results.

Median Filtering. Due to the erratic nature of the formant tracks, these results are further processed through a median filter. The median filter filters out unwanted spikes in the formant tracks. (See Appendix B, for the Lisp function MEDIAN-FILTER).

Frequency Compression. To reduce computation requirements, wide-band, narrow-band, and LPC spectrum results are compressed from 256 discrete frequency components down to 16. Further, this compression is done so as to emphasize resolution in the lower frequencies and de-emphasize resolution of the higher frequencies. Briefly, the lower 132 frequency components (0 to 4,125 Hz) are linearly compressed down to 12 components, and the upper 124 components (4,125 to 8,000 Hz) are linearly compressed down to 4 components. It should be noted here that since the speech waveforms are pre-emphisized by SPIRE before performing spectrum calculations, frequency components are averaged instead of added. (See Appendix B, for Lisp function FREQUENCY-COMPRESS-LFE).

Energy Normalization. Energy normalization is performed on each time slice of wide-band, narrow-band, and LPC spectrum. This is done so that energy disparities won't effect the word recognition process. (See Appendix B, for Lisp function ENERGY-NORMALIZE).

Ready-Utterance. A ready-utterance is simply a name used to represent the set of data which is the result of all the processing done on a given utterance. Once computed, a ready-utterances is stored to disk so that it may be used over and over with out having to re-compute all its features. A ready-utterance takes the Lisp form of a list of arrays, where each array corresponds to a processed SPIRE result.

Ready-Template. A ready-template is simply a name used to represent the processed version of the entire recognition vocabulary. Each utterance of the recognition vocabulary is processed into individual ready-utterances and combined into one large list of ready-utterances. Again, this is so that re-computation is reduced.

Dynamic Time Warping Algorithm(10)

Introduction. Dynamic time warping or dynamic programming is a method by which speech patterns are nonlinearly time aligned. This time alignment is necessary due to the nonlinear time variations common in speech. Dynamic time warping was invented by T. K. Vintsyuk. The algorithm used for this system is one adapted for continuous speech. It was originally presented by Vintsyuk and later translated by Hermann Ney (10:263). (See Appendix B, for the Lisp function SCAN-DTW).

Distance Arrays. A distance array is basically a two dimensional array, **M** by **N**, where **M** is proportional to the length of the template, and **N** is proportional to the length of the utterance or test pattern. (Preliminarily, assume isolated speech.) Both the template and utterance are represented by a sequence of **M** and **N** vectors respectively. Each vector represents the features of both the template and the utterance extracted at each moment **m** and n respectively. Each value of subscript (**m**, n) of the distance array then represents the vector distance between the template at moment **m** and the utterance at moment **n**.

Distance arrays are a key element in the word recognition process. For isolated speech, a measure of utterance-template similarity is taken by tracing the path from point (0, 0) to point (M, N) of the distance array that results in the smallest accumulated distance of all the points in that path.

Figure 3.3 shows a simplified example of a distance array using a hypothetical feature set consisting of energy in three frequency bands. For example, at any particular moment, the speech is represented by a 3-dimensional vector representing the energy in each of the three frequency bands. For isolated speech, the correct word would be identified by calculating a distance array between the test word and each word in the recognition vocabulary and then choosing the the template that results in the lowest accumulated distance. The distance rule used is Minkowski 1 distance, also known as the taxi distance. For

Template Pattern — SIX				
(0,2,5)	1	13	7	**1**
(0,2,5)	1	13	7	**1**
(0,0,0)	8	10	**2**	8
(5,4,0)	11	**1**	9	13
(0,2,5)	**1**	13	7	1
(0,2,5)	**1**	13	7	1
	(0,3,5)	(5,5,0)	(1,0,1)	(0,2,6)
	Test Pattern — SIX			

Figure 3.3 Hypothetical Distance Array

Template Pattern — SIX				
(0,2,5)	25	37	19	**7**
(0,2,5)	24	26	12	6
(2,0,0)	23	13	5	13
(5,4,0)	13	3	12	25
(0,2,5)	2	14	21	22
(0,2,5)	1	14	21	22
	(0,3,5)	(5,5,0)	(0,0,1)	(0,2,6)
	Test Pattern — SIX			

Figure 3.4 Hypothetical Accumulated Distance Array

example the distance between the vectors <0,2,5> and <5,5,0> would be 5 + 3 + 5 = 13. In order to find the minimum path through the connected speech distance array, a new "accumulated distance array" is constructed, shown in figure 3.4. In this array the value of each point represents an accumulated distance that is equal to the local distance of that point plus the minimum of the accumulated distances of all possible preceding points. Notice the problem for isolated speech is simplified by the fact the begin and end points are known. Also notice that certain constraints govern the route of the traced path. The path must continue forward in time for both the template and the test pattern. Therefore the path cannot go left or down in direction, and points may not be skipped or hopped over.

One-Stage Algorithm for Connected Speech. What follows is a brief summary of the algorithm given by Ney [10], which the reader should consult for further details. The algorithm is summarized as follows. A composite distance array of grid points (i,j,k) is computed as shown in figure 3.6. Individual time slices of the test pattern are referenced by index j. Individual time slices for each template k are referenced by index i. In order to find the minimum path through the composite array, a minimum accumulated distance $D(i,j,k)$ is defined for each grid point (i,j,k). Each point $D(i,j,k)$ is the minimum sum of local distances $d(i,j,k)$ along some path to grid point (i,j,k). For any grid point (i,j,k), $D(i,j,k)$ is found by selecting the predecessor with the minimum accumulated distance and adding that accumulated distance to the local distance $d(i,j,k)$. The transition rules consist of within-template rules and between-template rules. Thus for the template interior, $j > 1$, the recursion rule is,

$$D(i,j,k) = d(i,j,k) + \min[D(i-1,j,k), D(i-1,j-1,k), D(i,j-1,k)] \qquad (1)$$

At template boundaries with $j = 1$, the recursion rule is,

$$D(i,j,k) = d(i,j,k) + \min[D(i-1,J(k^*),k^*)] \qquad (2)$$

where $k^* = 1,\ldots,K$. Figure 3.7 depicts within-template and between-template transition rules for connected speech distance arrays. By keeping track of where the path crosses template boundaries, the problem of boundary detection in the test pattern is handled automatically.

Time Distortion Penalties. Ideally the total accumulated distance through the distance array should be independent of the slope of the path in order to allow all types of time axis distortion. Therefore, the algorithm applies time distortion penalties using slope dependent weights. Depending on the three directions, horizontal, diagonal, and vertical, the local distance is multiplied by the weights $(1 + a)$, 1, and b prior to evaluating the dynamic programming recursion:

$$\begin{aligned} D(i,j,k) = \min [(1 + a) \cdot d(i,j,k) &+ D(i-1,j,k), \\ d(i,j,k) &+ D(i-1,j-1,k), \\ b \cdot d(i,j-1,k) &+ D(i,j-1,k)] \end{aligned}$$

(In the actual algorithm, this recursive formula is not actually implemented recursively but forwardly as the accumulated distance array is computed.) The number of local distances per input frame is thus 1 + (a/2) for slope 1/2, 1 for slope 1, and 1 + b for slope 2. Figure 3.5 depicts the time distortion penalties. Weights of a = 1 and b = 1/2 are typically used

<u>Summary of Steps.</u> A summary of connected speech algorithm is given as follows.

```
Step 1)   Initialize $D(1,j,k) = \sum_{n=1}^{j} d(1,n,k)$.
Step 2)
  a) For $i = 2,\ldots, N$, do steps 2b-2e.
  b) For $k = 1,\ldots, K$, do steps 2c-2e.
  c) $D(i,1,k) = d(i,1,k) + \min[D(i-1,j(k^*),k^*)]$.
  d) For $j = 2,\ldots,j(k)$, do step 2e.
  e) $D(i,j,k) = \min [(1 + a) \bullet d(i,j,k) + D(i-1,j,k),$
                       $d(i,j,k) + D(i-1,j-1,k),$
                   $b \bullet d(i,j-1,k) + D(i,j-1,k)]$.

Step 3)  Trace back the path from the grid point at a template
         ending frame with the minimum total distance using array
         D(i,j,k) of accumulated distances.
```

The unknown sequence is recovered in step 3) above by tracing back the decisions taken by the "minimum" operator at each grid point. (10:265)

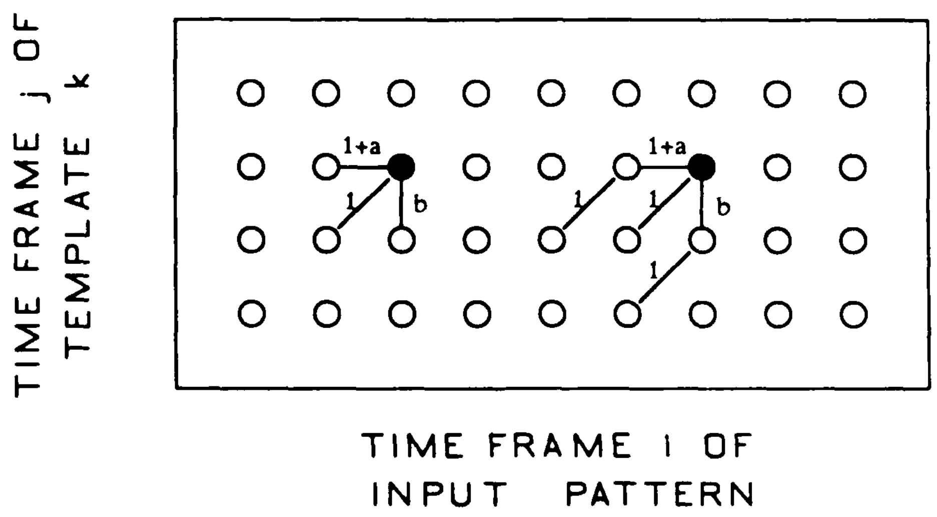


Figure 3.5 Time Distortion Penalties

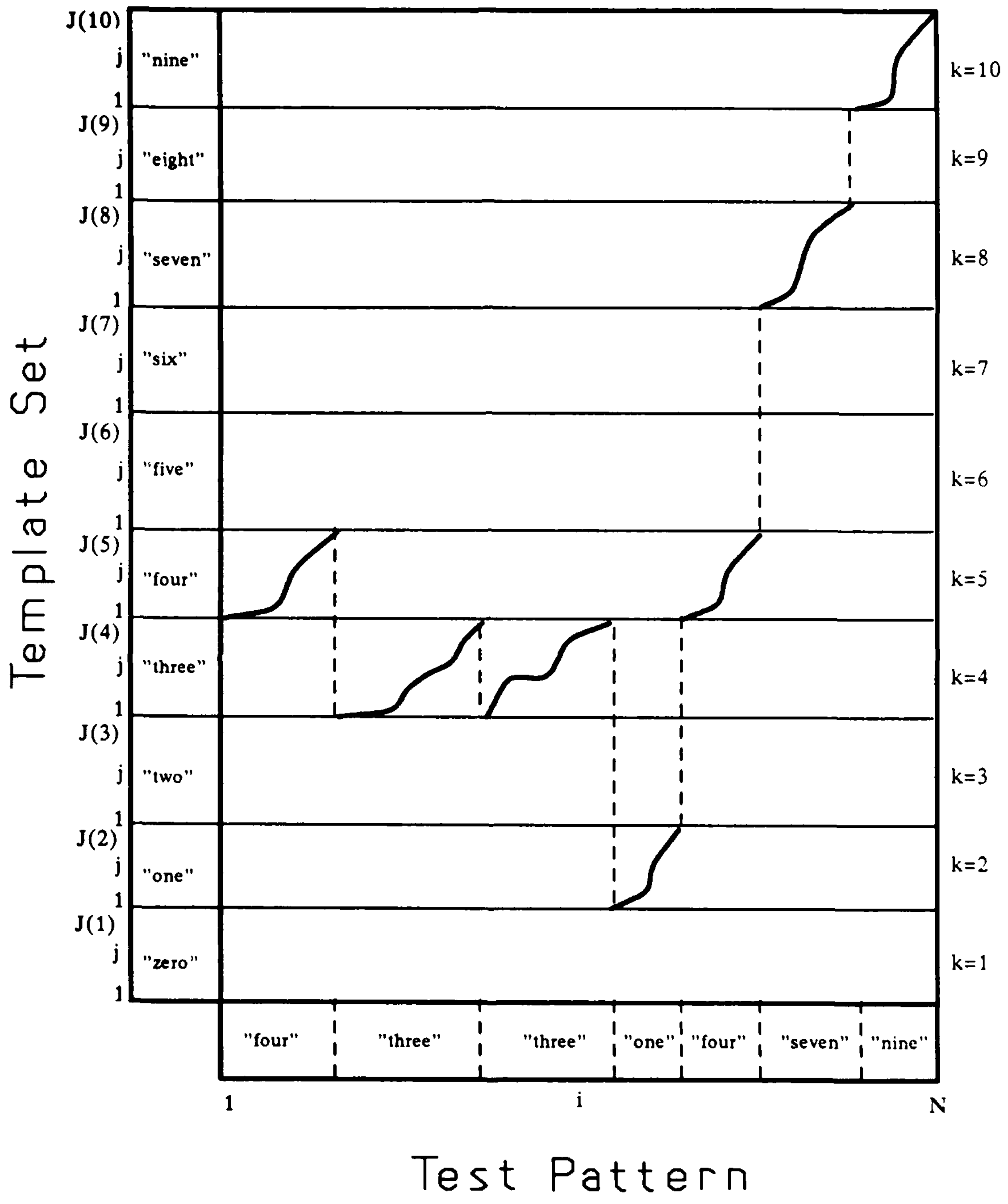
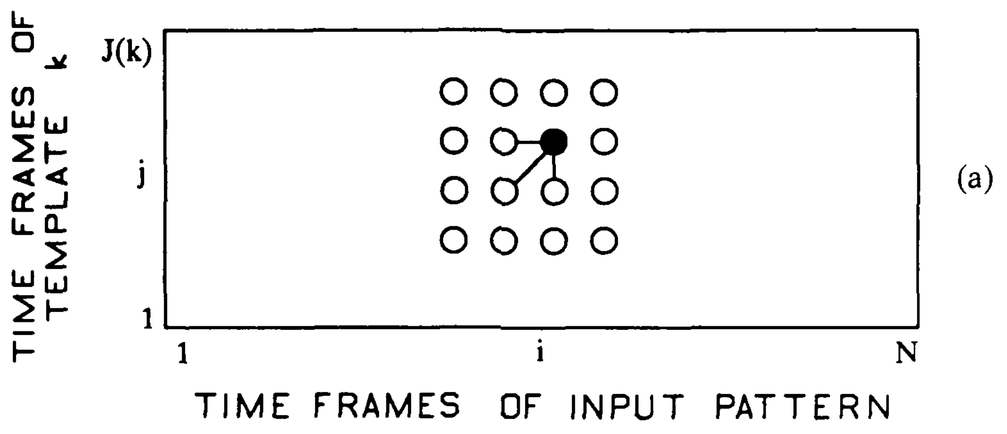


Figure 3.6 Hypothetical Distance Array for Continuous Speech



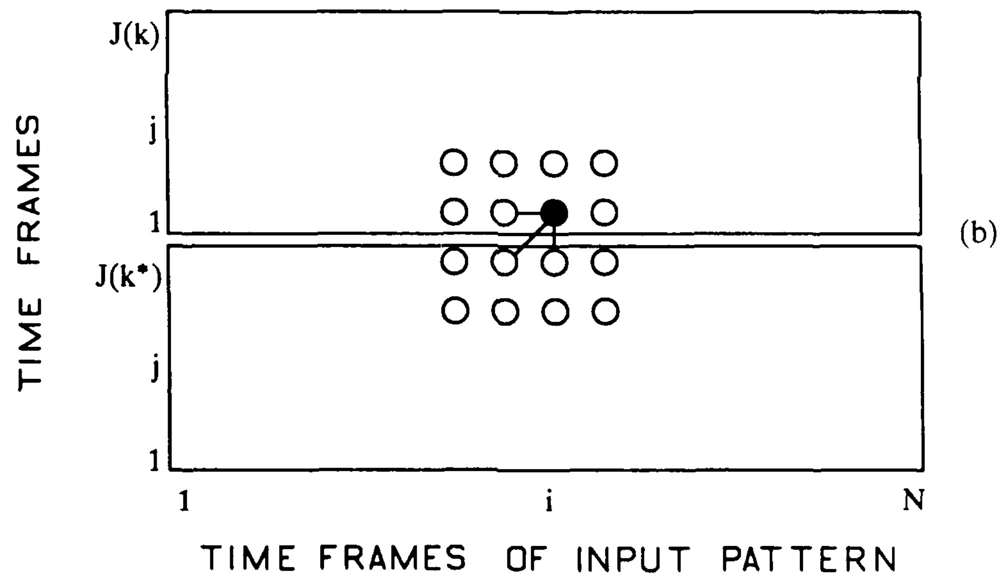


Figure 3.7 (a) Within Template Transition Rules
(b) Between Template Transition Rules

Storage. In reality, the whole accumulated array need not be computed and stored at once. To perform the dynamic programming recursions from a time frame i, only a small portion of the the complete array D(i,j,k) of accumulated distances is needed. Thus using only one column of storage, D(j,k), the recursions (1) and (3) are carried out by proceeding along the time axis of the test pattern and updating the storage column point by point. Using this method causes the details of the path to be lost, and backtracking information (boundary crossings) must be stored along the way. Two 1-dimensional arrays, length N, are used for this purpose. The words and boundaries are finally found by tracing back through the 1-dimensional arrays from end point to begin point, etc., until the beginning of the test pattern is reached. Figure 3.8 depicts the idea of backpointers for individual grid points while figure 3.9 depicts the traceback procedure. A flow diagram for the One-Stage Dynamic Time Warping Algorithm for Connected Speech is shown in figure 3.10. For more details refer to Ney [10] or Appendix B for the Lisp function SCAN-DTW.

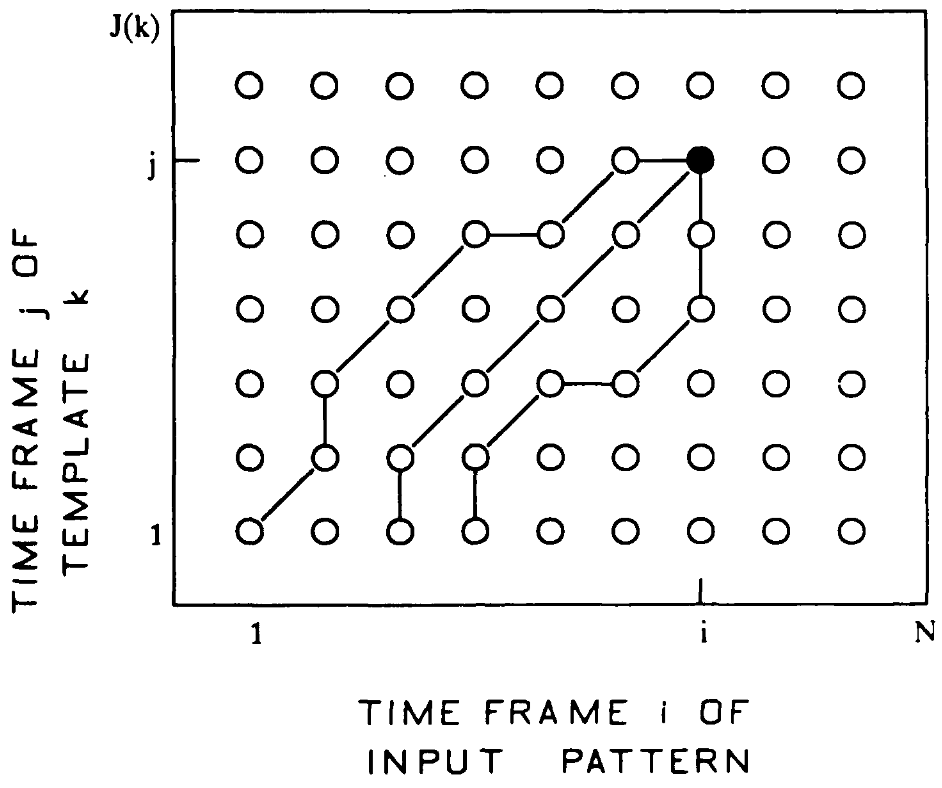


Figure 3.8 Backpointers from three preceding grid points (i,j,k) to their starting frames

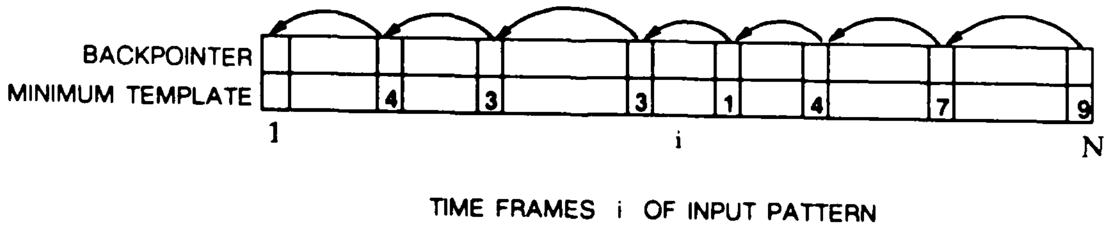
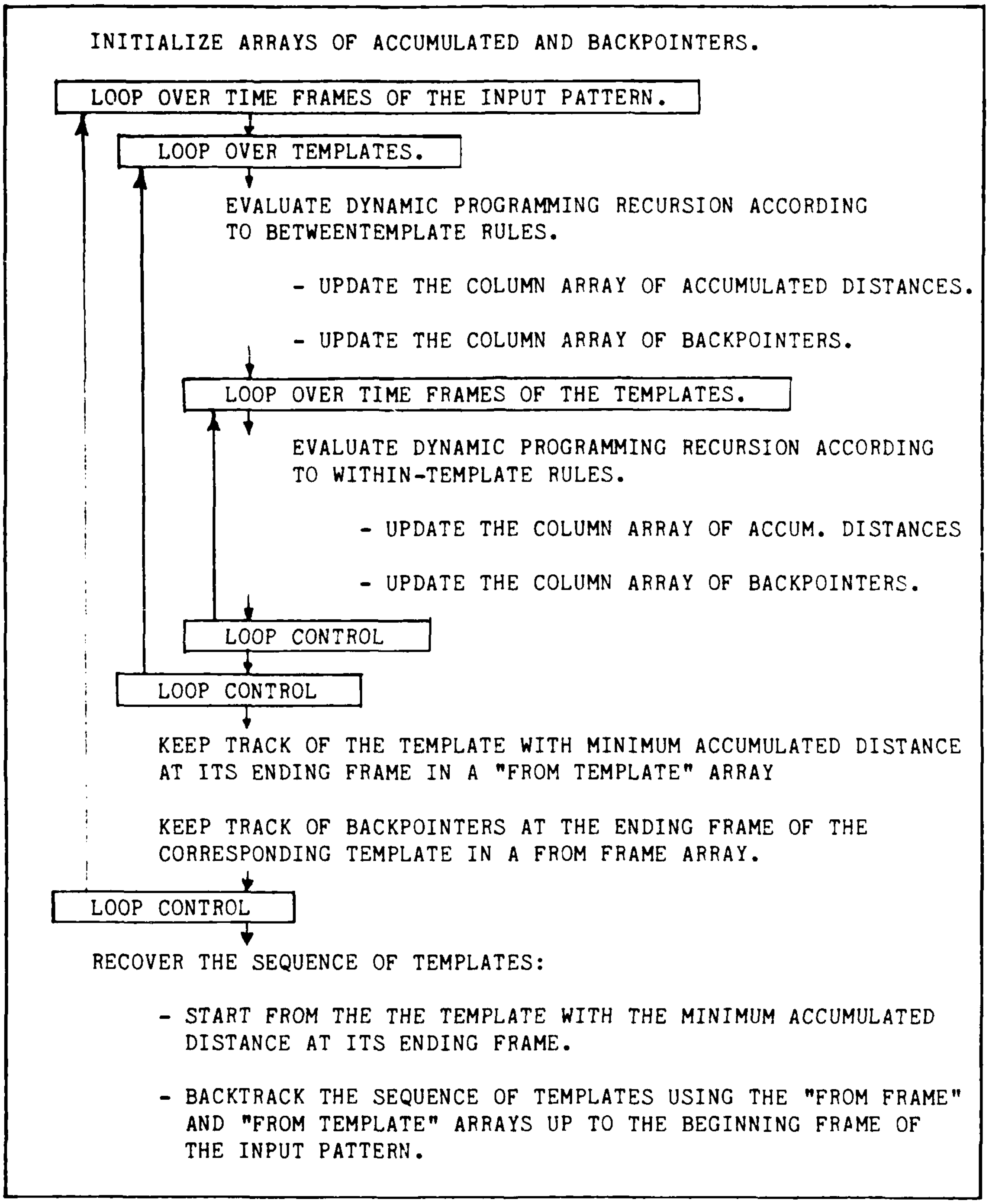


Figure 3.9 The Backtracking Procedure



```
INITIALIZE ARRAYS OF ACCUMULATED AND BACKPOINTERS.

LOOP OVER TIME FRAMES OF THE INPUT PATTERN.
    LOOP OVER TEMPLATES.
            EVALUATE DYNAMIC PROGRAMMING RECURSION ACCORDING
            TO BETWEENTEMPLATE RULES.

                 - UPDATE THE COLUMN ARRAY OF ACCUMULATED DISTANCES.

                 - UPDATE THE COLUMN ARRAY OF BACKPOINTERS.

        LOOP OVER TIME FRAMES OF THE TEMPLATES.
                 EVALUATE DYNAMIC PROGRAMMING RECURSION ACCORDING
                 TO WITHIN-TEMPLATE RULES.

                      - UPDATE THE COLUMN ARRAY OF ACCUM. DISTANCES

                      - UPDATE THE COLUMN ARRAY OF BACKPOINTERS.
        LOOP CONTROL
    LOOP CONTROL
         KEEP TRACK OF THE TEMPLATE WITH MINIMUM ACCUMULATED DISTANCE
         AT ITS ENDING FRAME IN A "FROM TEMPLATE" ARRAY

         KEEP TRACK OF BACKPOINTERS AT THE ENDING FRAME OF THE
         CORRESPONDING TEMPLATE IN A FROM FRAME ARRAY.
LOOP CONTROL
    RECOVER THE SEQUENCE OF TEMPLATES:

         - START FROM THE THE TEMPLATE WITH THE MINIMUM ACCUMULATED
           DISTANCE AT ITS ENDING FRAME.

         - BACKTRACK THE SEQUENCE OF TEMPLATES USING THE "FROM FRAME"
           AND "FROM TEMPLATE" ARRAYS UP TO THE BEGINNING FRAME OF
           THE INPUT PATTERN.
```

Figure 3.10 Schematic Diagram
Source (3:268).

IV. Results and Discussion

Introduction

The purpose of this chapter is to present the results of an investigation into the applicability of various feature sets to connected-speech recognition. The chapter begins by examining features one by one and observing their distance array contours. Distance distributions for the distance arrays are given in the form of histograms. Finally, individual features and combinations of features are tested at connected-speech recognition using the one-stage dynamic time warping algorithm for connected speech of chapter three.

The Distance Array Contour

The distance array contour is a useful way of observing a feature set's applicability to speech recognition. The distance array contour is made by calculating an array of distances between a single known template word and a single known utterance. Distances calculated are Minkowski 1 distances (taxi distance). Then the distance array is plotted with distances below a certain threshold represented by black, and distances above the threshold represented by white. The threshold is determined by trial and error until about half the area is dark and half is white. This threshold varies for different feature sets. Figures 4.1 through 4.7 show distance array contours using the template word "three" and the utterance "4-3-3-1-4-7-9", along with corresponding distribution histograms.

Wide-Band Spectrum. Figure 4.1 shows the distance array contour using Wide-Band Spectrum as a feature set. Notice where the word "three" appears in the test pattern, there are diagonal dark patterns extending from bottom to top of the distance array contour. Those dark diagonal patterns represent the occurrences of "three" in the test pattern matching up with the template version of "three". Even though the original waveforms of the two occurrences of "three" are markedly different, this feature is able to show agreement with each and the template.

Narrow-Band Spectrum. Figure 4.2 shows distance array contour using narrow-band spectrum as a feature set. This contour looks very similar to the one for wide-band spectrum except that the patterns are more distinct. There is less "noise" in the narrow-band spectrum representation.

LPC Coefficients. Figure 4.3 shows the distance array contour using LPC Coefficients as a feature set. In this feature set, the distances are calculated from the actual LPC filter coefficients. In this case the patterns are not clear. Using this feature in this way performs poorly in terms of showing agreement between the template and the test pattern.

LPC Spectrum. Figure 4.4 shows the distance array contour using LPC Spectrum as a feature set. In this feature set, the distance are calculated from the spectral components derived from the LPC filter coefficients. In this contour, the two occurrences of the word "three" appear even more clearly than wide-band and narow-band spectrum.

Formants. Figure 4.5 shows the distance array contour using Formants as a feature set. This feature set works well during vowels sounds, but works erratically during fricatives when the formant tracker loses track. This contour shows that formants can show agreement between vowel sounds but not fricatives.

Frication Frequency. Figure 4.6 shows the distance array contour using Frication Frequency as a feature set. This contour fails to show any agreement between the template and the input pattern.

Zero Crossing Rate. Figure 4.7 shows the distance array contour using Zero Crossing Rate as a feature set. This contour also fails to show any agreement between the template and the input pattern.

Recognition of Connected Speech.

In order to fully observe the feature sets' applicability to connected-speech recognition different features and combinations thereof are tested using the one-stage dynamic time warping algorithm for connected speech proposed by Ney (10). Appendix C contains sample results of the recognition system using the various feature sets for speaker-dependent continuous speech recognition. In these figures, the template set is displayed vertically on the left and the test pattern horizontally on the bottom. The composite distance array contour is shown in the middle. The template word boundaries are marked by horizontal lines. The vertical lines represent the word boundaries with the test pattern as computed by the recognition system. Below the test pattern waveform are the words as recognized by the system.

Speaker-Dependent Results. Feature sets are tested first for speaker-dependent performance. In this case the template patterns and the test patterns are made by the same speaker. The features tested here are wide-band, narrow-band, and LPC spectrum, formants, and a combination of formants, LPC spectrum, and frication frequency. Table 4-1 shows results for each of these feature sets.

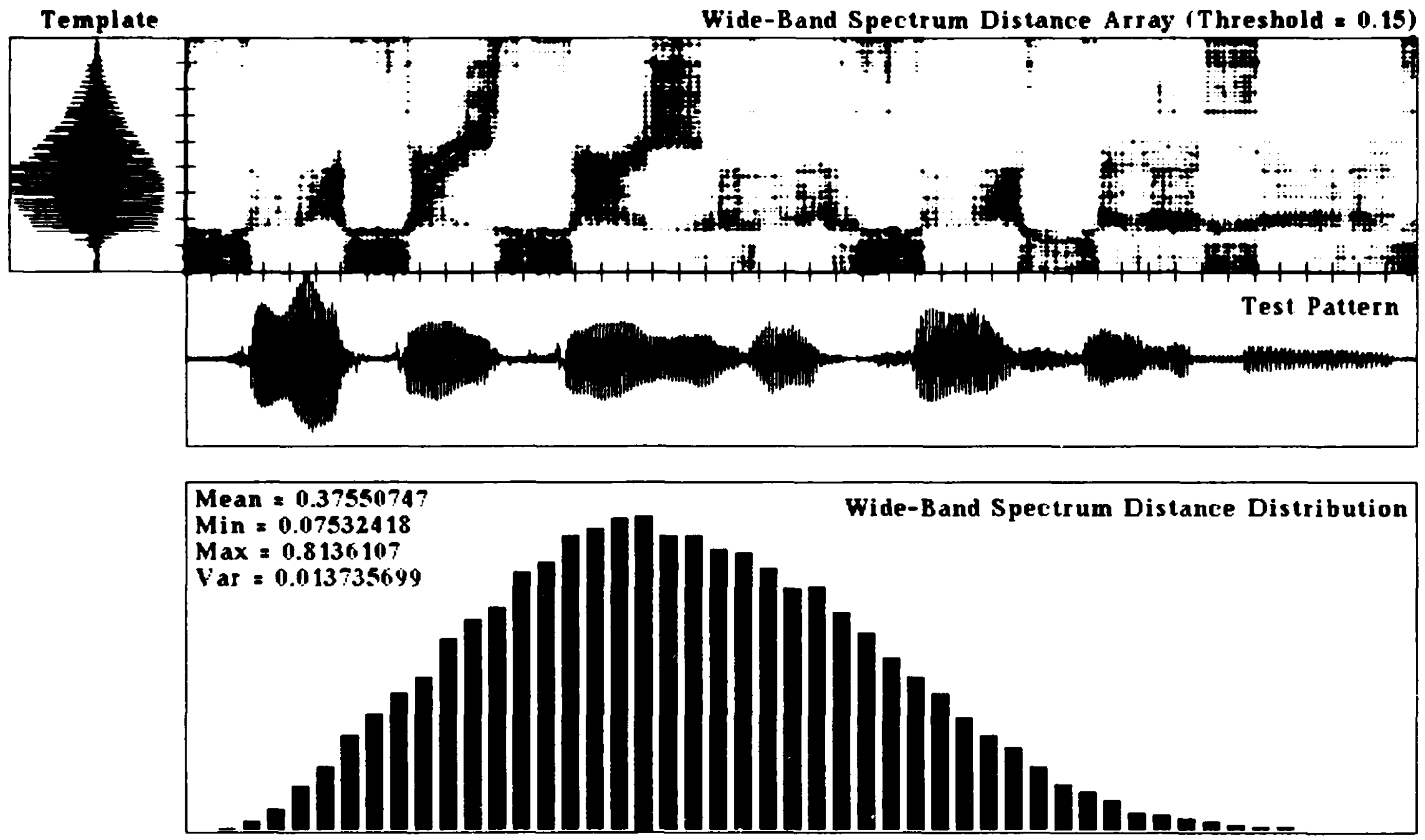


Figure 4.1 Distance Array Contour, Wide-Band Spectrum

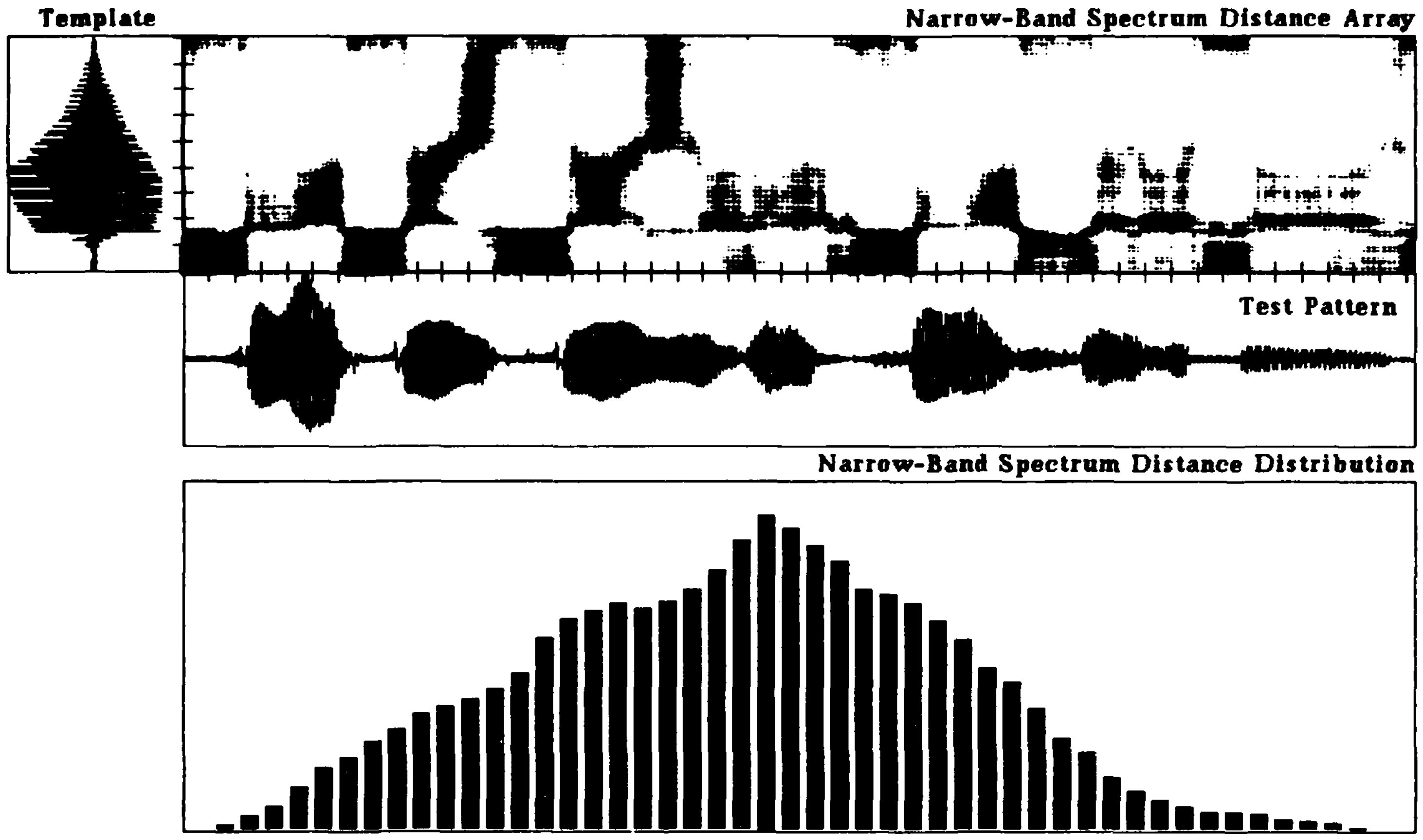


Figure 4.2 Distance Array Contour, Narrow-Band Spectrum

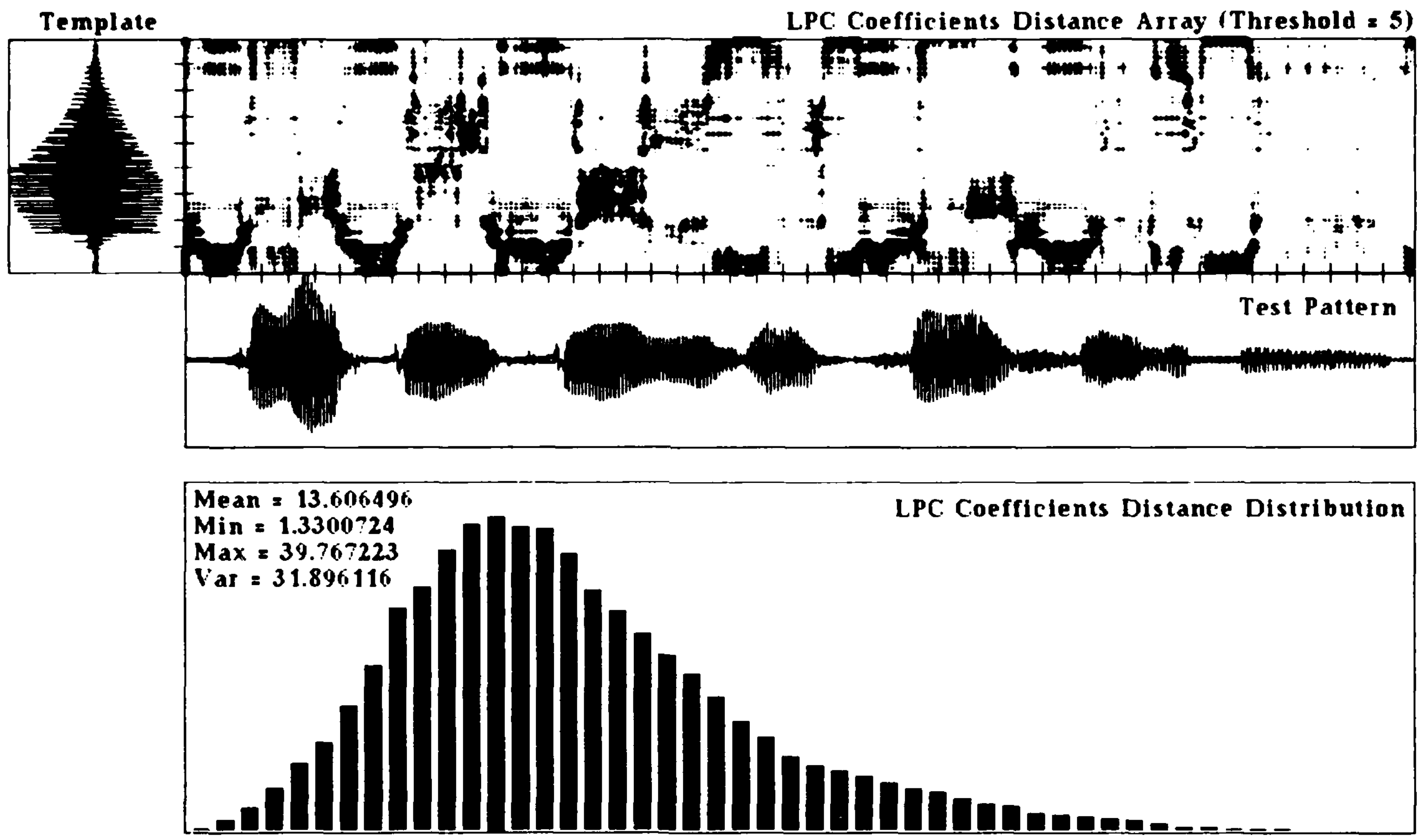


Figure 4.3 Distance Array Contour, LPC Coefficients

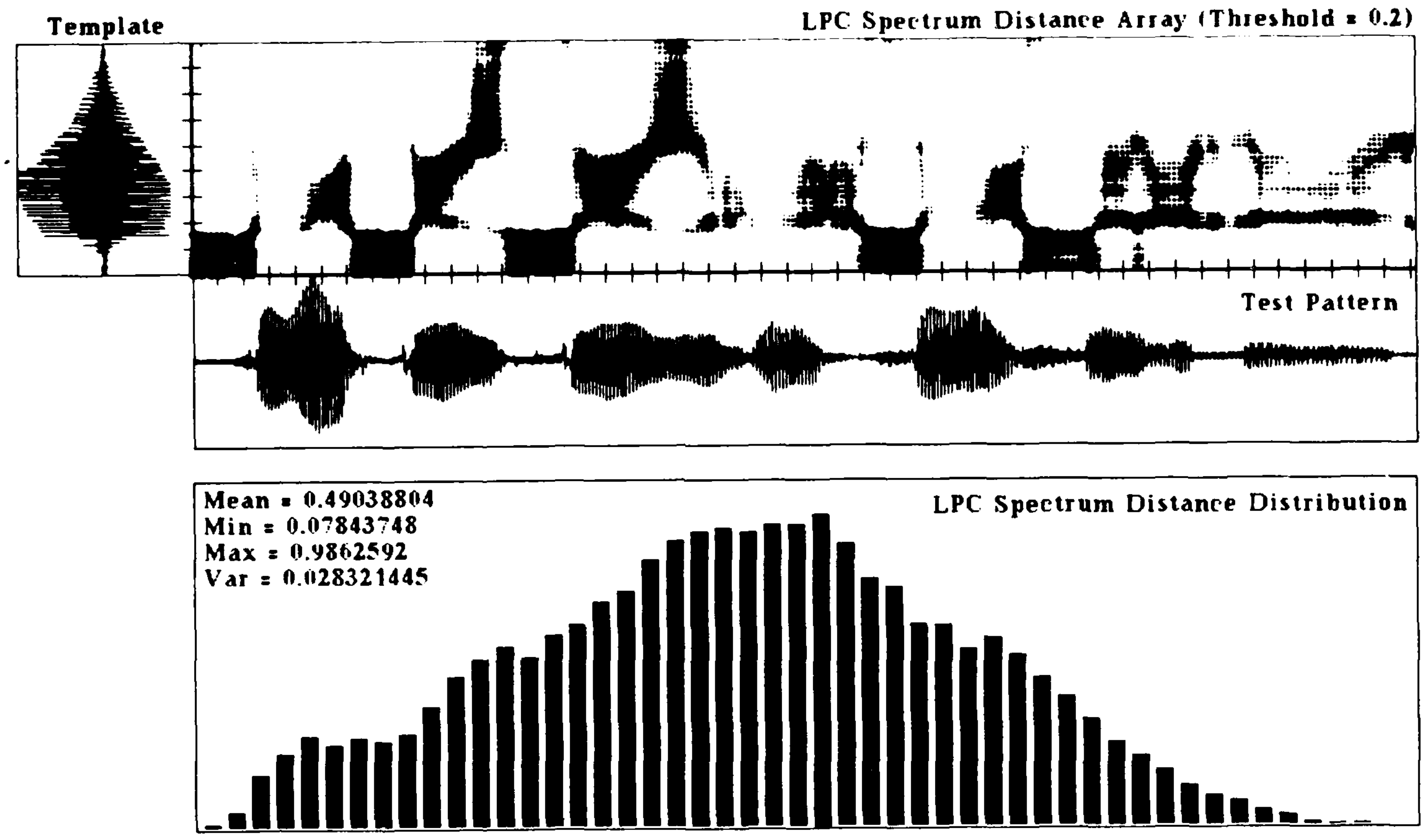


Figure 4.4 Distance Array Contour, LPC Spectrum

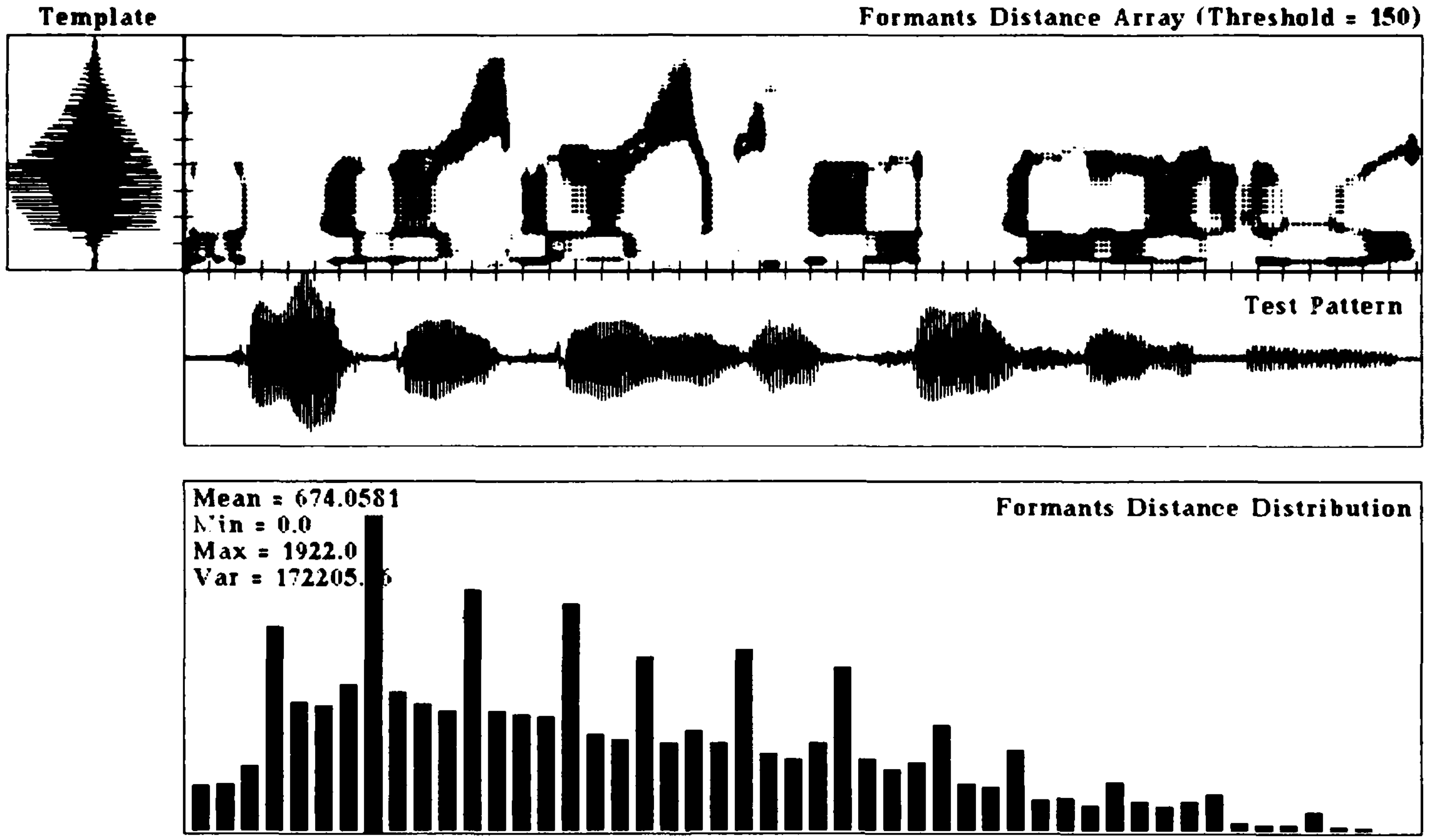


Figure 4.5 Distance Array Contour, Formants

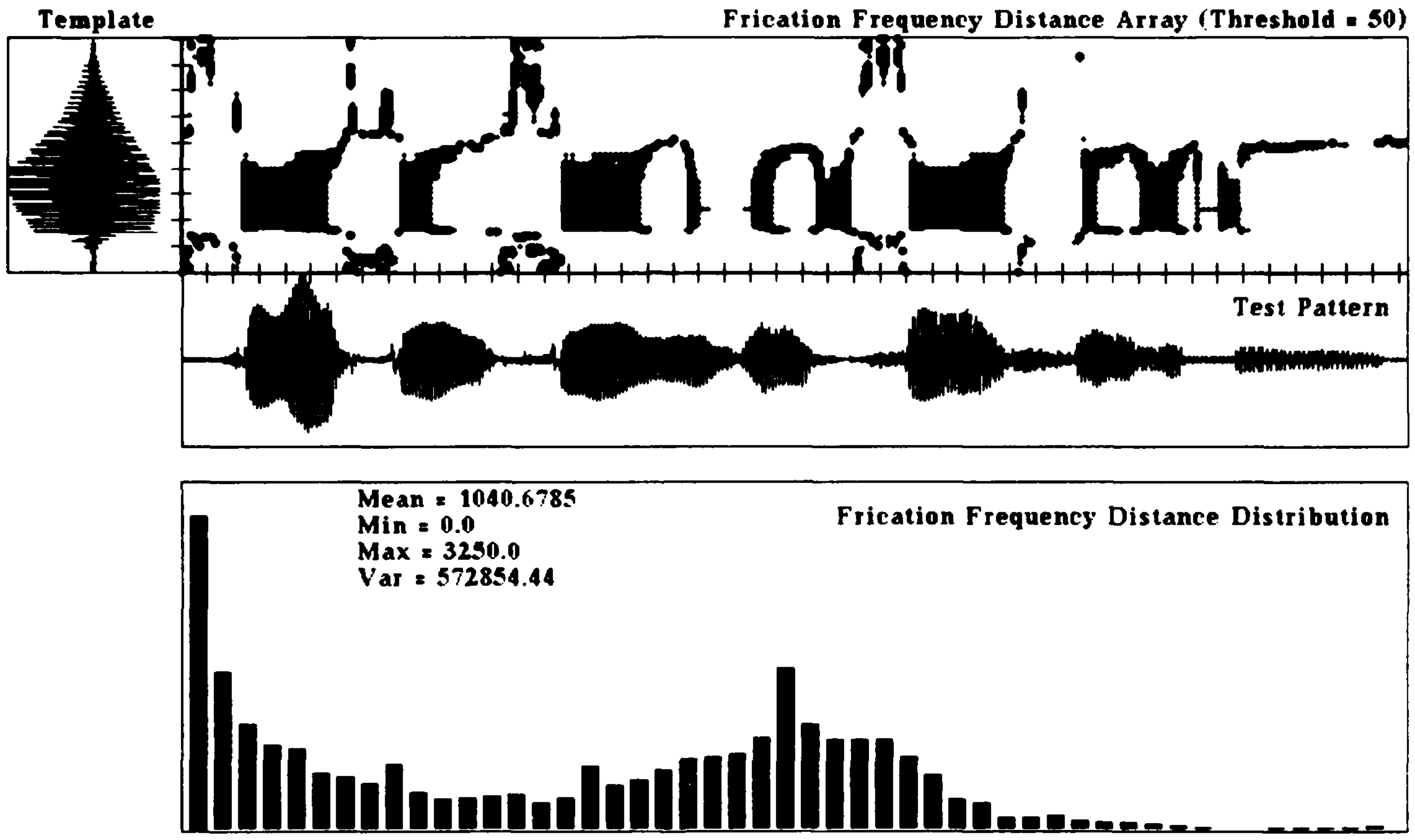


Figure 4.6 Distance Array Contour, Frication Frequency

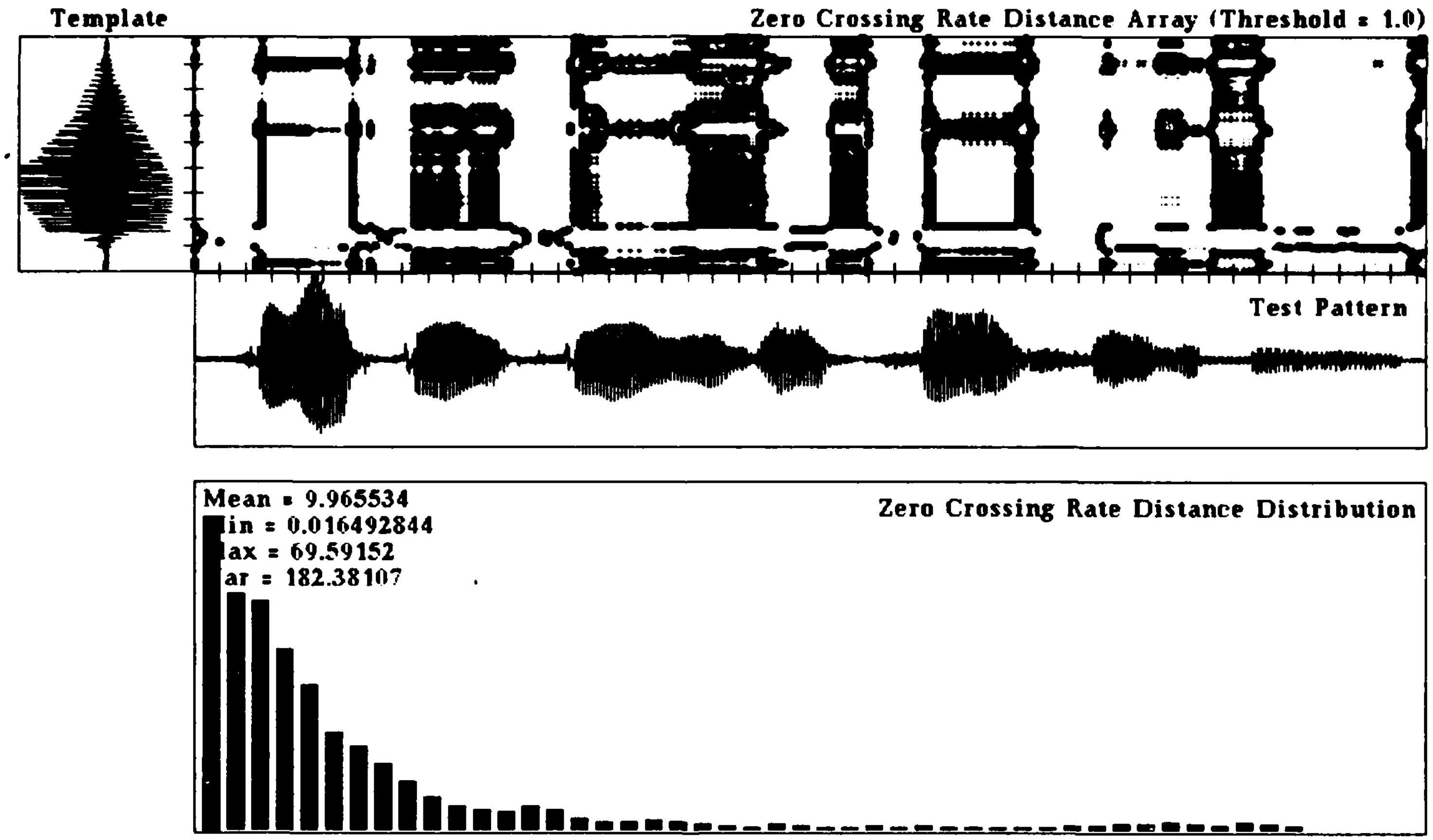


Figure 4.7 Distance Array Contour, Zero Crossing Rate

Wide-Band Spectrum. The feature set consisting of only wide-band spectrum performs only fair, correctly recognizing 29 out of 38 digits, spoken in five to seven word utterances for the speaker "RGD".

Narrow-Band Spectrum. This feature set performs about the same as wide-band spectrum, also correctly recognizing 29 out of 38 digits, spoken in five to seven word utterances for the speaker "RGD".

Formants. The feature set consisting of only the first and second formant frequencies performed surprisingly well, recognizing 31 out of 38 spoken in five to seven word utterances for the speaker "RGD". The good performance of this feature, considering how the formant tracts are lost during fricatives leads to the next feature set, which is a combination of LPC Spectrum and Formants.

LPC Spectrum. The feature set consisting of only LPC spectrum performs the best of the three spectrum features, correctly recognizing 35 out of 38 spoken in five to seven word utterances for the speaker "RGD".

LPC Spectrum, Formants, Frication Frequency. This feature set consists of a combination of LPC spectrum and formants. Frication frequency is used as a "gate" to determine whether vowels or fricatives are present. Zero crossing rate could also be used as a gate between vowel and fricative sounds, because a rate between about 300 and 900 usually indicates a vowel sound. However, zero crossing rate goes to zero during very low energy periods as shown by figure 4.8.

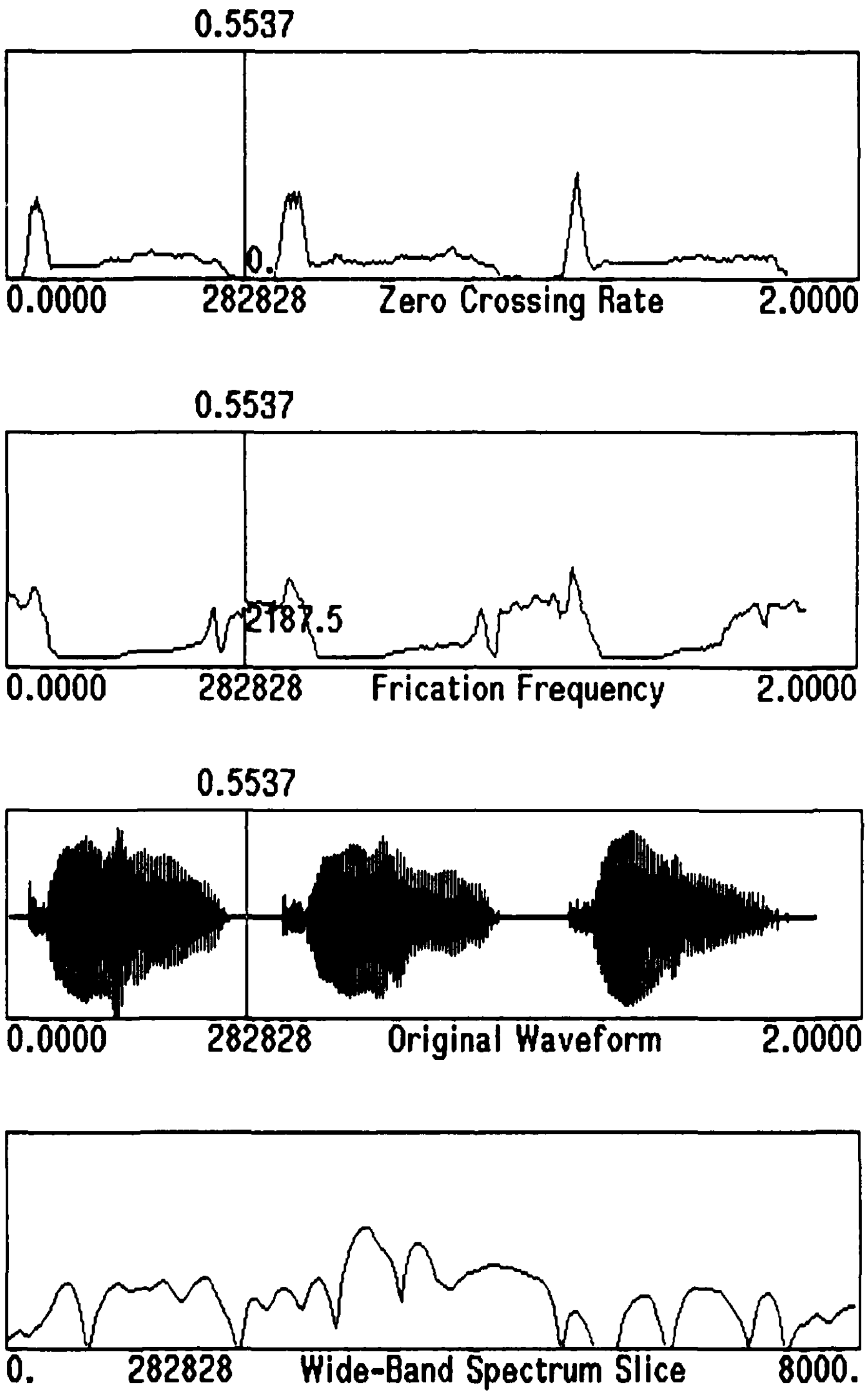


Figure 4.8 Frication Frequency vs. Zero Crossing Rate

Therefore, moments between silence and frication, as the zero crossing rate rises from 0 above 900, would be mistaken for vowel sounds. Using frication frequency as a "gate" enables formant tracts to be used while substituting LPC spectrum distances when the formant frequencies are not valid. This feature set performed very well, correctly recognizing all 38 of the digits spoken by "RGD". Even the troublesome "two-eight-two-eight-two-eight" combination was correctly recognized.

Utterance	Wide-Band Spectrum	Narrow-Band Spectrum	Formants	LPC Spectrum	LPC Spectrum Formants Fric. Freq.
4331479	7.0/7.0	7.0/7.0	7.0/7.0	7.0/7.0	7.0/7.0
282828	0.0/6.0	0.0/6.0	4.0/6.0	5.0/6.0	6.0/6.0
2468	3.0/4.0	3.0/4.0	4.0/4.0	4.0/4.0	4.0/4.0
28318	3.0/5.0	3.0/5.0	2.0/5.0	3.0/5.0	5.0/5.0
012345	6.0/6.0	6.0/6.0	6.0/6.0	6.0/6.0	6.0/6.0
56789	5.0/5.0	5.0/5.0	5.0/5.0	5.0/5.0	5.0/5.0
01379	5.0/5.0	5.0/5.0	4.0/5.0	5.0/5.0	5.0/5.0
Total	29.0/38.0	29.0/38.0	32.0/38.0	35.0/38.0	38.0/38.0
Percent	63%	63%	84%	92%	100%

Table 4.1 Speaker Dependent Feature Results

Speaker-Independent Results. Two feature sets are used to examine speaker-independent connected speech recognition. LPC spectrum, since it is so commonly used in practice, is used as a baseline. A possibly improved feature set, using LPC spectrum, formants, and frication frequency is also used. Speaker-independent performance is examined by simply trying the system out using various combinations of template sets and test patterns, of course each by different speakers. Finally, multiple speaker template sets are tested.

The improved feature set of LPC spectrum, frication frequency, and formants, is implemented differently than for the speaker-dependent case. Formant frequencies are rather consistent for given vowels sounds for a given speaker. However, formant frequencies of different speakers uttering vowels sounds that are perceived as being the same can be quite different. Figure 4.9 shows a plot of the first formant frequency (F1) versus the second formant frequency (F2) for a population of speakers uttering vowel sounds common to the English language. Those grouped together were perceived as the same sound. It is clear from figure 4.9 that simple Minkowski 1 distances are insufficient since points from separate groups can have Minkowsli 1 distances that are smaller than points from within the same group. Therefore, it is necessary to alter the way these individual features are combined for the speaker-dependent case.

First, a distance array is computed using only LPC spectrum. As in the speaker-dependent case, frication frequency is used to locate valid formant frequencies. Then, each point in the LPC spectrum distance array is multiplied by 0.4 and thereby emphasizing "agreement" if, (1) that point results from a valid vowel sound according to frication frequency in both the template and the test pattern, and (2) the first and second formants from both the template and the test pattern fall within the same group. Figure 4.10 shows the groupings used by the algorithm for each vowel sound.

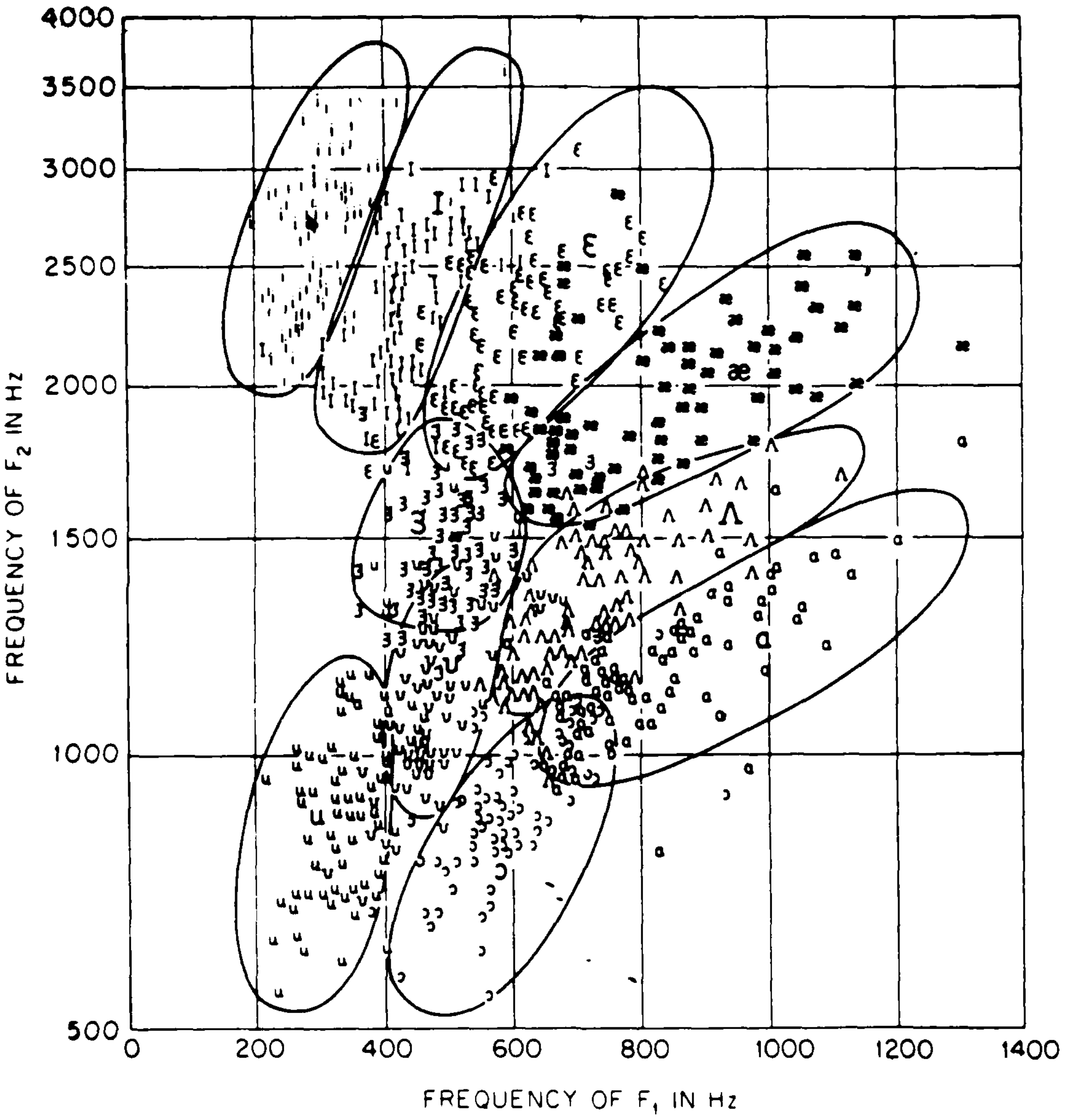


Figure 4.9 First Formant vs. Second Formant
Source (12:44)

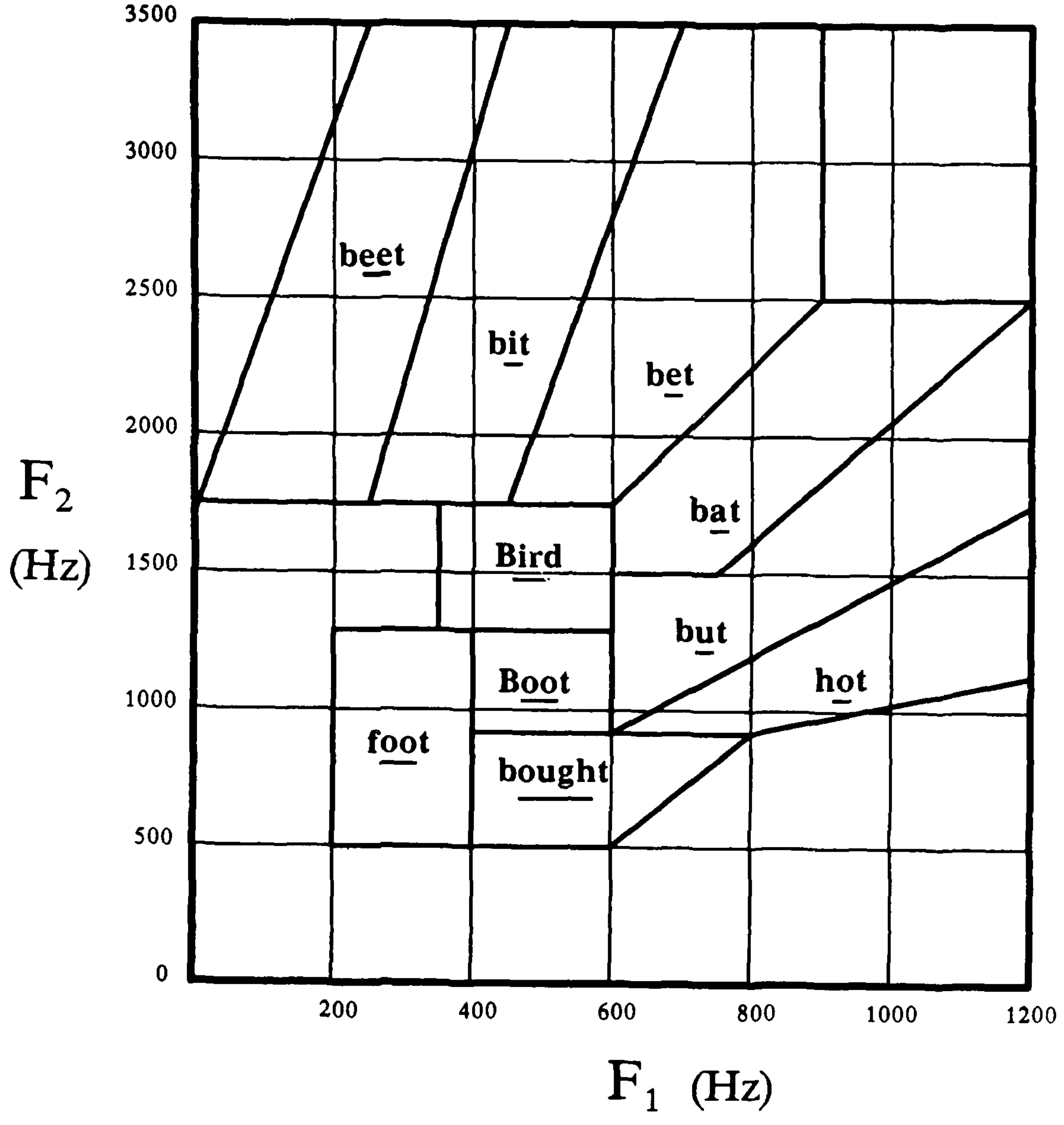


Figure 4.10 Formants Zones Used by Algorithm

Single-Speaker Template Sets. Table 4.2 shows results for various template and speaker combinations. In many cases, the addition of formant information improved recognition accuracy.

Template	Speaker	LPC Spectrum	Plus Formants
JONES	RGD	29.5/38.0	34.5/38.0
JONES	SKIP	7.5/18.0	11.5/18.0
RGD	SKIP	15.5/18.0	18.0/18.0
RGD	JONES	26.0/33.0	31.0/33.0
SKIP	RGD	26.5/38.0	28.0/38.0
SKIP	JONES	25.0/33.0	31.0/33.0
TOTAL		130.0/178.0	154.0/178.0
PERCENT		73%	87%

Table 4.2 Single Template Speaker Independent Results

Multi-Speaker Template Sets. Table 4.3 shows results for various multi-speaker template and speaker combinations. Using multi-speaker templates further improved recognition accuracy.

Template	Speaker	LPC Spectrum	Plus Formants
SKIP & RGD	JONES	27.5/33.0	31.0/33.0
JONES & RGD	SKIP	12.5/18.0	18.0/18.0
SKIP & JONES	RGD	31.5/38.0	35.5/38.0
TOTAL		71.5/89.0	84.5/89.0
PERCENT		80%	95%

Table 4.3 Multi-Template Speaker Independent Results

Overall Results Using LPC Spectrum. Table 4.4 shows the overall results including both single and multi-templates using only LPC spectrum as a feature set.

	Template (LPC Spectrum)					
Utterance	RGD	JONES	SKIP	SKIP RGD	JONES SKIP	JONES RGD
RGD:						
012345	-	5.0/6.0	6.0/6.0	-	6.0/6.0	-
4331479	-	7.0/7.0	5.0/7.0	-	7.0/7.0	-
56789	-	2.5/5.0	1.5/5.0	-	2.5/5.0	-
28318	-	2.0/5.0	3.0/5.0	-	2.0/5.0	-
01379	-	3.0/5.0	5.0/5.0	-	5.0/5.0	-
2468	-	4.0/4.0	3.0/4.0	-	3.0/4.0	-
282828	-	6.0/6.0	3.0/6.0	-	6.0/6.0	-
JONES:						
4331479	6.0/7.0	-	7.0/7.0	7.0/7.0	-	-
2555276	7.0/7.0	-	4.0/7.0	7.0/7.0	-	-
28318	2.5/5.0	-	3.0/5.0	3.0/5.0	-	-
2377097	3.5/7.0	-	4.0/7.0	3.5/7.0	-	-
8351561	7.0/7.0	-	7.0/7.0	7.0/7.0	-	-
SKIP:						
1234	4.0/4.0	1.0/4.0	-	-	-	2.0/4.0
1549768203	9.0/10.0	4.0/10.0	-	-	-	8.0/10.0
2468	2.5/4.0	2.5/4.0	-	-	-	2.5/4.0

Table 4.4 Overall Results
LPC Spectrum

Overall Results Using LPC Spectrum, Formants, and Frication Frequency. Table 4.5 shows the overall results including both single and multi-templates using only LPC spectrum, formants, and frication frequency combined as a feature set.

	Template (LPC Spectrum + Formants)					
Utterance	RGD	JONES	SKIP	SKIP RGD	JONES SKIP	JONES RGD
RGD:						
012345	-	6.0/6.0	6.0/6.0	-	6.0/6.0	-
4331479	-	7.0/7.0	5.5/7.0	-	7.0/7.0	-
56789	-	3.5/5.0	2.5/5.0	-	3.5/5.0	-
28318	-	5.0/5.0	1.0/5.0	-	5.0/5.0	-
01379	-	4.0/5.0	5.0/5.0	-	5.0/5.0	-
2468	-	3.0/4.0	3.0/4.0	-	3.0/4.0	-
282828	-	6.0/6.0	5.0/6.0	-	6.0/6.0	-
JONES:						
4331479	7.0/7.0	-	7.0/7.0	7.0/7.0	-	-
2555276	7.0/7.0	-	7.0/7.0	7.0/7.0	-	-
28318	4.5/5.0	-	4.0/5.0	5.0/5.0	-	-
2377097	5.5/7.0	-	6.0/7.0	6.0/7.0	-	-
8351561	7.0/7.0	-	7.0/7.0	7.0/7.0	-	-
SKIP:						
1234	4.0/4.0	2.0/4.0	-	-	-	4.0/4.0
1549768203	10.0/10.0	7.0/10.0	-	-	-	10.0/10.0
2468	4.0/4.0	2.5/4.0	-	-	-	4.0/4.0

Table 4.5 Overall Results
LPC Spectrum, Formants,
Frication Frequency

V. Conclusions and Recommendations

Introduction

The purpose of this chapter is to discuss conclusions that may be drawn based on the performance of this system as well as to give recommendations for further research in the area of speaker-independent continuous-speech recognition.

Conclusions.

This thesis is successful in producing a rather robust system for continuous-speech recognition. It is shown here that Ney's algorithm for connected speech works quite well. The idea of using template sets made up of multiple speech features is also shown to be advantageous. Results reveal that using formant information can significantly improve recognition accuracy, especially in the area of speaker-independent applications.

Recommendations.

Environmental Stress. As described in chapter 3, this system was tested with speech patterns virtually free of background noise. It would be interesting to study its performance under such conditions as background noise ie., cockpit noise. The Armstrong Aerospace Medical Research Laboratory at Wright-Patterson AFB has excellent facilities for recording speech under noise conditions.

Tailored Template Sets. From the results of this system it still isn't clear whether completely redundant template sets are necessary. They seem to be useful handling different pronunciations of certain words such as "eight" with or without the "t" sound at the end.

Unfortunately, redundant template sets pay a high price in terms of computational intensity. A better approach may be to store a few carefully selected template sets with only certain words redundant and let the user select the best one for him. This would greatly simplify the training process.

Additional Features. Although the system was able to discriminate between different vowel sounds well, it was not able to discriminate between similar fricative sounds. It would have trouble with something like "carp" versus "tarp". A logical extension would be to add ways to discriminate such sounds.

Syntactic Rules. Even humans have trouble identifying spoken utterances without the aid of syntax. Ney (10) describes methods for adapting the algorithm to include such constraints. Also, currently, the algorithm will apply every bit of the test pattern to some template. It has no way of handling words that are not part of the vocabulary. Syntactic constraints described by Ney could possibly be adapted to handle words not in the vocabulary.

Dedicated Hardware. Although Ney's algorithm is very efficient, dedicated hardware would be preferred for its interactive use. Hardware to perform real time LPC analysis is commonly available. The DoD standard is known as LPC-10. The next step would be to implement the dynamic time warping algorithm in hardware as well. Such a system then could conceivably by operated in real time.

Summary

In summary, this thesis shows that using additional speech features (formants) can be successfully applied to the problem of speaker-independent continuous speech recognition. Presumably, further improvements could be made by carefully utilizing other features of speech. Consequently, further research in this area could help to ultimately solve the problem of speech recognition.

Appendix A: Spire Default Values

```
;;; -*- mode: lisp; package: spire; base: 10 -*-
;;;
;;;          SPIRE -- Speech and Phonetics Interactive Research Environment
;;;
;;;                              ATTRIBUTE-DEFAULTS
;;;
;;;
;;; (c) Copyright 1983, Massachusetts Institute of Technology, All Rights Reserved
;;;
;;;

(define-attribute "Zero Crossing Rate" zero-crossing-rate-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :analysis-window-size .020
  :noise-threshold 40.)

(define-attribute "Vers Zero Crossing Rate" zero-crossing-rate-flavor
                  nil nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 400.
  :analysis-window-size .020
  :noise-threshold 40.)

(define-attribute "LPC Predictor Coefficients" lpc-flavor (indexed-attribute-window)
                  (("LPC Gain Term" (:gain) sampled-attribute-window))
  :waveform-attribute-name "Original Waveform"
  :filter-spec (:bandwidth 78.)
  :analysis-rate 200.)

(define-attribute "LPC Spectrum" lpc-spectral-flavor
                  nil nil
 :predictor-attribute-name "LPC Predictor Coefficients"
 :number-of-points 256.)

(define-attribute "LPC Spectrum Slice" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "LPC Spectrum"
  :cursor-name :cursor-time)

(define-attribute "LPC Spectrum Slice (marker)" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "LPC Spectrum"
  :cursor-name :marker-time)

(define-attribute "Energy -- 0 Hz to 5000 Hz" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200. ; 383.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.
  :preemphasis? :default
  :freq-lo-bound 0
  :freq-hi-bound 5000.)

(define-attribute "Vers Total Energy" energy-from-waveform-flavor
                  nil nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 400.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 128.
  :preemphasis? :default
  :freq-lo-bound nil
  :freq-hi-bound nil)

(define-attribute "Total Energy" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 128.
  :preemphasis? :default
  :freq-lo-bound nil
  :freq-hi-bound nil)

(define-attribute "Energy -- 120 Hz to 440 Hz" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.
  :preemphasis? :default
  :freq-lo-bound 120.
  :freq-hi-bound 440.)

(define-attribute "Vers Energy -- 125 Hz to 750 Hz" energy-from-waveform-flavor
                  nil nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 400.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 128.
  :preemphasis? :default
  :freq-lo-bound 125.
  :freq-hi-bound 750.)

(define-attribute "Energy -- 125 Hz to 750 Hz" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 128.
  :preemphasis? :default
  :freq-lo-bound 125.
  :freq-hi-bound 750.)

(define-attribute "Energy -- 640 Hz to 2800 Hz" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.
  :preemphasis? :default
  :freq-lo-bound 640.
  :freq-hi-bound 2800.)

(define-attribute "Energy -- 3400 Hz to 5000 Hz" energy-from-waveform-flavor
                  (sampled-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.
  :preemphasis? :default
  :freq-lo-bound 3400.
  :freq-hi-bound 5000.)

(define-attribute "Frication Frequency" energy-percentile-flavor
                  (sampled-attribute-window) nil
  :spectral-attribute-name "LPC Spectrum"
  :energy-fraction .25)

(define-attribute "LPC Center of Gravity" energy-mean-flavor
                  (sampled-attribute-window) nil
  :spectral-attribute-name "LPC Spectrum")

(define-attribute "Formants" spectral-peaks-flavor
                  (indexed-attribute-window) nil
  :spectral-attribute-name "LPC Spectrum"
  :number-of-peaks 4)

(define-attribute "Narrow-Band Spectrum" fft-spectral-flavor
                  nil nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.)

(define-attribute "Narrow-Band Spectrum Slice" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "Narrow-Band Spectrum"
  :cursor-name :cursor-time)

(define-attribute "Narrow-Band Spectrum Slice (marker)" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "Narrow-Band Spectrum"
  :cursor-name :marker-time)

(define-attribute "Narrow-Band Spectral Slice" fft-spectral-slice-flavor
                  (spectral-slice-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :cursor-name :cursor-time
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.)

(define-attribute "Narrow-Band Spectral Slice (marker)" fft-spectral-slice-flavor
                  (spectral-slice-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :cursor-name :marker-time
  :filter-type :hamming
  :filter-spec (:bandwidth 78.0)
  :number-of-points 256.)

(define-attribute "Wide-Band Spectrum" fft-spectral-flavor
                  nil nil
  :waveform-attribute-name "Original Waveform"
  :analysis-rate 200.
  :filter-type :hamming
  :filter-spec (:bandwidth 300.0)
  :number-of-points 256.)

(define-attribute "Wide-Band Spectrum Slice" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "Wide-Band Spectrum"
  :cursor-name :cursor-time)

(define-attribute "Wide-Band Spectrum Slice (marker)" spectrum-slice-flavor
                  (spectral-slice-attribute-window) nil
  :spectrum-name "Wide-Band Spectrum"
  :cursor-name :marker-time)

(define-attribute "Wide-Band Spectral Slice" fft-spectral-slice-flavor
                  (spectral-slice-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :cursor-name :cursor-time
  :filter-type :hamming
  :filter-spec (:bandwidth 300.0)
  :number-of-points 256.)

(define-attribute "Wide-Band Spectral Slice (marker)" fft-spectral-slice-flavor
                  (spectral-slice-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :cursor-name :marker-time
  :filter-type :hamming
  :filter-spec (:bandwidth 300.0)
  :number-of-points 256.)

(define-attribute "Narrow-Band Spectrogram" stretched-fft-spectrogram-flavor
                   (spectrogram-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :spectrogram-size 320.
  :analysis-rate 383.
  :filter-type :hamming
  :filter-spec (:bandwidth 76.0)
  :number-of-points 256.
  :white-value -96.                              ;-36
  :black-value -80.)                             ;-20

(define-attribute "Wide-Band Spectrogram" stretched-fft-spectrogram-flavor
                   (spectrogram-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :spectrogram-size 320.
  :analysis-rate 383.
  :filter-type :hamming
  :filter-spec (:bandwidth 300.0)
  :number-of-points 256.
  :white-value -96.                              ;-36
  :black-value -80.)                             ;-20

(define-attribute "Versatec Spectrogram" stretched-fft-spectrogram-flavor
                   (spectrogram-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :spectrogram-size 840.
  :analysis-rate 1000.
  :filter-type :hamming
  :filter-spec (:bandwidth 400.0)
  :number-of-points 128.
  :white-value -100.                             ;-36
  :black-value -75.)                             ;-20

(define-attribute "New Narrow-Band Spectrogram" stretched-fft-spectrogram-flavor
                   (spectrogram-attribute-window) nil
  :waveform-attribute-name "Original Waveform"
  :spectrogram-size 840.                                 ;try 840
  :analysis-rate 1000.                                   ;try 1000
  :filter-type :hamming
  :filter-spec (:bandwidth 76.0)                        ;try 400
  :number-of-points 128.
  :white-value -100.                             ;-36     try -102
  :black-value -75.)

(define-attribute "Phonetic Transcription" hand-transcription-flavor
                   (transcription-attribute-window)
                   (("New Phonetic Transcription"
                     (:values) token-attribute-window :x-scale 383.0
                     :string-font fonts:ipa12))
  :untranscribed-string "<ˆntr@nskrYbd>"
  :string-font fonts:ipa12)

(define-attribute "Orthographic Transcription" hand-transcription-flavor
                   (transcription-attribute-window)
                   (("New Orthographic Transcription"
                     (:values) token-attribute-window :x-scale 383.0))
  :token-separator #\space
  :untranscribed-string "<untranscribed>"
  :string-font fonts:hl12b)


(define-attribute "First Formant" formant-flavor
                   (sampled-attribute-window) nil
  :index 1
  :indexed-attribute-name "Formants")

(define-attribute "Second Formant" formant-flavor
                   (sampled-attribute-window) nil
  :index 2
  :indexed-attribute-name "Formants")

(define-attribute "Third Formant" formant-flavor
                   (sampled-attribute-window) nil
  :index 3
  :indexed-attribute-name "Formants")

(define-attribute "Fourth Formant" formant-flavor
                  (sampled-attribute-window) nil
  :index 4
  :indexed-attribute-name "Formants")

 #|
(define-attribute "Fundamental Frequency" pitch-flavor
                  nil (sampled-attribute-window)
  :voicing-attribute-name "Voicing")

(define-attribute "Voicing" voicing-flavor
                  nil (sampled-attribute-window)
  :analysis-rate 100.
  :waveform-attribute-name "Original Waveform")
 |#
```

Appendix B: Program Listing

```
;;; -*- Mode: LISP; Base: 10; Syntax: Zetalisp -*-
;;;
;;;  This file contains the necessary function to compute the dynamic
;;;  time warp array, given feature arrays from the template and the utterance.
;;;
;;;
;;;
;;;
;;;
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "TIMEWARP"
;;;
;;;  This function receives a pair of arrays, determines there dimensionality
;;;  and calls TIMEWARP-1D or TIMEWARP-2d accordingly
;;;
(defun timewarp (arrayM arrayN)
  (cond ((= 1 (array-#-dims arrayM)) (timewarp-1d arrayM arrayN))
        ((= 2 (array-#-dims arrayM)) (timewarp-2d arrayM arrayN))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "TIMEWARP-2D"
;;;
;;;  This function will compute the Dynamic Time Warp array given the arrays
;;;  arrayM and arrayN.  ArrayM is a x-by-M array, and arrayN is a x-by-N array.
;;;  x must be the same for both arrayM and arrayN.  This function is meant for
;;;  those spire att's that return 2-D arrays such as the "Wide-Band Spectrum" and
;;;  and "Formants".  The distance measure used is Minkowski 1 or 2:
;;;
;;;   distance = [(a0-b0]^2 + (a1-b1)^2 + ... (aM-bN)^2]^(1/2) or
;;;
;;;   distance = abs[a0-b0] + abs[a1-b1] ...
;;;
;;;  Input:   arrayM, arrayN
;;;  Output:  A M-by-N DTW array
;;;
(defun timewarp-2d (arrayM arrayN)
  (let* ((M (- (array-dimension-n 2 arrayM) 6))
         (N (- (array-dimension-n 2 arrayN) 6))
         (length (cond((= (array-dimension-n 1 arrayM) 5)
                        2)
                       ((= (array-dimension-n 1 arrayM) 16)
                        16)
                       ((= (array-dimension-n 1 arrayM) 19)
                        19)
                       (t
                        (princ "Timewarp ERROR.  Hit Control-Abort")
                        (do ((x 0))
                            ((= x 1))))))
         (start (cond((= (array-dimension-n 1 arrayM) 5)
                       1)
                      ((= (array-dimension-n 1 arrayM) 16)
                       0)
                      ((= (array-dimension-n 1 arrayM) 19)
                       0)))
         (distance 0)
         (result-array (make-array (list M N))))
    (loop for n-index from 0 below N do
      (loop for m-index from 0 below M do
        (setq Distance 0.0)
        (loop for v-index from start below (+ start length) do
          (setq distance (+ distance (abs (- (aref arrayM v-index m-index)
                                             (aref arrayN v-index n-index))))))
        (aset distance result-array m-index n-index)))
    result-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "TIMEWARP-1D"
;;;
```

```
;;;  This function computes a Dynamic Time Warp array given vectorM and vectorN.  In
;;;  this case the distance = abs[a-b] for each a and b in vectorM and vectorN.
;;;
;;;  Input:  vectorM, vectorN
;;;  Output: a M-by-N DTW array
;;;
(defun timewarp-1d (vectorM vectorN)
  (let* ((M (- (array-dimension-n 1 vectorM) 5))
         (N (- (array-dimension-n 1 vectorN) 5))
         (return-array (make-array (list M N))))
    (do* ((n-index 0 (1+ n-index)))
         ((= n-index N))
      (do* ((m-index 0 (1+ m-index)))
           ((= m-index M))
        (aset (abs (- (aref vectorM m-index) (aref vectorN n-index)))
              return-array m-index n-index)))
    return-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "PRINT-DTW"
;;;
;;;  This function will show the Dynamic Time Warp array.  This function is
;;;  really intended for testing/debugging purposes.
;;;  This function will print a section of a 2-D array beginning at (a,b).
;;;
(defun print-dtw (array a b)
  (clearscreen)
  (do ((i (+ a 40) (1- i)))
      ((= i (1- a)))
    (do (( j b (1+ j)))
        ((= j (+ b 14)))
      (princ (format nil "~2,1,8,' $" (aref array i j))))
    (terpri)))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "DRAWBORDER"
;;;
;;; This function will draw a border on the selected window
;;;
(defun drawborder (x1 y1 x2 y2)
  (send tv:selected-window :draw-line x1 y1 x2 y1)
  (send tv:selected-window :draw-line x1 y1 x1 y2)
  (send tv:selected-window :draw-line x1 y2 x2 y2)
  (send tv:selected-window :draw-line x2 y2 x2 y1))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "PLOT-COMPOSITE-DTW"
;;;
(defun plot-composite-dtw (array-list threshold &OPTIONAL search-list tempath uttpath title)
  (let* ((radius 0)
         (total-M (apply '+ (mapcar 'array-dimension-n (circular-list 1) array-list)))
         (total-N (array-dimension-n 2 (car array-list)))
         (x1 400)
         (y1 45)
         (y2 (+ y1 (min 595 total-M)))
         (yrange (- y2 y1))
         (x2 (fix (+ x1 (* 2 (* total-N (// (float yrange) total-M))))))
         (xrange (- x2 x1)))
    (clearscreen)
    (drawborder  x1 y1 x2 y2)
    (send tv:selected-window :draw-string title x2 (- y1 4) 0 (- y1 4) nil fonts:tr12b)
    (do* ((a-list array-list (cdr a-list))
          (k 0 (1+ k))
          (array (car a-list) (car a-list))
          (bottom y2 (- bottom current-yrange))
          (v-word (car *vocabulary*) (cond ((not (null a-list))
                                            (nth k *vocabulary*))
                                           (t nil)))
          (current-M (array-dimension-n 1 array) (cond ((not (null a-list))
                                                        (array-dimension-n 1 array))
```

```
                                                      (t 0)))
        (current-N (array-dimension-n 2 array) (cond ((not (null a-list))
                                                       (array-dimension-n 2 array))
                                                      (t 0)))
        (current-yrange (* yrange (// (float current-M) total-M))
                        (* yrange (// (float current-M) total-M))))
       ((null a-list))
     (send tv:selected-window :draw-line (- x1 70) (round bottom) (+ 10 x2) (round bottom))
     (display-waveform-rot (- x1 50) (round (- bottom current-yrange))
                           (1- x1) (round bottom)
                           (string-append tempath v-word ".utt"))
     (send tv:selected-window :draw-string
           (nth k *vo-list*)
           (- x1 60)
           (+ (round (- bottom (// current-yrange 2))) 6)
           0
           (+ (round (- bottom (// current-yrange 2))) 6)
           nil
           fonts:bigfnt)
     (do ((m-index 0 (1+ m-index)))
         ((= m-index current-M))
       (do ((n-index 0 (1+ n-index)))
           ((= n-index current-N))
         (setq radius (cond ((< (aref array m-index n-index) threshold) 0)
                            (t -1)))
         (cond ((= radius -1) nil)
               (t (send tv:selected-window :draw-point
                        (round (+ x1 (* n-index (// (float xrange) current-N))))
                        (round (- bottom (* m-index (// current-yrange current-M))))
                        ))))))
   (do ((n-index 0 (+ n-index 10)))
       ((> n-index total-N))
     (send tv:selected-window :draw-line
           (round (+ x1 (* n-index (// (float xrange) total-N))))
           (- y2 4)
           (round (+ x1 (* n-index (// (float xrange) total-N))))
           (+ 5 y2)))
   (do ((m-index 0 (+ m-index 10)))
       ((> m-index total-M))
     (send tv:selected-window :draw-line
           (- x1 5)
           (round (- y2 (* m-index (// (float yrange) total-M))))
           (+ x1 5)
           (round (- y2 (* m-index (// (float yrange) total-M))))))
   (cond ((not search-list))
         (t (loop for word in search-list do
              (send tv:selected-window :draw-line
                    (round (+ x1 (* (nth 1 word) (// (float xrange) total-N))))
                    y1
                    (round (+ x1 (* (nth 1 word) (// (float xrange) total-N))))
                    (+ y2 70))
              (send tv:selected-window :draw-string
                    (nth  (car word) *vo-list*)
                    (- (round (+ x1
                                 (* (// (+ (nth 1 word) (nth 2 word)) 2)
                                    (// (float xrange) total-N)))) 10)
                    (+ 70 y2)
                    (- (round (+ x1
                                 (* (// (+ (nth 1 word) (nth 2 word)) 2)
                                    (// (float xrange) total-N)))) 10)
                    (+ 70 y2)
                    nil
                    fonts:bigfnt))
            (send tv:selected-window :draw-line
                  x2 y2 x2 (+ y2 70))))
   (cond ((not uttpath))
         (t (display-waveform x1 (1+ y2) x2 (+ y2 50) uttpath)))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "PLOT-DTW"
;;;
(defun plot-dtw (array path1 path2 threshold )
  (let* ((radius 0)
         (M (array-dimension-n 1 array))
         (N (array-dimension-n 2 array))
         (x1 200)
         (x2 900)
         (squish-factor (// (- x2 x1) (float N)))
         (y1 150)
         (y2 (fix (+ (* M squish-factor) y1)))
         (note (prompt-and-read :string "Distance Array Name? ")))
    (clearscreen)
    (display-waveform x1 (1+ y2)  x2 (+ y2 100) path2)
    (display-waveform-rot (- x1 100) y1 (1- x1) y2 path1)
    (distribution x1 (+ y2 120) x2 (+ y2 320) (list array) 50)
    (drawborder x1 y1 x2 y2)
    (do ((m-index 0 (1+ m-index)))
        ((= m-index M))
      (do ((n-index 0 (1+ n-index)))
          ((= n-index N))
          (setq radius (cond ((< (aref array m-index n-index) threshold) 2)
                             ((< (aref array m-index n-index) (* 1.5 threshold)) 1)
                             ((< (aref array m-index n-index) (* 1.75 threshold)) 0)
                             (t -1)))
          (cond ((= radius -1) nil)
                (t (send tv:selected-window :draw-filled-in-circle
                         (fix (+ x1 (* n-index (// (- x2 x1) (float N)))))
                         (fix (- y2 (* m-index (// (- y2 y1) (float M)))))
                         radius)))))
    (send tv:selected-window :draw-string note x2 (- y1 5) x1 (- y1 5) nil fonts:tr12b)
    (send tv:selected-window :draw-string
          "Test Pattern"
          (- x2 10) (+ y2 25) 0 (+ y2 25) nil fonts:tr12b)
    (send tv:selected-window :draw-string
          "Template"
          (- x1 15) (- y1 5) 0 (- y1 5) nil fonts:tr12b)
    (do ((n-index 0 (+ n-index 10)))
        ((> n-index N))
      (send tv:selected-window :draw-line
            (fix (+ x1 (* n-index (// (- x2 x1) (float N)))))
            (- y2 5)
            (fix (+ x1 (* n-index (// (- x2 x1) (float N)))))
            (+ y2 6)))
    (do ((m-index 0 (+ m-index 10)))
        ((> m-index M))
      (send tv:selected-window :draw-line
            (- x1 6)
            (fix (- y2 (* m-index (// (- y2 y1) (float M)))))
            (+ x1 5)
            (fix (- y2 (* m-index (// (- y2 y1) (float M)))))))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "COMBINE-DTW"
;;;
;;;  This function will weight and combine two or more dtw-arrays.
;;;
;;;  Input:  dtwlist => a list of dtw's to combine
;;;          weightlist => list of weight factors to apply to dtwlist
;;;
;;;  Output: new dtw-array
;;;
;;;
(defun combine-dtw (dtwlist weightlist)
  (let* ((m-dimension (array-dimension-n 1 (car dtwlist)))
         (n-dimension (array-dimension-n 2 (car dtwlist)))
         (return-dtw (make-array (list m-dimension n-dimension)))
         (sum 0))
    (do ((m 0 (1+ m)))
        ((= m m-dimension))
      (do ((n 0 (1+ n)))
          ((= n n-dimension))
        (setq sum 0.0)
        (do* ((dtw-index dtwlist (cdr dtw-index))
              (dtw-array (car dtw-index) (car dtw-index))
              (weight-index weightlist (cdr weight-index))
              (weight-value (car weight-index) (car weight-index)))
             ((null dtw-index))
           (setq sum (+ sum (* weight-index (aref dtw-array m n)))))
         (aset sum return-dtw m n)))
    return-dtw))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "MAKE-DTW"
;;;
;;;  This routine computes a Dynamic Time Warp Array give the pathnames of
;;;  two utterances and a Spire attribute name (ex. "Formants").
;;;  The order in which the pathnames are passed is significant, ie.,
;;;  when plotted the first pathname will run along the vertical axis, and
;;;  the second pathname will run across the horizontal axis.  When matching
;;;  individual word utterances against continous speech utterances, it is best
;;;  to pass the individual word pathname first.
;;;
;;;  Input:    pathname1, pathname2, spire attribute
;;;  Example Call: (make-dtw ">dawson>three>" ">dawson>phone-no" "Wide-Band Spectrum")
;;;  Returns:  a two dimensional array.  The number of columns (width) is
;;;  proportional to the length of pathname2.  The number of rows (height) is
;;;  proprtional to the length of pathname1.
;;;
(defun make-dtw (path1 path2 att)
  (let* ((a (cond ((equal att "Wide-Band Spectrum") (column-normalize-array
                                                     (frequency-compress-lfe
                                                       (compute-att path1 att))))
                  ((equal att "LPC Spectrum") (column-normalize-array
                                               (frequency-compress-lfe
                                                 (compute-att path1 att))))
                  ((equal att "Narrow-Band Spectrum") (column-normalize-array
                                                       (frequency-compress-lfe
                                                         (compute-att path1 att))))
                  ((equal att "Formants") (regionize
                                           (median-filter
                                             (compute-att path1 att))))
                  ((equal att "zero crossing rate") (vector-energy-normalize
                                                     (compute-att path1 att)))
                  (t
                   (compute-att path1 att))))
         (b (cond ((equal att "Wide-Band Spectrum") (column-normalize-array
                                                     (frequency-compress-lfe
                                                       (compute-att path2 att))))
                  ((equal att "LPC Spectrum") (column-normalize-array
                                               (frequency-compress-lfe
                                                 (compute-att path2 att))))
                  ((equal att "Narrow-Band Spectrum") (column-normalize-array
                                                       (frequency-compress-lfe
                                                         (compute-att path2 att))))
                  ((equal att "Formants") (regionize
                                           (median-filter
                                             (compute-att path2 att))))
                  ((equal att "zero crossing rate") (vector-energy-normalize
                                                     (compute-att path2 att)))
                  (t
                   (compute-att path2 att))))
         (return-array (timewarp a b)))
     return-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "NEW-READY-DTW-LPC-FORMANTS"
;;;
(defun new-ready-dtw-lpc-formants (template utterance)
  (let* ((dtw-list (list (timewarp (car template) (car utterance))
                         (timewarp (cadr template) (cadr utterance))))
         (m-dimension (array-dimension-n 1 (car dtw-list)))
         (n-dimension (array-dimension-n 2 (car dtw-list)))
         (return-dtw (make-array (array-dimensions (car dtw-list)) :type 'art-16b)))
    (loop for m from 0 below m-dimension do
      (loop for n from 0 below n-dimension do
        (let ((frfrt (aref (caddr template) m))
              (frfru (aref (caddr utterance) n))
              (t-region (aref (cadr template) m))
              (u-region (aref (cadr utterance) n))

              (distance (* 1000 (car *weight-list*) (aref (car dtw-list) m n))))
          (cond ((or (> frfrt 1500)
                     (> frfru 1500)
                     (= t-region 0)
                     (not (= t-region u-region)))
                 (aset (fix distance) return-dtw m n))
                (t (aset (fix (* 0.4 distance)) return-dtw m n))))))
    return-dtw))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "READY-DTW-LPC-FORMANTS-FF"
;;;
(defun ready-dtw-lpc-formants-ff (template utterance)
  (let* ((dtw-list (list (timewarp (car template) (car utterance))
                         (timewarp (cadr template) (cadr utterance))))
         (m-dimension (array-dimension-n 1 (car dtw-list)))
         (n-dimension (array-dimension-n 2 (car dtw-list)))
         (return-dtw (make-array (array-dimensions (car dtw-list)) :type 'art-16b))
         (sum 0.0))
    (do ((m 0 (1+ m)))
        ((= m m-dimension))
      (do ((n 0 (1+ n)))
          ((= n n-dimension))
        (setq sum 0.0)
        (cond((and
                (< (aref (caddr template) m) 1700)
                (< (aref (caddr utterance) n) 1700)
                (< (aref (cadr template) 1 m) 750)
                (< (aref (cadr utterance) 1 n) 750)
                (< (aref (cadr template) 2 m) 2200)
                (< (aref (cadr utterance) 2 n) 2200))
              (setq sum (* 1000 (cadr *weight-list*) (aref (cadr dtw-list) m n))))
             (t
              (setq sum (* 1000 (car *weight-list*) (aref (car dtw-list) m n)))))
        (cond ((< (fix sum) 65535)
               (aset (fix sum) return-dtw m n))
              (t (princ "Overflow")))))
    return-dtw))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "READY-DTW"
;;;
;;;  This function computes a combined dtw from a couple lists of feature arrays
;;;  and return that combined dtw array.  It receives as input two lists of feature
;;;  arrays.  It then calls TIMEWARP to do the Dynamic Time Warps and then calls
;;;  COMBINE-DTW to average together the individual dtw's into one dtw.  Remember
;;;  the feature arrays have already been computed by PROCESS-UTTERANCE.
;;;
(defun ready-dtw (template utterance)
  (let* ((dtw-list (mapcar 'timewarp template utterance))
         (m-dimension (array-dimension-n 1 (car dtw-list)))
         (n-dimension (array-dimension-n 2 (car dtw-list)))
         (return-dtw (make-array (array-dimensions (car dtw-list)) :type 'art-16b))
         (sum 0.0))
    (do ((m 0 (1+ m)))
        ((= m m-dimension))
      (do ((n 0 (1+ n)))
          ((= n n-dimension))
        (setq sum 0.0)
        (loop for dtw in dtw-list
              for weight in *weight-list* do
          (setq sum (+ sum (* 1000 weight (aref dtw m n)))))
        (cond ((< (fix sum) 65535)
               (aset (fix sum) return-dtw m n))
              (t (princ "Overflow")))))
    return-dtw))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;
;;; "COMPUTE-COMPOSITE-DTW"
;;;
;;;  This function computes a composite dtw array between a Ready-Template
;;;  and a Ready-Utterance.  In other words dtw's (Dynamic Time Warps) are performed
;;;  between the utterance and each word of the vocabulary.  The separate dtw arrays
;;;  put in a list to form one composite array.
;;;
;;;  Input:  None, *t-set* and *ready-utterance* are used.
;;;  Output: composite dtw
;;;
;;;
(defun compute-composite-dtw ()
  (let ((result-list nil))
    (princ "Count-Down: ")
    (loop for template in *t-set*
          for count from (length *t-set*) downto 0 do
      (princ (format nil "~D-" count))
      (setq result-list (append result-list
                                (list (new-ready-dtw-lpc-formants
                                        template *ready-utterance*)))))
    (terpri)
    result-list))
;;;
;;;  "old" ==> (mapcar 'ready-dtw *t-set* (circular-list *ready-utterance*)))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "DISTRIBUTION"
;;;
;;;  This function take a composite dtw array and computes the distribution
;;;  of its values.  The second argument specifies the number of bars to
;;;  be drawn.
;;;
(defun distribution (x1 y1 x2 y2 cdtw res)
  (let* ((mean 0.0)
         (min +1e)
         (max -1e)
         (sum 0.0)
         (sum-sq 0.0)
         (vari 0.0)
         (num 0)
         (pdf (make-array res ':type art-16b ':initial-value 0))
         (width (fix (// (- x2 x1) res)))
         (space (fix (// width 3)))
         (bar (- width space))
         (pdf-max -1e)
         (title (prompt-and-read :string "Title? ")))
    (drawborder x1 y1 x2 y2)
    (send tv:selected-window :draw-string title x2 (- y1 5) 0 (- y1 5) nil fonts:tr12b)
    (loop for dtw in cdtw do
      (loop for i (fixnum) from 0 below (array-dimension-n 1 dtw) do
        (loop for j (fixnum) from 0 below (array-dimension-n 2 dtw) do
          (cond ((< (aref dtw i j) min)
                 (setq min (aref dtw i j)))
                ((> (aref dtw i j) max)
                 (setq max (aref dtw i j))))
          (setq sum (+ sum (aref dtw i j)))
          (setq sum-sq (+ sum-sq (sqr (aref dtw i j))))
          (setq num (1+ num)))))
    (setq mean (// sum num))
    (setq vari (// (- (* num sum-sq) (sqr sum)) (* num (1- num))))
    (loop for dtw in cdtw do
      (loop for i (fixnum) from 0 below (array-dimension-n 1 dtw) do
        (loop for j (fixnum) from 0 below (array-dimension-n 2 dtw) do
          (setq num (fix (* (- (aref dtw i j) min)
                            (// (1- (array-dimension-n 1 pdf)) (float (- max min))))))
          (aset (1+ (aref pdf num)) pdf num))))

    (loop for i from 0 below (array-dimension-n 1 pdf) do
      (cond ((> (aref pdf i) pdf-max)
             (setq pdf-max (aref pdf i)))))
    (loop for i from 0 below (array-dimension-n 1 pdf) do
          (send tv:selected-window :draw-rectangle
                bar
                (fix (* (aref pdf i) (// (float (- y2 y1 20)) pdf-max)))
                (+ 1 x1 space (* i width))
                (fix (- y2 (* (aref pdf i) (// (float (- y2 y1 20)) pdf-max))))))
        (send tv:selected-window :draw-string
              (format nil "Mean = ~D" mean) x1 (+ y1 15) x2 (+ y1 15) nil fonts:tr12b)
        (send tv:selected-window :draw-string
              (format nil "Min = ~D" min) x1 (+ y1 30) x2 (+ y1 30) nil fonts:tr12b)
        (send tv:selected-window :draw-string
              (format nil "Max = ~D" max) x1 (+ y1 45) x2 (+ y1 45) nil fonts:tr12b)
        (send tv:selected-window :draw-string
              (format nil "Var = ~D" vari) x1 (+ y1 60) x2 (+ y1 60) nil fonts:tr12b)
        ))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "MAKE-DTW-LIST"
;;;
;;;   This function makes repeated calls to "MAKE-DTW" and setq's each
;;;   variable-list to the corresponding item in attribute-list.
;;;
(defun make-dtw-list (pathname1 pathname2 variable-list attribute-list)
  (do* ((dtw-list variable-list (cdr dtw-list))
        (dtw-name (car dtw-list) (car dtw-list))
        (att-list attribute-list (cdr att-list))
        (att-name (car att-list) (car att-list)))
       ((null dtw-list))
    (set dtw-name (make-dtw pathname1 pathname2 att-name))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   "SCAN-DTW"
;;;
;;;   This function scans the composite Dynamic Time Warp Array and
;;;   determines what words are contained in the test utterance.  The algorythm
;;;   used is the "One-Stage Dynamic Programming Algorythm for Connected Word
;;;   Recognition" by Hermann Ney. See IEEE Transactions ASSP-32 No. 2 April 1984.
;;;
(defun scan-dtw (composite-dtw)
  (let* ((title (prompt-and-read :string "Title? "))
         (N (array-dimension-n 2 (car composite-dtw)))
         (D-list (mapcar 'make-array
                    (mapcar 'array-dimension-n (circular-list 1) composite-dtw)))
         (B-list (mapcar 'make-array
                    (mapcar 'array-dimension-n (circular-list 1) composite-dtw)))
         (from-template (make-array N :type 'art-8b))
         (from-frame (make-array N :type 'art-16b))
         (d-min)
         (save-b)
         (save-d)
         (save-temp)
         (a 1.0)
         (b 0.5)
         (return-list)
         (dummy +1e))

    ;;; STEP 1

    (terpri) (princ "Computing Accumulated Distance Array")
    (terpri) (princ "Begin Step 1 ... ")
    (loop for current-dtw in composite-dtw
          for current-ada in D-list
          for current-B in B-list do                          ; each k
      (loop for n from 0 below (array-dimension-n 1 current-dtw)  ; n := 0 .. j-1
            sum (aref current-dtw n 0) into local-sum         ; Sum for i=0
            do (aset local-sum current-ada n)
               (aset 0 current-B n)))                          ; aset initial values
    (princ "Done.")

    ;;;STEP 2

    (terpri) (princ "Begin Step 2 ... ")
    (loop for i fixnum from 1 below N do
      (setq dummy +le)
      (loop for current-dtw in composite-dtw
            for current-ada in D-list
            for current-B in B-list
            for k from 0 to (length composite-dtw) do
        (setq d-min (min (aref current-ada 0)
                         (apply 'min (mapcar 'aref D-list
                                             (mapcar '1- (mapcar 'array-dimension-n
                                                                 (circular-list 1)
                                                                 D-list))))))

        (cond ((not (= d-min (aref current-ada 0)))
               (aset (+ i 1) current-B 0)))

        (setq save-d (aref current-ada 0))
        (setq save-b (aref current-B 0))
        (aset (+ (aref current-dtw 0 i) d-min) current-ada 0)
        (loop for j fixnum from 1 below (array-dimension-n 1 current-ada) do
          (setq d-min (min (+ (* (1+ a) (aref current-dtw j i))
                              (aref current-ada j))             ;list of
                           (+ (aref current-dtw j i) save-d)      ;possible
                           (+ (* b (aref current-dtw (1- j) i))
                              (aref current-ada (1- j)))))       ;predecessors
          (setq save-temp (aref current-B j))
          (cond ((= d-min (+ (aref current-dtw j i) save-d))      ;Update
                 (aset save-b current-B j))                       ;Backpointer
                ((= d-min (+ (* b (aref current-dtw (1- j) i))
                             (aref current-ada (1- j))))          ;Array
                 (aset (aref current-B (1- j)) current-B j)))
          (setq save-d (aref current-ada j))                      ;save diagonal
          (setq save-b save-temp)                                 ;predecessor and
          (aset d-min current-ada j))                             ;and Backpointer
                                                                  ;
                                                                  ;Update "From Template"
                                                                  ;Array T[i]
                                                                  ;and "From Frame"
                                                                  ;Array F[i]
        (cond ((< (aref current-ada (1- (array-dimension-n 1 current-ada))) dummy)
               (setq dummy (aref current-ada (1- (array-dimension-n 1 current-ada))))
               (aset k from-template i)
               (aset (aref current-B (1- (array-dimension-n 1 current-B)))
                     from-frame i)))))
    (terpri) (princ "Done.")

    ;;;STEP 3

    (terpri) (princ "Begin Step 3 ...")
    (loop for i from (1- N) downto 0 do
      (princ (format nil "~D" (aref from-template i))))
    (terpri)
    (setq return-list
          (do* ((word-end (1- N) pred)
                (word (aref from-template (1- N)) (aref from-template pred))
                (pred (aref from-frame (1- N)) (aref from-frame pred))
                (answer (list word) (append (list word) answer))
                (boundry-list (list (list word pred word-end))
                              (append  (list (list word pred word-end)) boundry-list)))
               ((<= pred 1) boundry-list)))
    (plot-composite-dtw composite-dtw
                        (* *thresh* (length *weight-list*))
                        return-list
                        *tempath*
                        *uttpath*
                        title)))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "CREATE-COMPOSITE-DTW-FILE"
;;;
;;;  This function creates a Composite DTW File from *t-set* and *utterance*
;;;
(defun create-composite-dtw-file ()
  (let* ((write-path (string-append
                       "spl:>dawson>thesis>dtw>"
                       (prompt-and-read :string
                                        "Please enter CDTW name to create: "))))
    (setq *cdtw* (compute-composite-dtw))
    (dump-to-disk write-path (list *cdtw* *weight-list* *tempath* *uttpath*))
    (word-search!)))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   "LOAD-COMPOSITE-DTW-FILE"
;;;
;;;   This function prompts for a cdtw file name, loads it and setq's it to *cdtw*
;;;
(defun load-composite-dtw-file ()
  (load (string-append
          "spl:>dawson>thesis>dtw>"
          (prompt-and-read :string
                           "Please enter CDTW name to load: ")))
  (setq *cdtw* (car *data*))
  (setq *weight-list* (nth 1 *data*))
  (setq *tempath* (nth 2 *data*))
  (setq *uttpath* (nth 3 *data*))
  (word-search!))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
(defun add-template (tempname2)
  (load (string-append ">dawson>thesis>templates>" tempname2))
  (setq *t-set* (append *t-set* (car *data*)))
  (setq *tempath2* (string-append ">dawson>thesis>templates>" (cadr *data*)))
  (setq *vo-list* '("0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
                   "10" "11" "12" "13" "14" "15" "16" "17" "18" "19"))
  (setq *vocabulary* (append *vocabulary* *vocabulary*)))
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Zetalisp -*-
;;;
;;; "UTILITIES"
;;;
;;; This file contains various utilities used by WORD-SEARCH!
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "COMPUTE-ATT"
;;;
;;;    This is a function to get the att values for a
;;;    given utterance stored on disk.
;;;
;;; Calling Procedure:
;;;
;;;                    ( compute-att  utt-name att-name )
;;;
;;; Example Usage:
;;;
;;;    (setq result-array (compute-att "spl:>dawson>alpha.utt" "LPC Gain Term"))
;;;
;;; Note: result-array now contains the result of the att computation.
;;;
;;;
(defun compute-att (pathname att-name)
  (let ((return-array))
    (terpri)
    (princ "Computing ")
    (princ att-name)
    (princ "...")
    (setq return-array (spire:att-val (send (spire:utterance pathname) :find-att att-name)))
    (princ "Done.")
    return-array))
;;;
;;;
;;; Note :   This leaves the utterance described by pathname loaded until whenever.
;;;          In order to kill an utterance (unload is a better term) the following
;;;          statement will do the trick:
;;;
;;;                          (send (spire:utterance pathname) :kill)
;;;
;;;;' ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-LPC"
;;;
;;; Function to perform LPC computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-lpc (pathname)
  (let ((return-list (list (column-normalize-array
                             (frequency-compress-lfe
                               (compute-att
                                 pathname
                                 "LPC Spectrum"))))))
    (setq *weight-list* '(4.5))
    return-list))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-NBS"
;;;
;;; Function to perform NBS computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-nbs (pathname)
  (let ((return-list (list (column-normalize-array
                             (frequency-compress-lfe
                               (compute-att
                                      pathname
                                      "Narrow-Band Spectrum"))))))
    (setq *weight-list* '(5.0))
    return-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-WBS"
;;;
;;; Function to perform LPC computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-wbs (pathname)
  (let ((return-list (list (column-normalize-array
                                  (frequency-compress-lfe
                                    (compute-att
                                      pathname
                                      "Wide-Band Spectrum"))))))
    (setq *weight-list* '(4.5))
    return-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-FORMANTS"
;;;
;;; Function to perform Formant calculations on a single utterance.
;;;
(defun process-utterance-formants (pathname)
  (let ((return-list (list (median-filter
                                  (compute-att
                                    pathname
                                    "Formants")))))
    (setq *weight-list* '(.0016))
    return-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-ZCR"
;;;
(defun process-utterance-zcr (pathname)
  (let ((return-list nil))
    (terpri)
    (setq return-list (list (vector-mag-norm
                                   (compute-att
                                     pathname
                                     "Zero Crossing Rate"))))
    (setq *weight-list* '(0.02))
    return-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-LPC-FORMANTS"
;;;
;;;
;;; Function to perform family of computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-lpc-formants (pathname)
  (let ((returned-list nil))
    (setq returned-list (list (column-normalize-array
                                     (frequency-compress-lfe
                                       (compute-att
                                         pathname
                                         "LPC Spectrum")))))
    (setq returned-list (append returned-list (list (median-filter
                                                          (compute-att
                                                            pathname
                                                          "Formants")))))
   (setq *weight-list* '(2.44 0.0024))
   returned-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "PROCESS-UTTERANCE-LPC-FORMANTS-FF"
;;;
;;;  Processes utterances for LPC Spectrum, Formants, and Frication Frequency.
;;;
(defun process-utterance-lpc-formants-ff (pathname)
  (let ((returned-list nil))
    (setq returned-list (list (column-normalize-array
                                (frequency-compress-lfe
                                  (compute-att
                                    pathname
                                    "LPC Spectrum")))))
    (setq returned-list (append returned-list (list (regionize
                                                      (median-filter
                                                       (compute-att
                                                         pathname
                                                         "Formants"))))))
    (setq returned-list (append returned-list (list (compute-att
                                                       pathname
                                                       "Frication Frequency"))))
    (setq *weight-list* '(4.5 2))
    returned-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-WBS-LPC"
;;;
;;; Function to perform family of computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-wbs-lpc (pathname)
  (let ((returned-list nil))
    (setq returned-list (list (column-normalize-array
                                (frequency-compress-lfe
                                  (compute-att
                                    pathname
                                    "Wide-Band Spectrum")))))
    (setq returned-list (append returned-list (list (column-normalize-array
                                                       (frequency-compress-lfe
                                                         (compute-att
                                                           pathname
                                                           "LPC Spectrum"))))))
    (setq *weight-list* '(3.6 5.0))
    returned-list))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "PROCESS-UTTERANCE-NBS-LPC"
;;;
;;; Function to perform family of computations on a single utterance.
;;; This function makes repeated calls to "COMPUTE-ATT".
;;;
;;; Input  : Full pathname to utterance
;;; Output : List of arrays ie., computed features
;;;
(defun process-utterance-nbs-lpc (pathname)
  (let ((returned-list nil))
    (setq returned-list (list (column-normalize-array
                                (frequency-compress-lfe
                                  (compute-att
                                    pathname
                                    "Narrow-Band Spectrum")))))
    (setq returned-list (append returned-list (list (column-normalize-array
                                                       (frequency-compress-lfe
                                                         (compute-att
                                                           pathname
```

```
                                                                "LPC Spectrum"))))))
    (setq *weight-list* '(3.6 5.0))
    returned-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "COLUMN-NORMALIZE-ARRAY"
;;;
(defun column-normalize-array (array)
  (let* ((height (array-dimension-n 1 array))
         (length (array-dimension-n 2 array))
         (total-energy 0)
         (result-array (make-array (list height length) ':initial-value 0)))
    (do ((column 0 (1+ column)))
        ((= column length))
      (setq total-energy 0)
      (do ((row 0 (1+ row)))
          ((= row height))
        (setq total-energy (+ total-energy (sqr (aref array row column)))))
      (setq total-energy (sqrt total-energy))
      (do ((row 0 (1+ row)))
          ((= row height))
        (aset (// (aref array row column) (cond ((= total-energy 0) 1)
                                                (t total-energy)))
              result-array row column)))
    result-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "REGIONIZE"
;;;
;;;   This function takes as input Formants and assigns a region for
;;;   each point in time according to the first and second formants.
;;;   Each region represents a specific vowel sound.
;;;
;;;
;;;
(defun xor (alist)
  (let ((count 0))
    (loop for thing in alist do
      (cond (thing
             (setq count (1+ count)))))
    (oddp count)))



(defun intersect (seg1 seg2)
  (let* ((x11 (nth 0 seg1))
         (y11 (nth 1 seg1))
         (x12 (nth 2 seg1))
         (y12 (nth 3 seg1))
         (x21 (nth 0 seg2))
         (y21 (nth 1 seg2))
         (x22 (nth 2 seg2))
         (y22 (nth 3 seg2))
         (m1 (// (float (- y12 y11)) (- x12 x11)))
         (m2 (// (float (- y22 y21)) (- x22 x21)))
         (x (// (+ y22 (* m1 x12) (- 0 y12 (* m2 x22))) (- m1 m2)))
         (t1 (// (- x x11) (- x12 x11)))
         (t2 (// (- x x21) (- x22 x21)))
         (result (cond ((and (<= t1 1.0)
                             (>= t1 0.0)
                             (<= t2 1.0)
                             (>= t2 0.0))
                        T)
                       (T nil))))
    result))



(defun regionize (formants)
  (let* ((f1 0)
         (f2 0)
```

```
     (result (make-array (array-dimension-n 2 formants) :type 'art-8b)))
(loop for time fixnum from 0 below (array-dimension-n 2 formants) do
  (setq f1 (aref formants 1 time))
  (setq f2 (aref formants 2 time))
  ;(terpri) (princ f1) (princ ",") (princ f2) (princ "-")
  (cond ((xor (list (intersect (list f1 f2 1500 f2) '(0 1750 250 3500))
                    (intersect (list f1 f2 1500 f2) '(250 1750 450 3500))))
         ;(princ 1)
         (aset 1 result time))
         ;(aset 300 formants 1 time)
         ;(aset 2750 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(250 1750 450 3500))
                    (intersect (list f1 f2 1500 f2) '(450 1750 700 3500))))
         ;(princ 2)
         (aset 2 result time))
         ;(aset 420 formants 1 time)
         ;(aset 2300 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(450 1750 700 3500))
                    (intersect (list f1 f2 1500 f2) '(900 2500 901 3500))
                    (intersect (list f1 f2 1500 f2) '(600 1750 900 2500))))
         ;(princ 3)
         (aset 3 result time))
         ;(aset 600 formants 1 time)
         ;(aset 2200 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(600 1500 601 1750))
                    (intersect (list f1 f2 1500 f2) '(600 1750 900 2500))
                    (intersect (list f1 f2 1500 f2) '(750 1500 1200 2500))))
         ;(princ 4)
         (aset 4 result time))
         ;(aset 700 formants 1 time)
         ;(aset 1800 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(750 1500 1200 2500))
                    (intersect (list f1 f2 1500 f2) '(600 1100 601 1500))
                    (intersect (list f1 f2 1500 f2) '(650 1100 1200 1750))
                    (intersect (list f1 f2 1500 f2) '(1200 1750 1201 2500))))
         ;(princ 5)
         (aset 5 result time))
         ;(aset 800 formants 1 time)
         ;(aset 1500 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(650 950 651 1100))
                    (intersect (list f1 f2 1500 f2) '(650 1100 1200 1750))
                    (intersect (list f1 f2 1500 f2) '(800 950 1200 1100))
                    (intersect (list f1 f2 1500 f2) '(1200 1100 1201 1750))))
         ;(princ 6)
         (aset 6 result time))
         ;(aset 900 formants 1 time)
         ;(aset 1100 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(350 1300 351 1750))
                    (intersect (list f1 f2 1500 f2) '(600 1300 601 1750))))
         ;(princ 7)
         (aset 7 result time))
         ;(aset 500 formants 1 time)
         ;(aset 1500 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(400 950 401 1300))
                    (intersect (list f1 f2 1500 f2) '(600 950 601 1300))))
         ;(princ 8)
         (aset 8 result time))
         ;(aset 500 formants 1 time)
         ;(aset 1000 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(200 500 201 1300))
                    (intersect (list f1 f2 1500 f2) '(400 500 401 1300))))
         ;(princ 9)
         (aset 9 result time))
         ;(aset 300 formants 1 time)
         ;(aset 900 formants 2 time))
        ((xor (list (intersect (list f1 f2 1500 f2) '(400 500 401 950))
                    (intersect (list f1 f2 1500 f2) '(600 950 601 1100))
                    (intersect (list f1 f2 1500 f2) '(650 950 651 1100))
                    (intersect (list f1 f2 1500 f2) '(600 500 800 950))))
         ;(princ 10)
         (aset 10 result time))
         ;(aset 600 formants 1 time)
         ;(aset 800 formants 2 time))
```

```
              (t ;(princ 0)
               (aset 0 result time))))
               ;(aset 0 formants 1 time)
               ;(aset 0 formants 2 time))))
    result))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "MEDIAN-FILTER"
;;;
;;;  This function median filters the 5 by length formant array returned by SPIRE.  This
;;;  is an effort to smooth the formants values to remove the gliches when the formant
;;;  tracker loses track.  Note that the (0,i) row has all zeroe values.
;;;
(defun median-filter (array)
  (let* ((rows (array-dimension-n 1 array))
         (columns (array-dimension-n 2 array))
         (return-array (make-array (list rows columns)))
         (window-vector (make-array 11)))
    (copy-array-contents array return-array)
    (do* ((row-index 1 (1+ row-index)))
         ((= row-index rows))
      (do* ((column-index 5 (1+ column-index)))
           ((= column-index (- columns 5)))
        (do* ((window-index (- column-index 5) (1+ window-index))
              (window-vector-index 0 (1+ window-vector-index)))
             ((= window-vector-index 11))
          (aset (aref array row-index window-index) window-vector window-vector-index))
        (aset (aref (sort window-vector '<) 4) return-array row-index column-index)))
    return-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "GET-PATHNAME"
;;;
;;; Function to get a pathname from user
;;; providing prompt and default pathname.
;;;
(defun get-pathname (default)
  (fs:set-default-pathname default )
  (prompt-and-read `(:pathname :visible-default ,fs:*default-pathname-defaults*)
                   "Enter pathname => "))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "SHOW-LIST"
;;;
(defun show-list (alist)
  (loop for element in alist
        do (print element)))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;
;;; "DUMP-TO-DISK"
;;;
;;; Function to dump data to a disk file.
;;;
;;; Input  : Full Path and Filename, thing to dump
;;; Output : Writes a compiled Lisp form to disk
;;;          such that when loaded (like any ordinary lisp form)
;;;          the data is setq'd to, in this case, *data*.
;;;
(defun dump-to-disk (pathname data)
  (sys:dump-forms-to-file pathname (list `(setq *data* ',data))))
;;;
;;; Note: To read this data, (load pathname).
;;;       The global variable *data* will then contain the data.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "SQR"
;;;
```

```
;;; Function to square a number
;;;
(defun sqr (number) (* number number))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "CLEARSCREEN"
;;;
;;; Function to clear the screen.
;;; No arguments required.
;;;
(defun clearscreen ()
  (send tv:selected-window :clear-window))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "SUBLIST"
;;;
;;;  This function takes as input a list and returns a sublist of
;;;  elements i thru j.
;;;
;;;  Example: foo => (a b c d e f)
;;;           (sublist foo 1 3) => (b c d)
;;;
(defun sublist (alist i j)
  (let ((return-list (list (nth i alist))))
    (do* ((marker (1+ i) (1+ marker))
          (end (1+ j)))
         ((= marker end))
(setq return-list (append return-list (list (nth marker alist)))))
      return-list))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  "COLUMN-AVERAGE"
;;;
;;;  This function will average a subcolumn from a column of a 2-D array.
;;;  It takes as input the array, the column number, indexes i and j.  It averages
;;;  the array elements i thru j of the specified column number.
;;;
;;;  Input:  2-D arrar, column, i, j
;;;  Output: Average
;;;
(defun column-average (array column i j)
  (let ((sum 0.0)
        (end (1+ j)))
    (do ((count i (1+ count)))
        ((= count end))
      (setq sum (+ sum (aref array count column))))
    (// sum (1+ (- j i)))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;
;;;  "VECTOR-ENERGY-NORMALIZE"
;;;
;;;  Function to normalize a one dimensional array by energy
;;;
;;;  Description:
;;;              The total energy of the array is calculated
;;;              by summing the squares of all the elements and
;;;              taking the square root of that sum.
;;;              The normalized array is formed by dividing
;;;              each element of the input array by the total energy.
;;;
;;;  Input: one dimensional array
;;;  Returns: normalized version of input
;;;
(defun vector-energy-normalize (vector)
  (let ((return-array (make-array (array-length vector)))
        (total-energy 0))
  (do ((counter 0 (1+ counter))
       (endmark (array-length vector)))
```

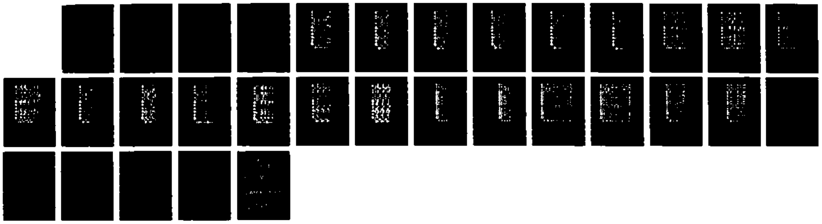
```
        ((= counter endmark))
      (setq total-energy (+ total-energy (sqr (aref vector counter)))))
    (setq total-energy (// (sqrt total-energy) (array-length return-array)))
    (do ((counter 0 (1+ counter))
         (endmark (array-length vector)))
        ((= counter endmark))
      (aset (// (aref vector counter) total-energy) return-array counter))
    return-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "VECTOR-MAGNITUDE-NORMALIZE"
;;;
;;;  This function is similar to VECTOR-ENERGY-NORMALIZE except that the
;;;  values from the input vector are simply mapped into a range of 0 to 1.
;;;  In other words, the smallest value of the input array will be mapped to zero
;;;  and the largest value mapped to one; all others will fall somewhere in
;;;  between.  This normalization technique is arises from the fact that the
;;;  VECTOR-ENERGY-NORMALIZATION technique fails for vectors of unequal length.
;;;
;;;  Input:    One dimensional array.
;;;  Returns:  Normalized version of input.
;;;
(defun vector-mag-norm (vector)
  (let* ((result-array (make-array (array-length vector)))
         (vector-max -999999.0)
         (vector-min  999999.0)
         (diff 0.0)
         (scale 0.0)
         (mapmin 0.0)
         (mapmax 1.0)
         (length (array-dimension-n 1 vector)))
        (do ((i 0 (1+ i)))
        ((= i length))
      (cond ((< (aref vector i) vector-min) (setq vector-min (aref vector i)))
            ((> (aref vector i) vector-max) (setq vector-max (aref vector i)))))
    (setq diff (- mapmin vector-min))
    (setq scale (// mapmax (+ vector-max diff)))
    (do ((i 0 (1+ i)))
        ((= i length))
      (aset (* (+ (aref vector i) diff) scale) result-array i))
    result-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "FREQUENCY-COMPRESS-LC"
;;;
;;;  This function takes an array returned by (compute-att utt-name "Wide-Band Spectrum")
;;;  which is a 256 by length array.  256 represents the frequency components
;;;  of the utterance and length is proprtional to time.  This function reduces
;;;  the frequency resolution from 256 to 16.  This is a linear compression (LC).
;;;
;;;  Input:  Two dimensional array returned by (compute-att utt-name "Wide-Band Spectrum")
;;;  Output: Compressed version of input
;;;
(defun frequency-compress-lc (array)
  (let* ((row-length (array-dimension-n 2 array))
         (return-array (make-array (list 16 row-length)))
         (block-sum 0))
    (do* ((current-column 0 (1+ current-column)))
         ((= current-column row-length))
      (do* ((current-block 0 (1+ current-block)))
           ((= current-block 16))
        (setq block-sum 0)
        (do* ((current-element (* current-block 16) (1+ current-element)))
             ((= current-element (* (1+ current-block) 16)))
          (setq block-sum (+ block-sum (aref array current-element current-column))))
        (aset (// block-sum 16) return-array current-block current-column)))
    return-array))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "FREQUENCY-COMPRESS-LFE"
```

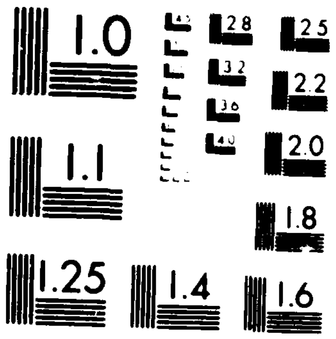
```
;;;
;;;  This function takes the array returned by (compute-att utt-name "Wide-Band Spectrum")
;;;  which is 256 by lenghth array.  The 256 discrete frequency components will be
;;;  compressed down to 16.  This compression is done with low frequency emphasise (LFE).
;;;  It is not a logrithmic compression.  Rather, the lower 132 frequency components
;;;  are are linearly compressed down to 12, and the higher 124 components are
;;;  linearly compressed down to 4.  This algorythm is written so as to make changing
;;;  the emphasise easy if desired.
;;;
;;;  Input:  Two dimensional array returned by (compute-att utt-name "Wide-Band Spectrum")
;;;  Output: Compressed version of input
;;;
(defun frequency-compress-lfe (array)
  (let* ((length (array-dimension-n 2 array))
         (return-array (make-array (list 16 length))))
    (do ((count 0 (1+ count)))
        ((= count length))
      (aset (column-average array count 0 10) return-array 0 count)
      (aset (column-average array count 11 21) return-array 1 count)
      (aset (column-average array count 22 32) return-array 2 count)
      (aset (column-average array count 33 43) return-array 3 count)
      (aset (column-average array count 44 54) return-array 4 count)
      (aset (column-average array count 55 65) return-array 5 count)
      (aset (column-average array count 66 76) return-array 6 count)
      (aset (column-average array count 77 87) return-array 7 count)
      (aset (column-average array count 88 98) return-array 8 count)
      (aset (column-average array count 99 109) return-array 9 count)
      (aset (column-average array count 110 120) return-array 10 count)
      (aset (column-average array count 121 131) return-array 11 count)
      (aset (column-average array count 132 162) return-array 12 count)
      (aset (column-average array count 163 193) return-array 13 count)
      (aset (column-average array count 194 224) return-array 14 count)
      (aset (column-average array count 225 255) return-array 15 count))
    return-array))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "MENU-FEATURE-SET"
;;;
(defun menu-feature-set ()
  (let* ((item-list '("Wide Band Spectrum"
                      "Narrow Band Spectrum"
                      "LPC Spectrum"
                      "Formants"
                      "LPC, Formants, Fr. Freq."
                      ""
                      ""
                      ""))
         (menu (tv:make-window 'tv:momentary-menu
                               ':label "Word-Search!
Select Feature Set to Use..."))
         (choice))
    (send menu ':set-item-list item-list)
    (setq choice (send menu ':choose))
  choice))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "CREATE-READY-TEMPLATE-FILE"
;;;
;;; This is the function for creating a Ready-Template file
;;; (see word-search!.doc).  This is accomplished by
;;; reading each word of the vocabulary (digits "zero" thru "nine")
;;; one by one.  Various SPIRE computations are performed, and saved
;;; to a disk file.  The user is prompted for both input and
;;; output pathnames.
;;;
;;;  Input:   None (User is prompted for read and write pathnames)
;;;  Output:  Writes Ready-Template File to Disk
;;;
(defun create-ready-template-file ()
  (let* ((read-directory (string-append
                                "spl:>dawson>thesis>templates>"
                                (prompt-and-read :string
                                                 "Please enter speaker name: ")
                                ">"))
          (read-path)
          (write-path (string-append
                        "spl:>dawson>thesis>templates>"
                        (prompt-and-read :string
                                         "Please enter Ready-Template name: ")))
          (choice nil))
     (setq *t-set* nil)
     (setq *tempath* read-directory)
     (setq choice (menu-feature-set))
     (loop for v-word in *vocabulary* do
       (setq read-path (string-append read-directory v-word ".utt"))
       (terpri)
       (princ "Processing ")
       (princ read-path)
       (princ "...")
       (setq *t-set* (append *t-set*
                             (list
                               (cond ((equal choice "Wide Band Spectrum")
                                      (process-utterance-wbs read-path))
                                     ((equal choice "Narrow Band Spectrum")
                                      (process-utterance-nbs read-path))
                                     ((equal choice "LPC Spectrum")
                                      (process-utterance-lpc read-path))
                                     ((equal choice "Formants")
                                      (process-utterance-formants read-path))
                                     ((equal choice "LPC, Formants, Fr. Freq.")
                                      (process-utterance-lpc-formants-ff read-path)))))))
;       (send (spire:utterance read-path) :kill))
     (dump-to-disk write-path (list *t-set* *tempath*)))
   (word-search!))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "CREATE-READY-UTTERANCE-FILE"
;;;
;;; This is the function for creating a Ready-Utterance file
;;; (see word-search!.doc).  This is accomplished by
;;; reading a Digitized Continuous Utterance.
;;; Various SPIRE computations are performed, and saved
;;; to a disk file.  The user is prompted for both input and
;;; output pathnames.
;;;
;;;  Input:   None (User is prompted for read and write pathnames)
;;;  Output:  Writes Ready-Template File to Disk
;;;
(defun create-ready-utterance-file ()
  (let* ((read-path
           (string-append
             "spl:>dawson>thesis>utterances>"
             (prompt-and-read :string "Name of Digitized Continuous Utterance :")
             ".utt"))
         (write-path
           (string-append
             "spl:>dawson>thesis>utterances>"
             (prompt-and-read :string "Name of Ready-Utterance : ")))
         (choice nil))
     (setq choice (menu-feature-set))
     (setq *ready-utterance*
           (cond ((equal choice "Wide Band Spectrum")
                                      (process-utterance-wbs read-path))
                                     ((equal choice "Narrow Band Spectrum")
                                      (process-utterance-nbs read-path))
                                     ((equal choice "LPC Spectrum")
                                      (process-utterance-lpc read-path))
                                     ((equal choice "Formants")
                                      (process-utterance-formants read-path))
                                     ((equal choice "LPC, Formants, Fr. Freq.")
                                      (process-utterance-lpc-formants-ff read-path))))
     (setq *uttpath* read-path)
```

AD-A188 834 SPIRE BASED SPEAKER-INDEPENDENT CONTINUOUS SPEECH RECOGNITION USING MIXED..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R G DAWSON 2/2

UNCLASSIFIED DEC 87 AFIT/GE/ENG/87D-14 F/G 12/9 NL





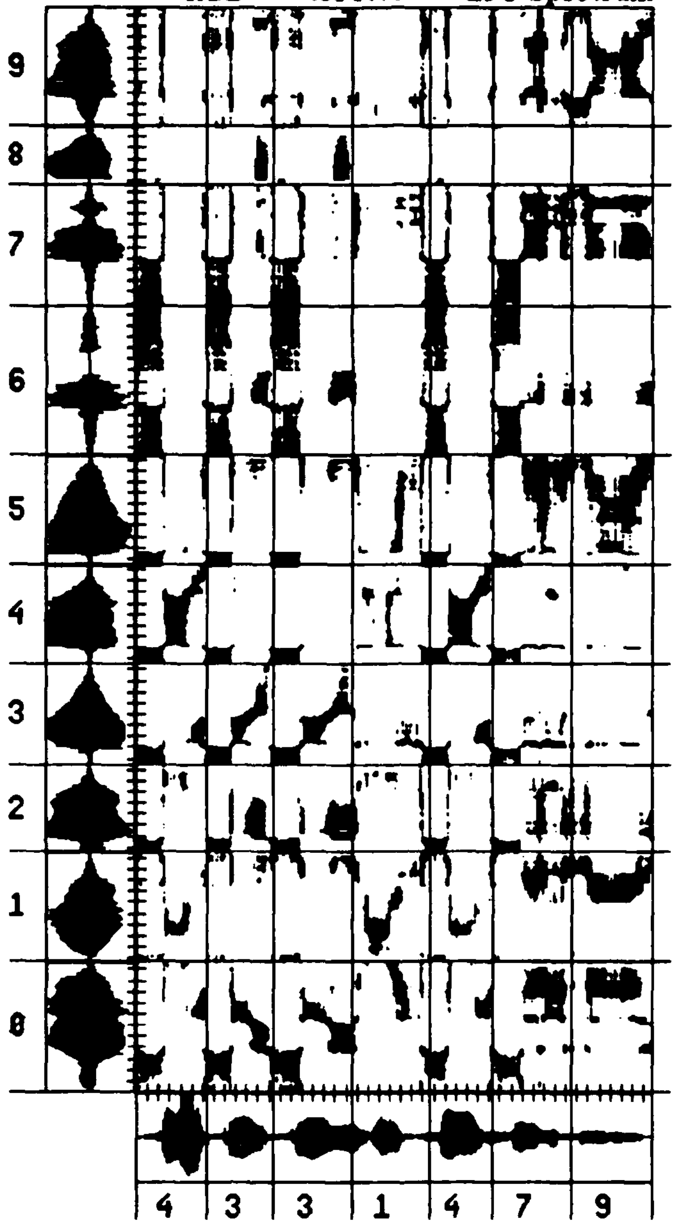
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
;    (send (spire:utterance read-path) :kill)
    (dump-to-disk write-path (list *ready-utterance* *weight-list* *uttpath*)))
  (word-search!))
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "LOAD-READY-TEMPLATE-FILE"
;;;
;;;  This function loads a Ready-Template-File and set's it to *t-set*
;;;
;;;  Input  : None, user is prompted for Ready-Template Name
;;;  Output : The global *t-set* is set to Ready-Template Name
;;;
(defun load-ready-template-file ()
  (let* ((read-path (string-append
                      "spl:>dawson>thesis>templates>"
                      (prompt-and-read :string "Name of Ready-Template : "))))
    (load read-path)
    (setq *t-set* (car *data*))
    (setq *tempath* (cadr *data*)))
  (word-search!))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "LOAD-READY-UTTERANCE-FILE"
;;;
;;;  This function loads a Ready-Utterance-File and setq's it to *ready-utterance*
;;;
;;;  Input  : None, the user is prompted for Ready-Utterance Name
;;;  Output : The global *ready-uttenance* is setq'd to Ready Utterance Name
;;;
(defun load-ready-utterance-file ()
  (let* ((read-path (string-append
                      "spl:>dawson>thesis>utterances>"
                      (prompt-and-read :string "Name of Ready-Utterance : "))))
    (load read-path)
    (setq *ready-utterance* (car *data*))
    (setq *weight-list* (cadr *data*))
    (setq *uttpath* (caddr *data*)))
  (word-search!))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "DISPLAY-WAVEFORM"
;;;
(defun display-waveform (x1 y1 x2 y2 pathname)
  (let* ((display-array (spire:att-val (send (spire:utterance pathname)
                                              :find-att "original waveform")))
         (length (array-length display-array))
         (width (- x2 x1))
         (height (- y2 y1)))
    (declare (sys:array-register display-array))
    (drawborder x1 y1 x2 y2)
    (loop for index1 fixnum from 0 to (- length 2)
          for index2 fixnum from 1 to (1- length) do
      (send tv:selected-window :draw-line
            (+ x1 (fix (* index1 (// width (float length)))))
            (+ y1 (fix (* (+ (aref display-array index1) 32767.0)
                          (// height 65535.0))))
            (+ x1 (fix (* index2 (// width (float length)))))
            (+ y1 (fix (* (+ (aref display-array index2) 32767.0)
                          (// height 65535.0))))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; "DISPLAY-WAVEFORM-ROT"
;;;
(defun display-waveform-rot (x1 y1 x2 y2 pathname)
  (let* ((display-array (spire:att-val (send (spire:utterance pathname)
                                              :find-att "original waveform")))
         (length (array-length display-array))
         (width (- x2 x1))
         (height (- y2 y1)))
```

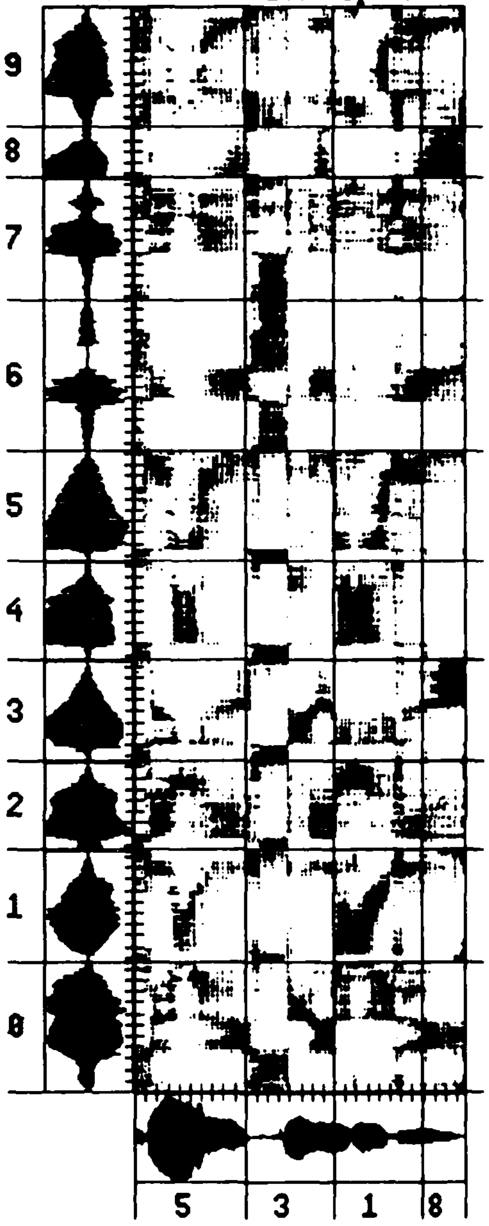
```
(declare (sys:array-register display-array))
(drawborder x1 y1 x2 y2)
(loop for index1 fixnum from 0 to (- length 2)
      for index2 fixnum from 1 to (1- length) do
  (send tv:selected-window :draw-line
        (+ x1 (fix (* (+ (aref display-array index1) 32767.0)
                      (// width 65535.0))))
        (- y2 (fix (* index1 (// height (float length)))))
        (+ x1 (fix (* (+ (aref display-array index2) 32767.0)
                      (// width 65535.0))))
        (- y2 (fix (* index2 (// height (float length)))))))))
```

Appendix C: Sample Results

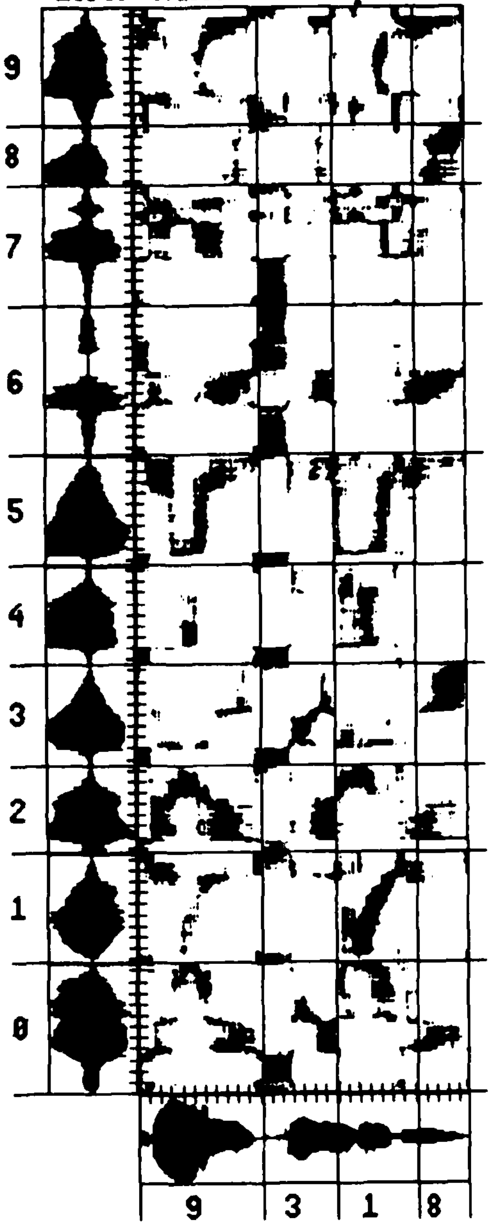
RGD -- "4331479" -- LPC Spectrum

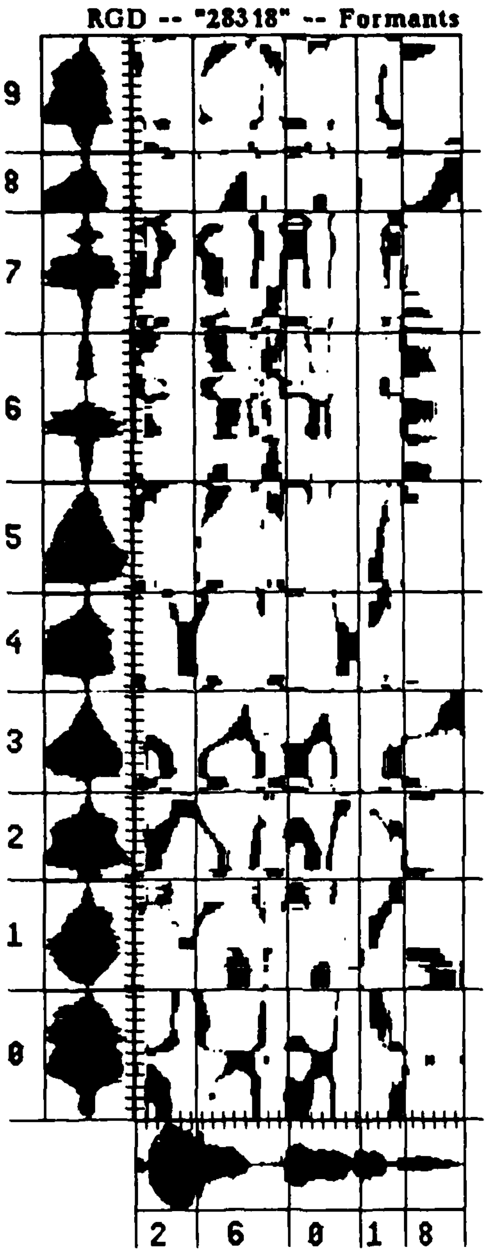


RGD -- "28318" -- Wide Band Spectrum

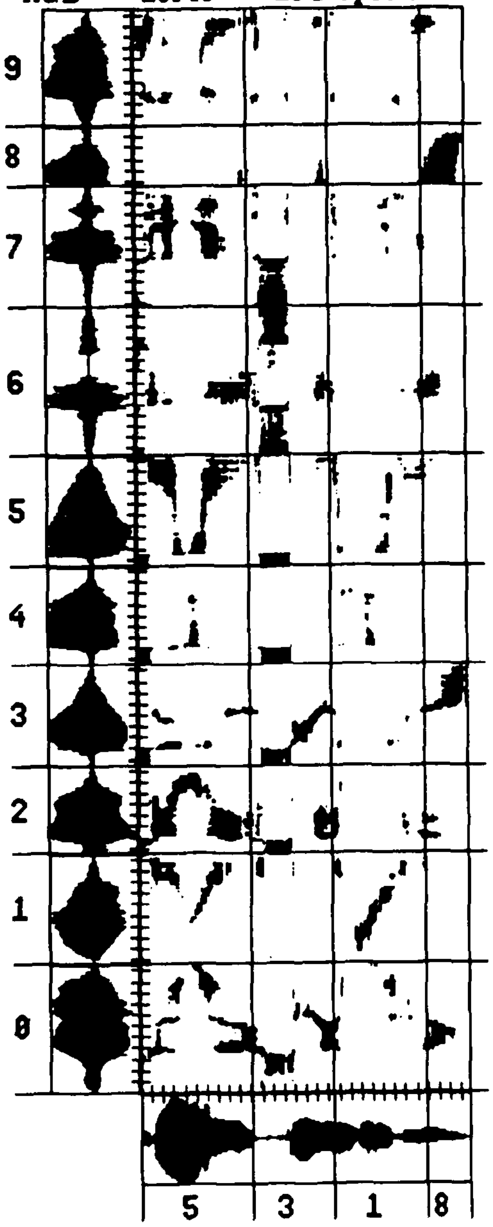


RGD -- "28318" Narrow Band Spectrum

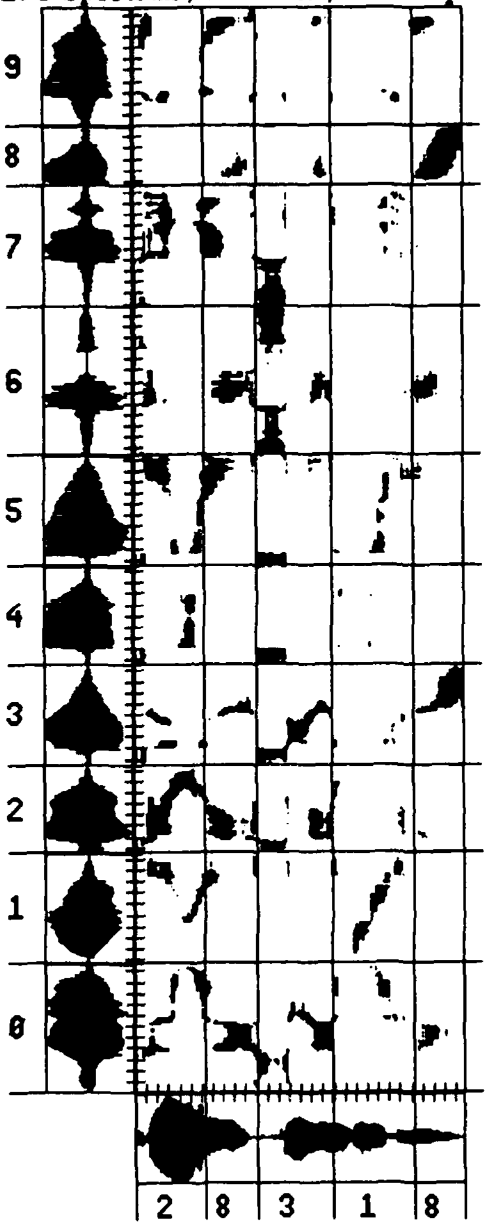



RGD -- "28318" -- Formants
9
8
7
6
5
4
3
2
1
0
2
6
0
1
8

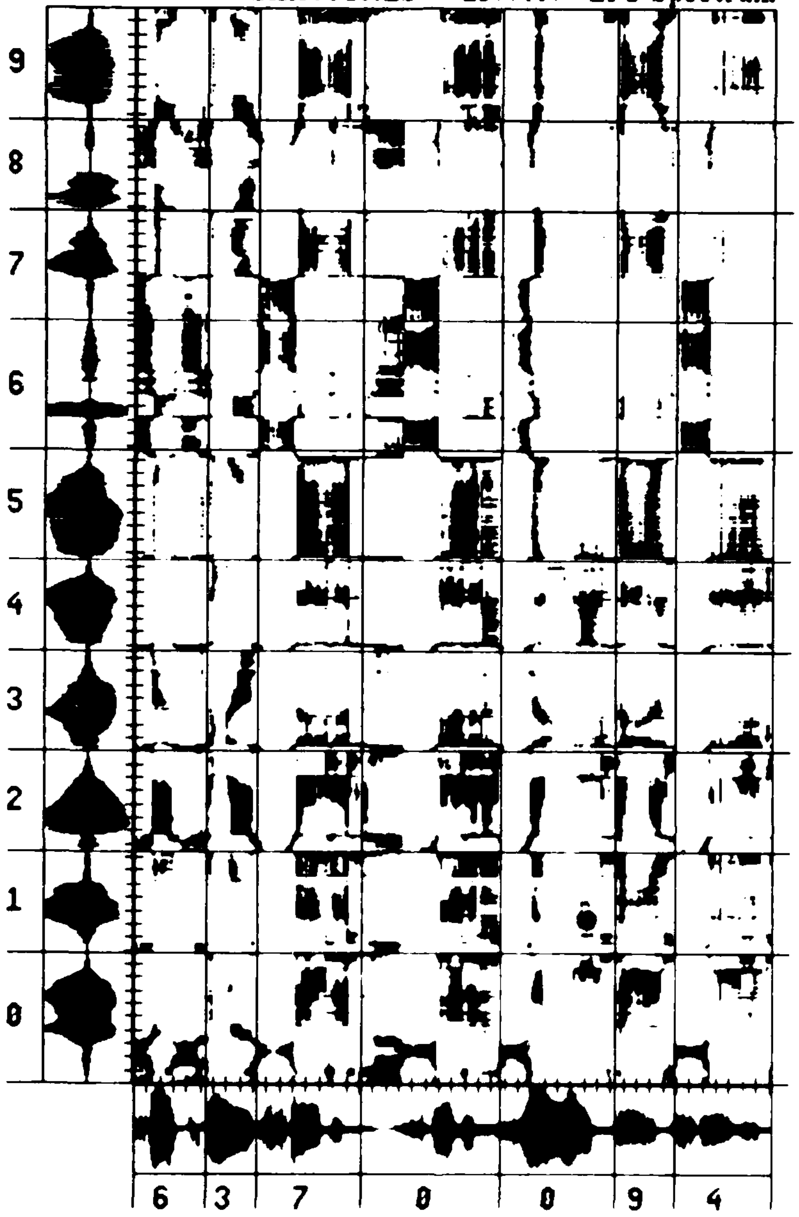
RGD -- "28318" -- LPC Spectrum



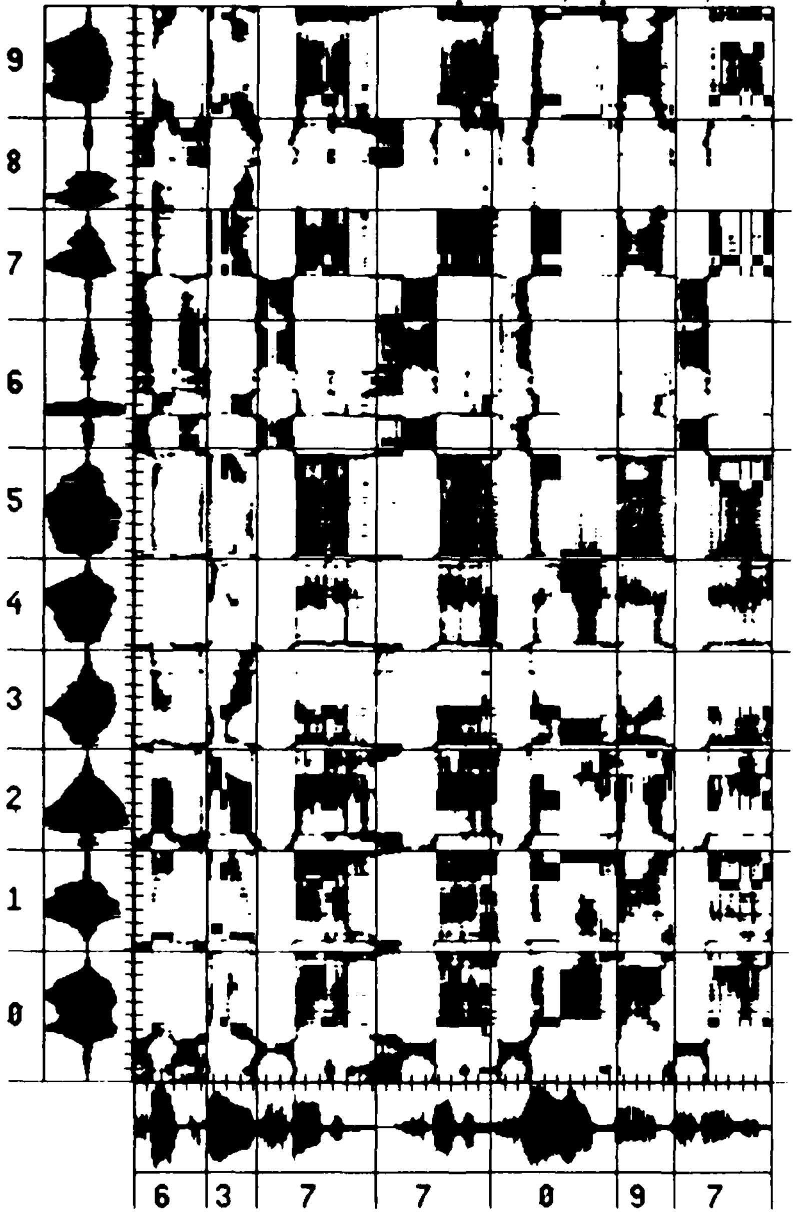
RGD - "28318" - LPC Spectrum, Formants, Fr. Freq.



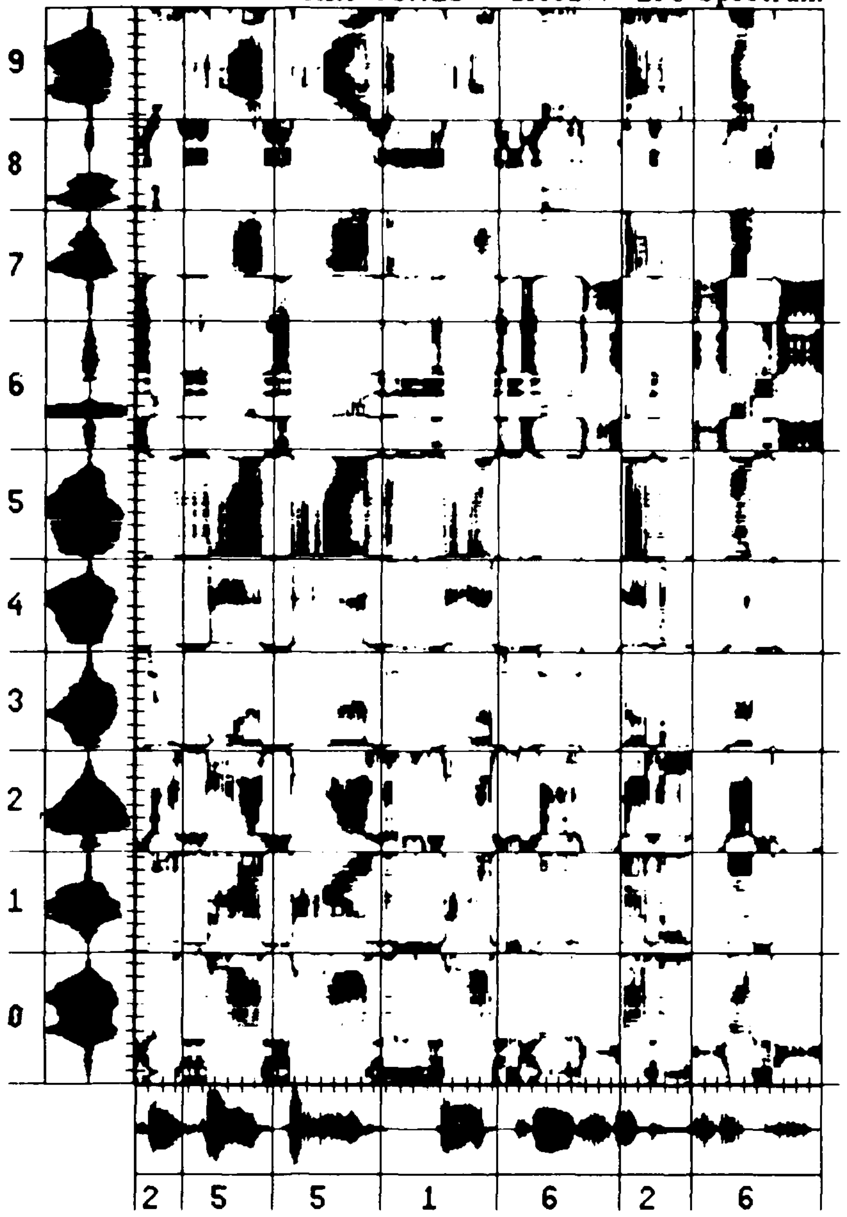
SKIP/JONES - "2377097" LPC Spectrum



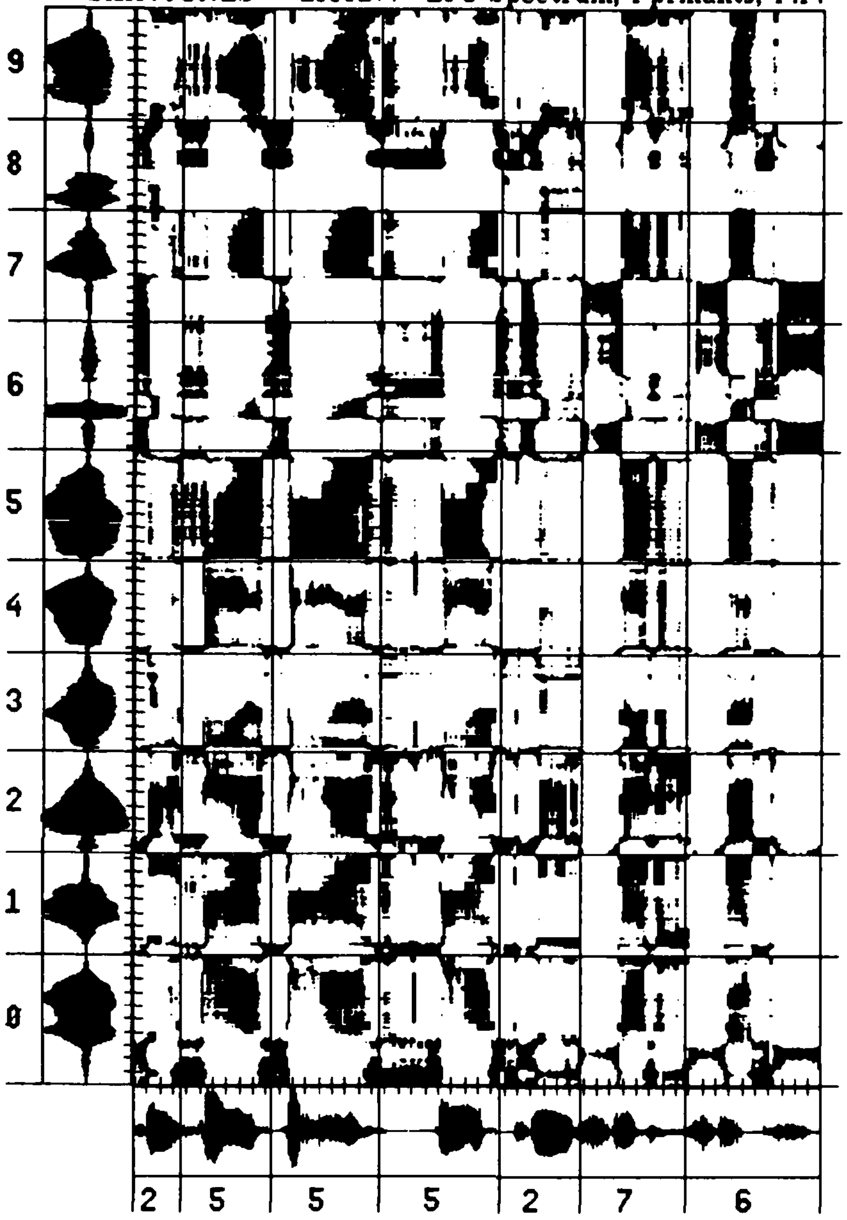
SKIP/JONES - "2377097" LPC Spectrum, Formants, F.F.



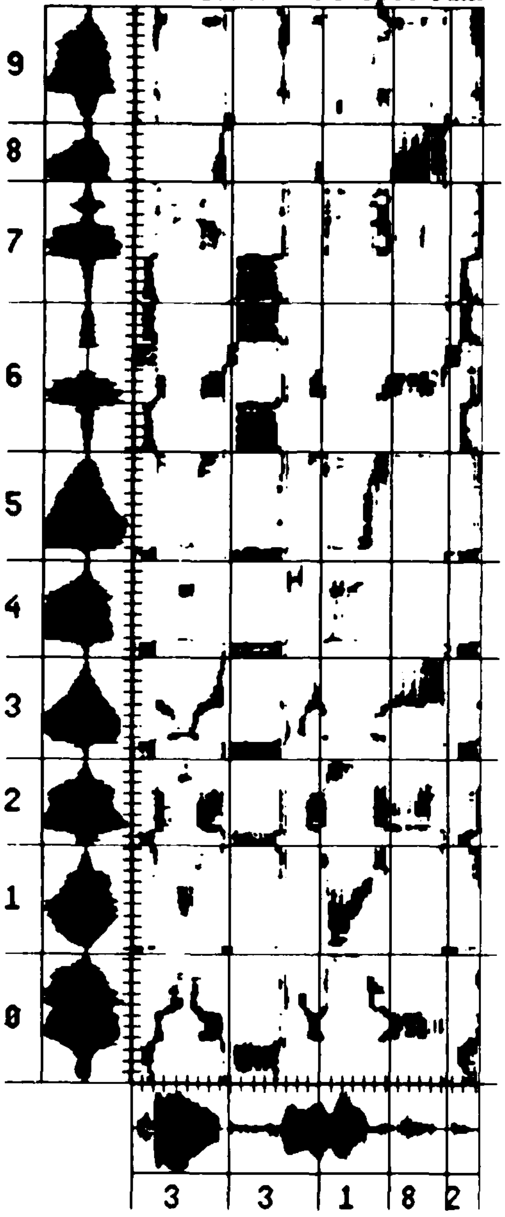
SKIP/JONES - "2555276" LPC Spectrum



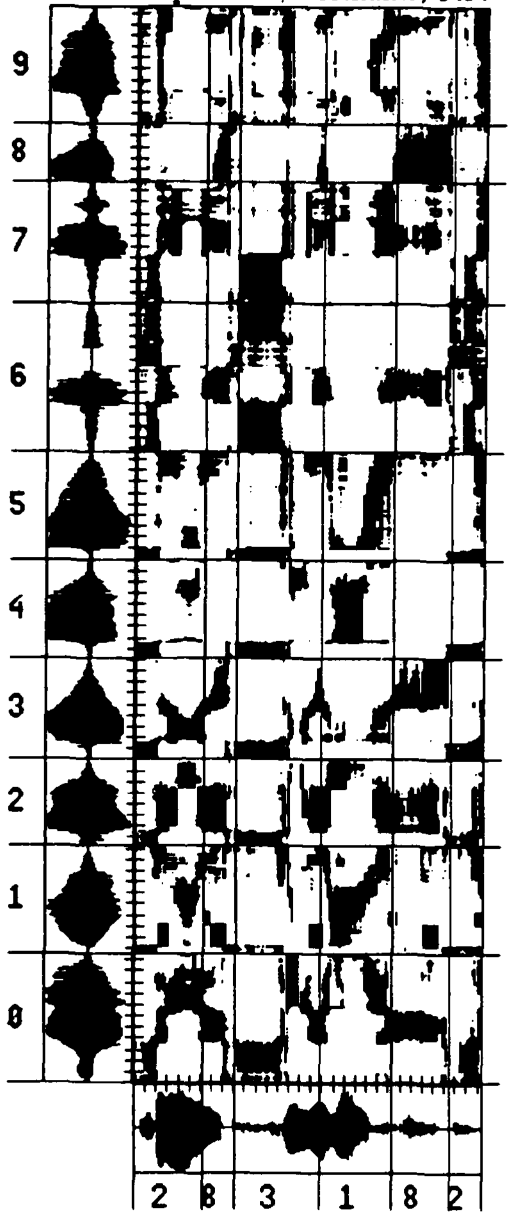
SKIP/JONES - "2555276" LPC Spectrum, Formants, F.F.



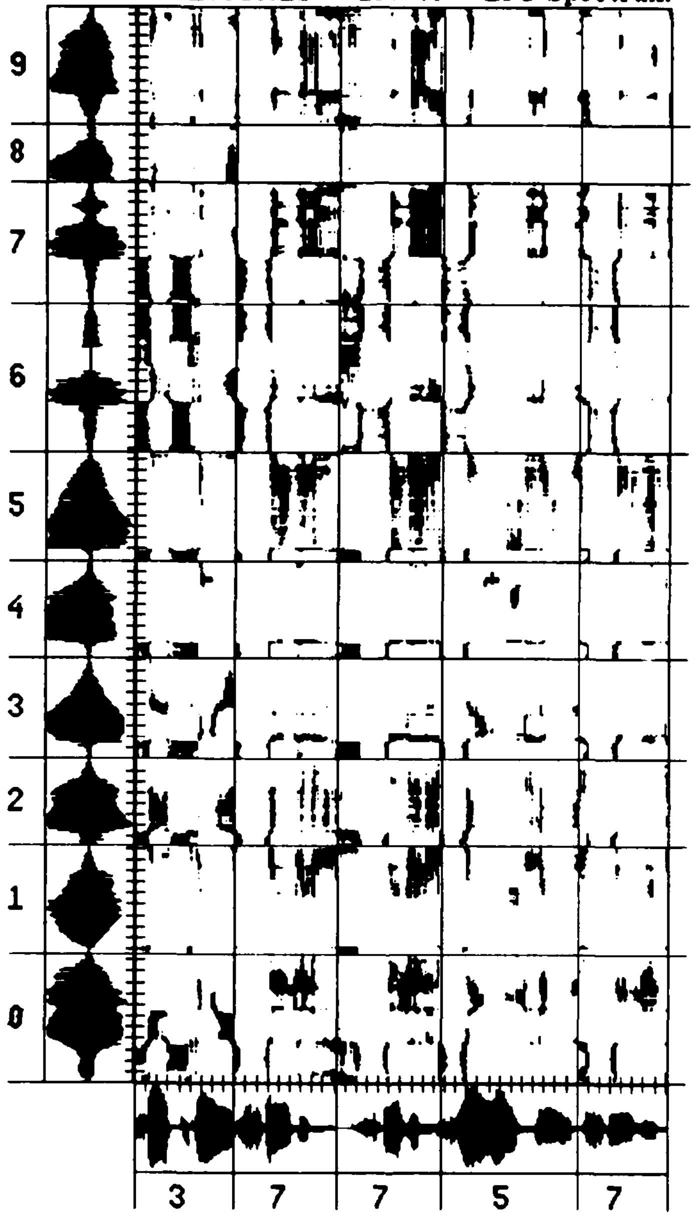
RGD/JONES - "28318" LPC Spectrum



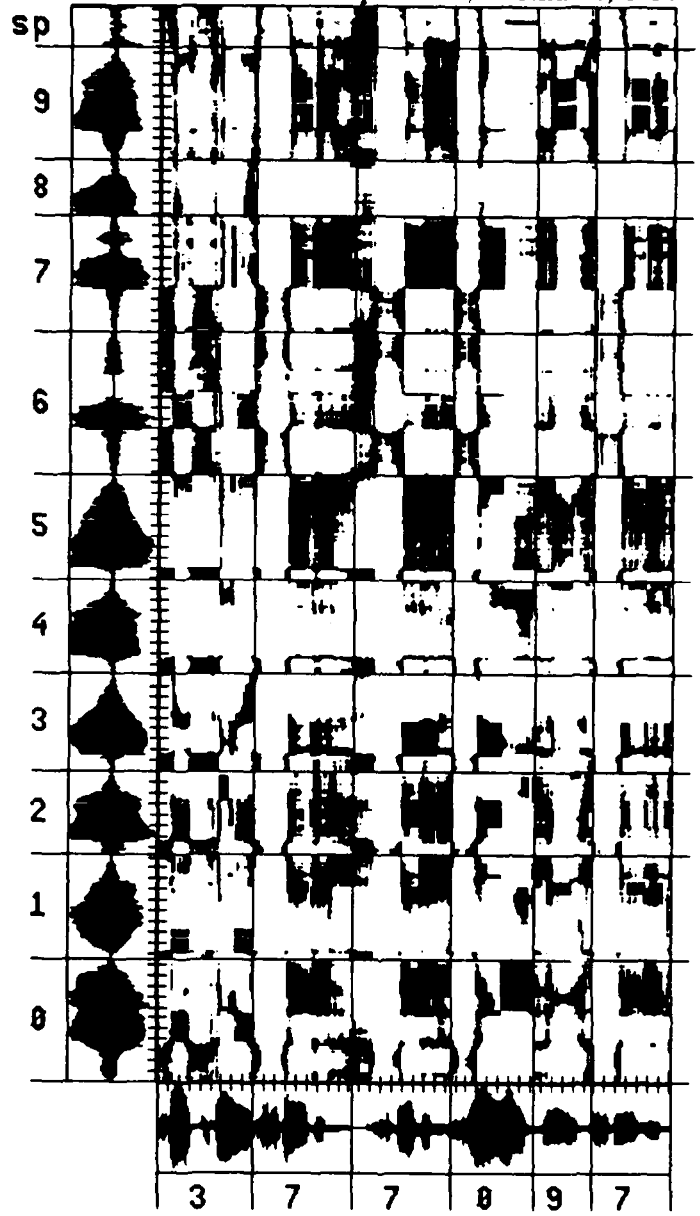
RGD/JONES - "28318" LPC Spectrum, Formants, F.F.



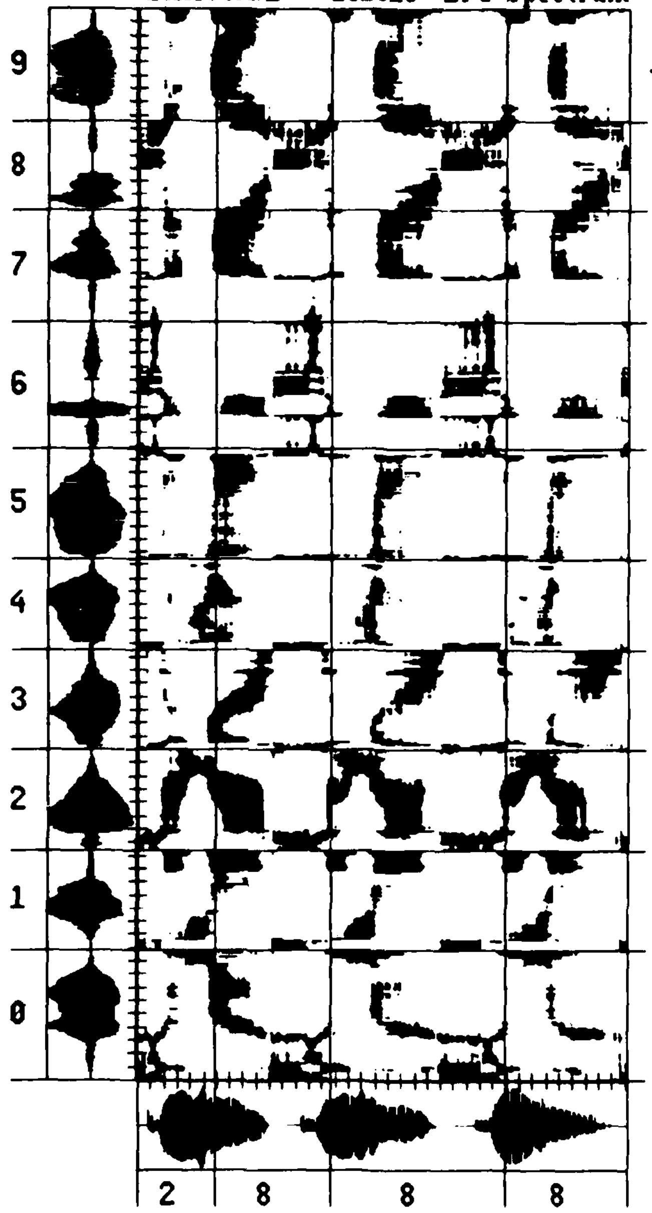
RGD/JONES - "2377097" LPC Spectrum



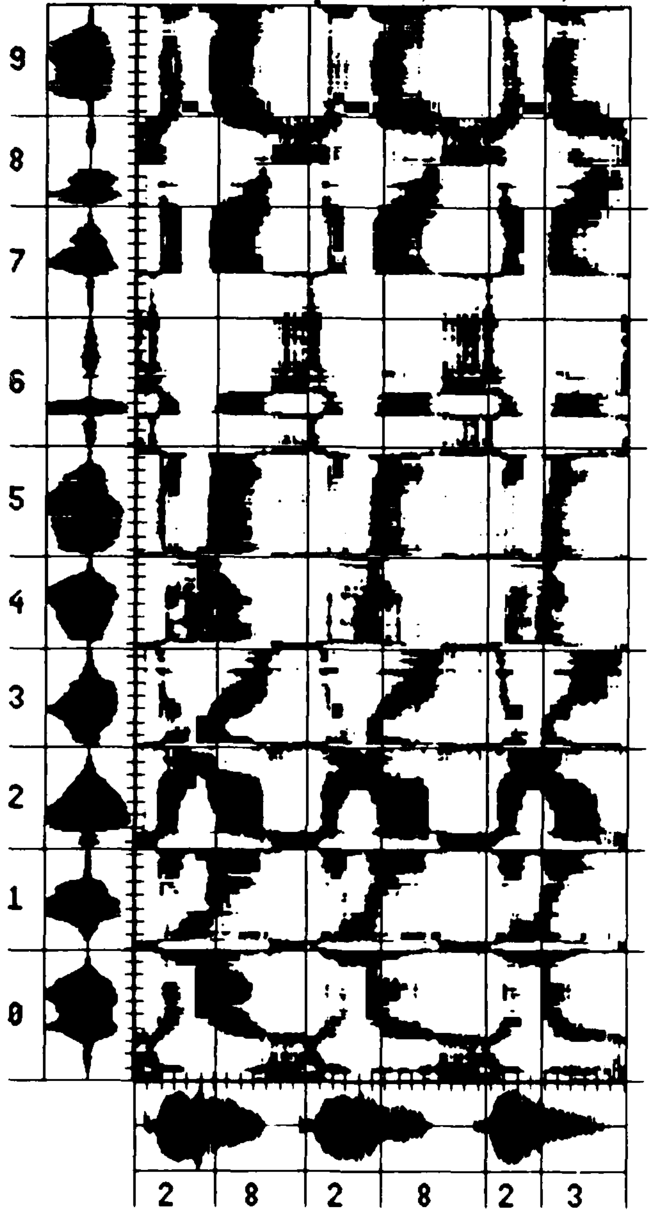
RGD/JONES - "2377097" LPC Spectrum, Formants, F.F.



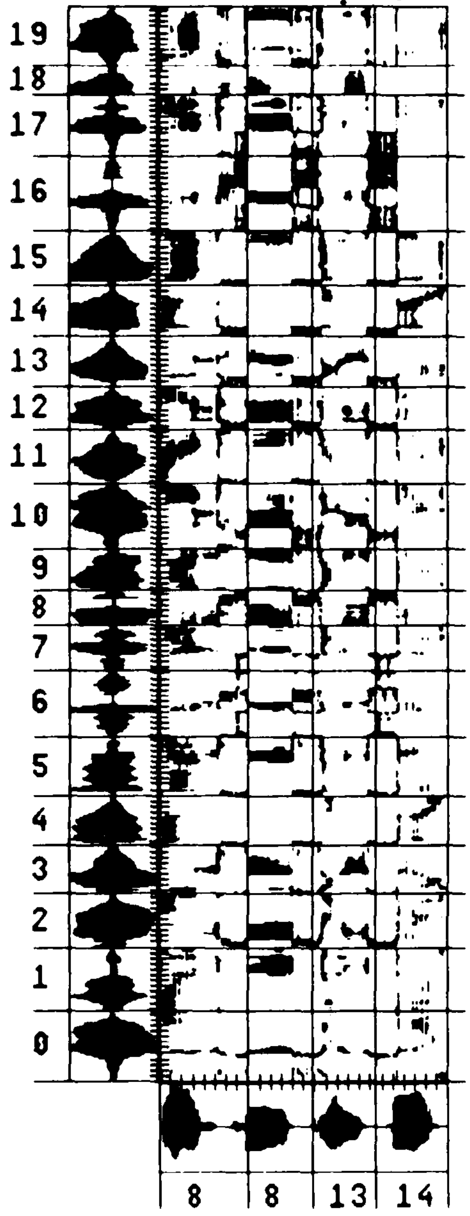
SKIP/RGD - "282828" LPC Spectrum



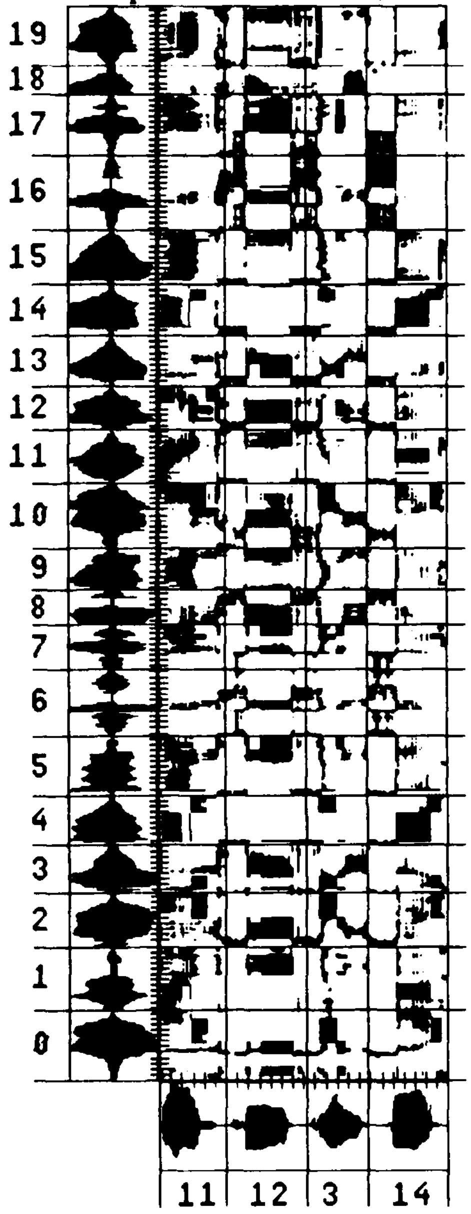
SKIP/RGD - "282828" LPC Spectrum, Formants, F.F.



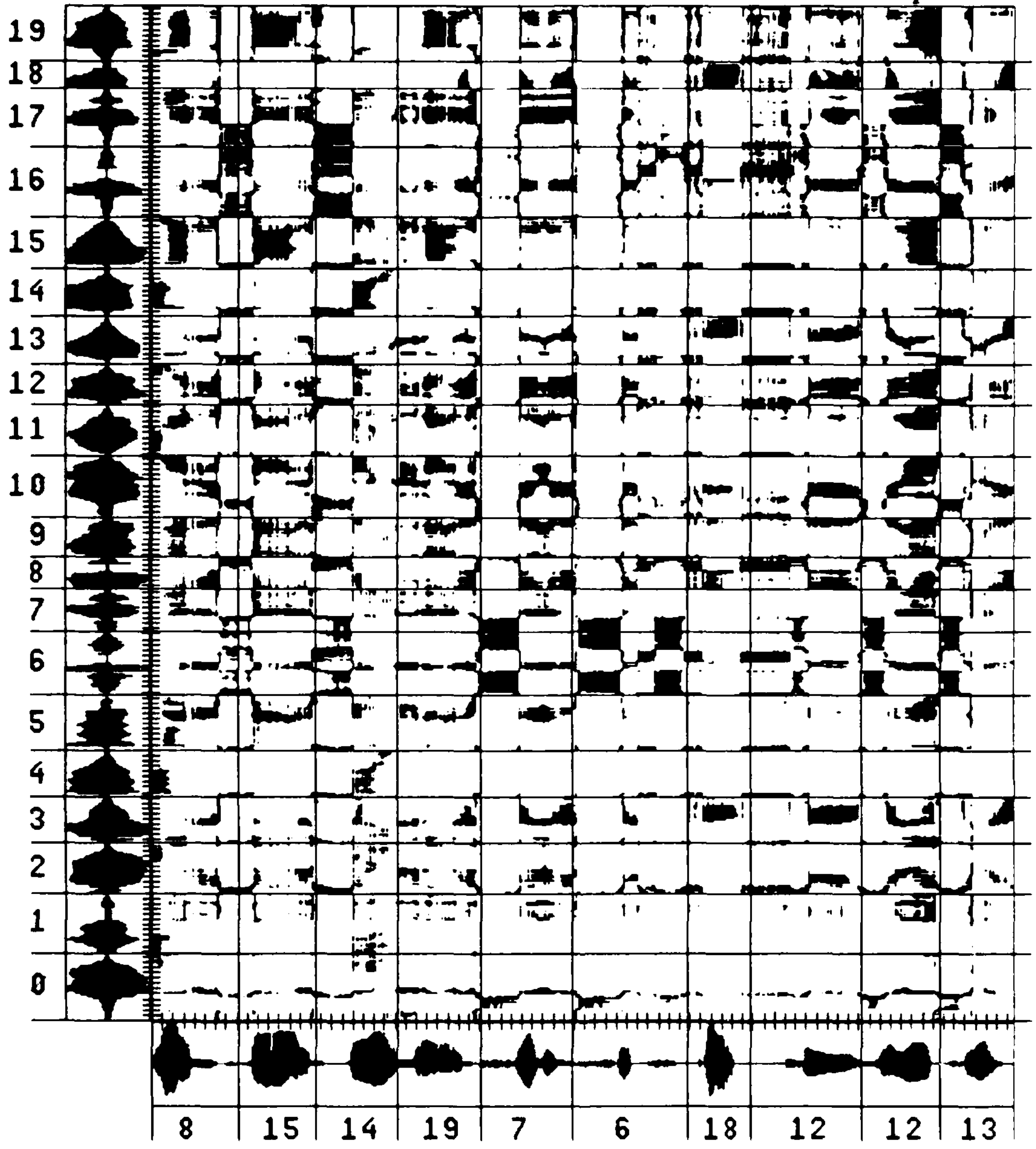
JONES & RGD/SKIP - "1234" LPC Spectrum



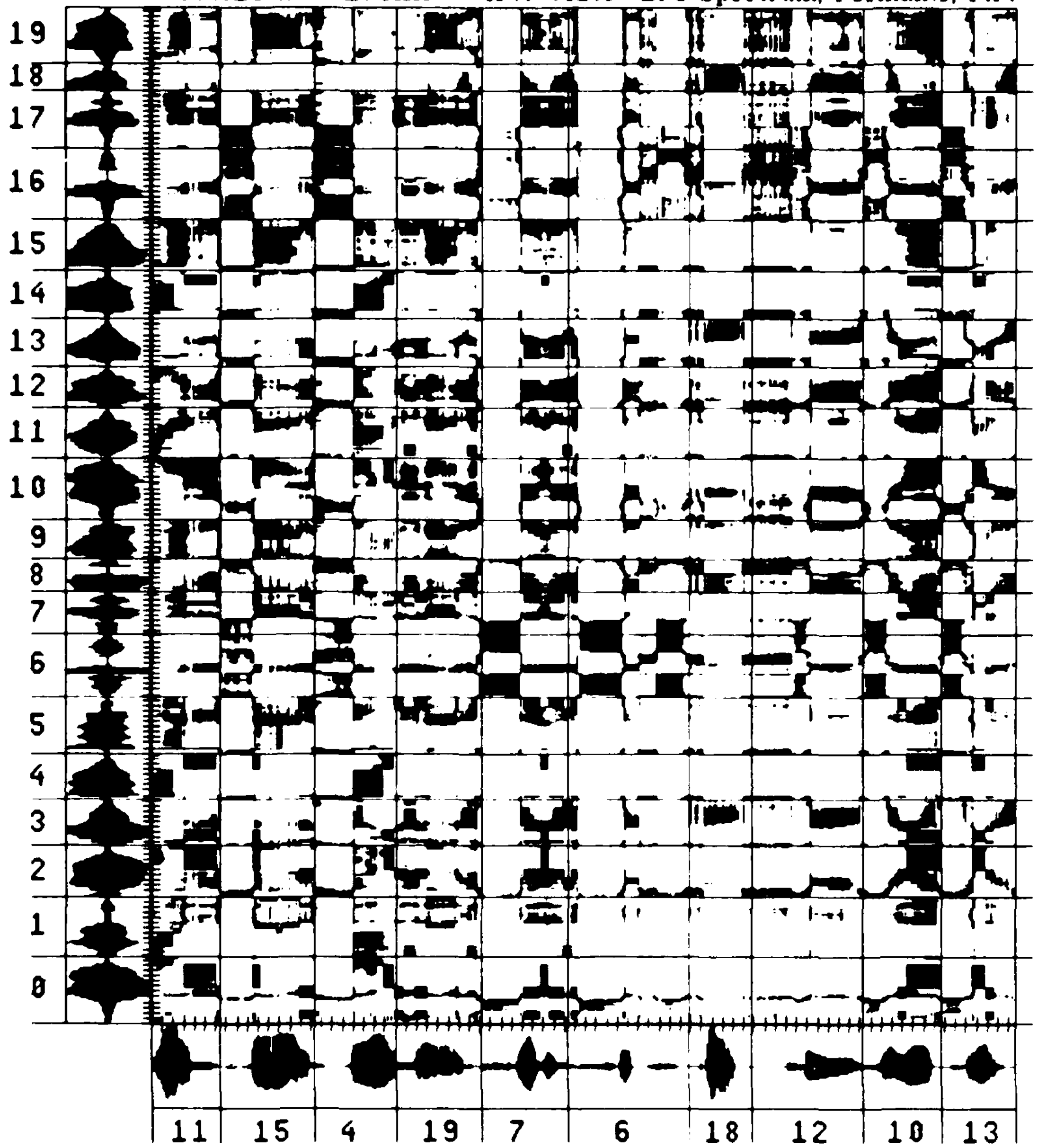
JONES & RGD/SKIP - "1234" LPC Spectrum, Formants, F.F.



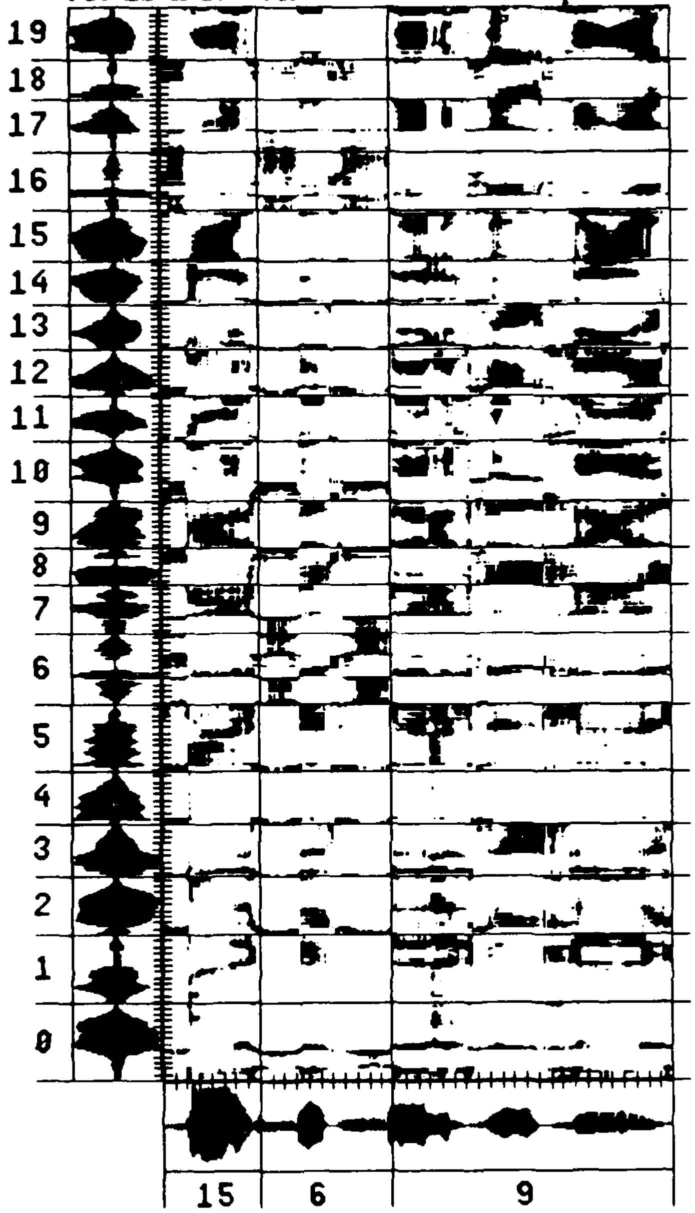
JONES & RGD - "1549768203" LPC Spectrum



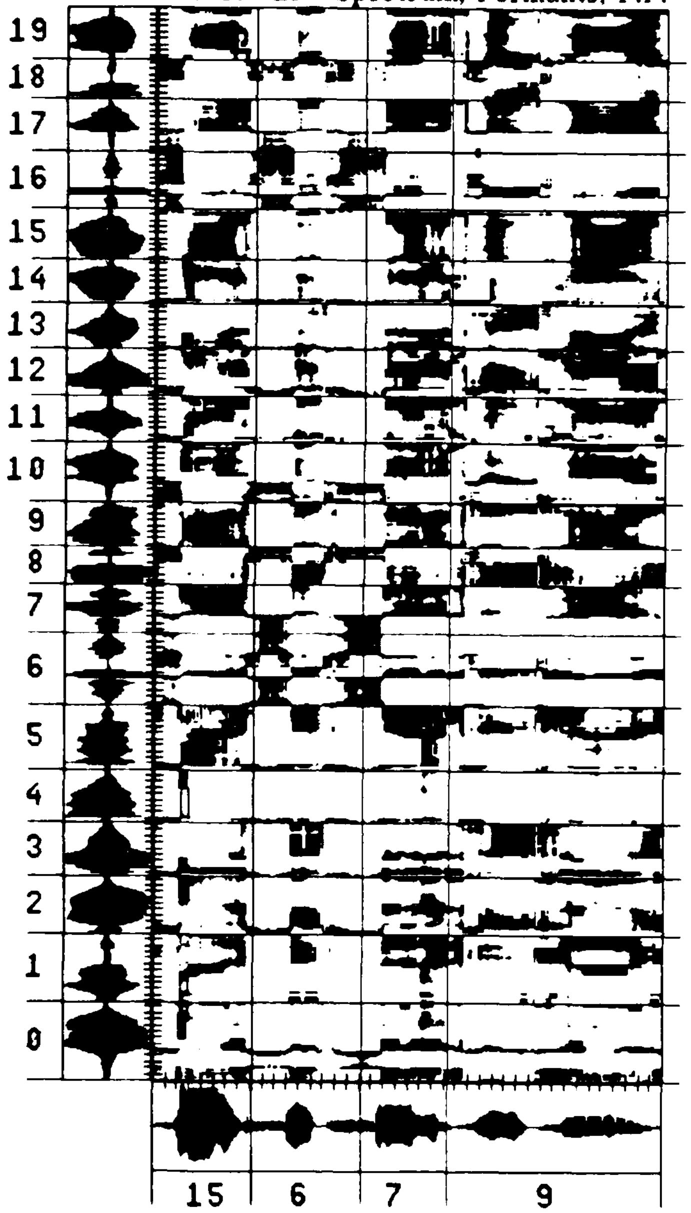
JONES & RGD/SKIP - "1549768203" LPC Spectrum, Formants, F.F.



JONES & SKIP/RGD - "56789" LPC Spectrum



JONES & SKIP/RGD - "56789" LPC Spectrum, Formants, F.F.



Bibliography

1. Abut, Huseyin and Robert M. Gray. "Vector Quantization of Speech and Speech-Like Waveforms," IEEE Trans. Acoust., Speech, Signal Processing. ASSP-25: 299-309 (August 1977).

2. Brusuelas, Capt Micheal A. Investigation of Speaker-Independent Word Recognition Using Multiple Features, Decision Mechanisms, and Template Sets. MS Thesis, AFIT/GCE/ENG/86D-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

3. Burton, David K., John E. Shore and Joseph T. Buck. "Isolated-Word Speech Recognition Using Multisection Vector Quantization Codebooks," IEEE Trans. Acoust., Speech, Signal Processing. ASSP-33: 837-849 (August 1985).

4. Doddington, George R. and Thomas B. Schalk. "Speech Recognition, Turning Theory to Practice," IEEE Spectrum. 18: 26-32 (September 1981).

5. Juang, Biing-Hwang and Lawrence R. Rabiner. "Mixture Autoregressive Hidden Markov Models for Speech Signals," IEEE Trans. Acoust., Speech, Signal Processing. ASSP-33: 1404-1413 (December 1985).

6. Kabrisky, Mathew, Professor. Personal Interview. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 3 February 1987.

7. Kassel, Robert H. A User's Guide to SPIRE. [correspnds to version 17.5] MIT Speech Recognition Group, Mar 1985.

9. Kauffman, David H. SPIRE 17 Release Notes. MIT Speech Group [supported by DARPA contract N00039-85-C-0290 monitored through Naval Electronic Systems Command], January 1986.

10. Ney, Hermann. "The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition," IEEE Trans. Acoust., Speech, Signal Processing. ASSP-32: 263-271 (April 1984).

11. Potter, R. K., George Kopp, and Harriet Green. Visible Processing of Speech Signals. New York: D. Van Nostrand Company, Inc., 1947.

12. Rabiner, Lawrence R. and Ronald Schafer. Digital Processing of Speech Signals. New Jersey: Prentice Hall, Inc., 1978.

13. Rabiner, Lawrence R. and Jay G. Wilpon. "Speaker-Independent Isolated Word Recognition for a Moderate Size (54 Word) Vocabulary," IEEE Trans. Acoust., Speech, Signal Processing. ASSP-27: 583-587 (December 1979).

14. Rothfeder, Jeffery. "Hardware: A Few Words about Voice Technology," PC Magazine. 5: 191-205 (30 September 1986).

15. SPIRE 17.2 Preliminary User's Guide. Speech Communications Group, Research Laboratory of Electronics, Massachussetts Institute of Technology, February 1986.

16. SPIRE 17.2 Reference Manual. Speech Communications Group, Research Laboratory of Electronics, Massachussetts Institute of Technology, February 1986.

17. Winston, Patrick H. and Berthold Horn. LISP. (Second Edition) Massachussetts: Addison-Wesley Publishing Company, 1984.

VITA

Captain Robert G. Dawson was born 28 November 1960 at Burderop Park, England. He graduated from Sylvan Hills High School, North Little Rock, Arkansas in 1979. He received the degree Bachelor of Science Electrical Engineering from the University of Arkansas in August 1983. Upon graduation he received a commission in the USAF and was assigned to the Electronic Systems Division (AFSC), Hanscom AFB, Massachusetts. In May 1986, Captain Dawson entered the School of Engineering, Air Force Institute of Technology.

Permanent Address: 84 Shoshoni Drive
Sherwood, AR 72116

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

A188834

1a REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GE/ENG/87D-14	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: School of Engineering — 6b OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, OH 45433	7b ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING SPONSORING ORGANIZATION — 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO — PROJECT NO — TASK NO — WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
SPIRE based Speaker-Independent Continuous Speech Recognition Using Mixed Feature Sets — UNCLASSIFIED

12 PERSONAL AUTHOR(S)
Dawson, Robert G. Captain USAF

13a TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
MS Thesis	FROM ______ TO ______	1987 December	125

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP
17	02	

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Speech Recognition, SPIRE, Dynamic Programming
Mixed Feature Sets

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Matthew Kabriski, PhD

Professor of Electrical Engineering

Approved for public release: IAW AFR 190-1
[illegible]
LYNN E. WOLAVER [illegible] 31 Dec 87
Dean for Res[illegible] Development
Air Force Inst[illegible]
Wright-Patters[illegible]

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL: Dr. Matthew Kabriski Professor, GS-15	22b TELEPHONE (Include Area Code): (513) 255-5276 — 22c OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Continued from block 19: Abstract

A system was developed to investigate continuous speech recognition. The system incorporates multiple features and dynamic programming to recognize continuous inputs of the spoken digits (zero through nine). The fundamental design concept extends from previous successful recognition research efforts involving both isolated and continuous speech using multiple feature sets, multiple template sets, and dynamic programming. Among the features used in the investigation are wide band spectrogram, narrow band spectrogram, linera predictive coding (LPC) coefficients, LPC spectrum, frication frequency, and formant tracks. An advanced speech research tool called SPIRE provided the computational functions needed to extract the raw features. The system is implemented in LISP on a Symbolics 3600 series LISP machine.

END

FILMED

MARCH, 1988

DTIC