A STUDY IN SPEECH RECOGNITION USING A KOHONEN NEURAL NETWORK MIC PROGRAMMING AND MULTI-FEATURE FU:

THESIS

Wayne F. Recla
Captain, USAF

AFIT/GE/ENG/89D-41

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/89D-41

A STUDY IN SPEECH RECOGNITION USING A KOHONEN NEURAL NETWORK DYNAMIC PROGRAMMING AND MULTI-FEATURE FUSION

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

Wayne F. Recla, B.S.E.E.
Captain, USAF

December 1989

Acknowledgments

Many people helped in making this research effort a success. Dave Doak helped with the processing of the speech utterances on the SPIRE system and offered important guidance on the many features of speech throughout the research effort. Dan Zambon provided the necessary VAX-VMS computing environment and offered invaluable assistance even at odd hours and weekends.

My advisor, Dr. Matthew Kabrisky, is a unique and special person. I feel fortunate to have been under his guidance and direction for this research effort as well as for several classes. Dr. Kabrisky is a true *teacher* whose knowledge, insights, and perspective on engineering and life made studying at AFIT especially rewarding. Maj. Steven Rogers also deserves special recognition. He gave me the initial guidance and direction for this thesis. Maj. Rogers demands and expects the best from his students and his personal level of excellence and achievement set the standard for the research effort.

Finally, I want to thank the person most responsible for my success. Mary, my wife and best friend, provided the help and support within our marriage and family during many years of schooling. Because of her sacrifice and understanding, I was able to fulfill many personal dreams.

Wayne F. Recla

Table of Contents

List of Figures

List of Tables

Abstract

Usually speech systems are one-dimensional. One feature or representation of speech is processed using a specific design methodology. In contrast, the human perception system is multi-dimensional; humans process more than just the sound of the word. Any speech recognition system that mimics human speech perception will need to be multi-dimensional. This methodology formed the basis for the design approach used in this research effort. Linear Predictive Coefficients (LPC) and formants were used as distinct and independent inputs into a recognition system consisting of a Kohonen neural network and a dynamic programming word classifier. A feature-fusion section and rule-based system were used to integrate the two input feature sets into one output result.

The first half of this research effort involved extensive testing of the Kohonen network. Using a speech input signal, different Kohonen gain reduction methods, initial gain values, and conscience values were tested for various iteration times in an effort to quantify the response and capabilities of the Kohonen network. Three-dimensional Kohonen-Dynamic Programming surfaces were developed that graphically showed the effects of gain, conscience, and iteration time on the speech recognition response of a Kohonen neural network. A new standard iteration time called a *multiple* was used during training of the Kohonen networks.

The results of the basic research on the Kohonen network produced an optimized Kohonen configuration that was used in the multiple-feature recognition system. A 70-word vocabulary of F-16 cockpit commands were used to evaluate the new feature-fusion method. The feature-fusion section performed well and was able to correctly classify words even when each individual input feature gave an incorrect answer.

A STUDY IN SPEECH RECOGNITION USING A KOHONEN NEURAL NETWORK DYNAMIC PROGRAMMING AND MULTI-FEATURE FUSION

I. Introduction

The ability to communicate by speech seems fundamental to our nature. From prehistoric times to the present, speech has increased our chance of survival and has contributed to our development as social animals. It is not surprising that people would want to communicate with machines by speech, especially since machines have become such an integral part of our environment.

The benefits of an accurate voice interface between humans and machines, that approaches the capabilities of human speech perception, would be enormous. Most service industries, small businesses, and large corporations would share in the multiple potential uses of automatic speech machines. Whether adjusting a television set, preparing a meal, or driving an automobile; controlling machines by voice would be quick and nearly effortless. With the computer controlling the generation and transfer of information in today's society, the effect of having a speech recognizer that replaces the keyboard as the primary human-machine interface would be revolutionary.

Military applications of speech recognition systems, although potentially widespread, focus on today's advanced combat aircraft environment [5:311]. A recent study of F-15 and F-16 aircraft accidents indicates that speech recognition systems would have prevented many of these accidents. Pilots became preoccupied or distracted by the myriad of tasks within the cockpit and lost their "situational

awareness" [10:95]. The results of this study show the importance of continued research and development of adequate speech recognition systems within the military research community. Continued research will eventually allow us to fully comprehend our own ability to process and decode human sounds. The question now is not *if* but *when* will humans fully understand and replicate speech and human speech perception.

Background

The growing realization that the human-machine interface within today's advanced fighter aircraft has reached the limits of efficiency and reliability prompted the Air Force to include a Voice Interactive System in the Advanced Fighter Technology Integration (AFTI) program. The AFTI program, which uses an F-16 testbed aircraft, seeks new technological answers to tomorrow's combat environment [10:88].

Although state-of-the-art recognizers were used, the AFTI Voice Interactive Systems attained accuracies only in the 85% range under operational conditions [10:96]. More accurate systems are now available, but overall, current speech recognition systems are inadequate to the task of decoding and processing human speech sounds with accuracies that approach human speech perception.

Most people easily perceive what is being said regardless of the speaker or the time interval between words if they know a common language. Speech recognizers have not reached this level of maturity. The complexity involved in speech recognition has forced researchers to seek intermediate solutions by classifying recognizers according to the speaker and according to the time relationship between words. A recognizer that is accurate only with voice input from one speaker is termed a speaker-dependent recognizer. A recognizer requiring pauses between adjacent words is termed an isolated-word recognizer. The most accurate existing speech recognizers are speaker-dependent, isolated-word systems [21:1].

A speech recognition system will be accepted by the user community only if the system allows for the natural flow of information between humans and machines. This fact effectively precludes the widespread use of isolated-word systems or their acceptance by many potential users. The solution space for a connected-word system, however, is significantly more complicated. Connected-word systems that are speaker-dependent would offer an acceptable compromise between our desire to mimic human speech perception and the complex nature of the problem. A speaker-dependent system would eliminate a significant number of recognition errors due to cross-speaker variability. A speaker-dependent system does not have to account for cross-speaker differences related to the acoustic effects of dissimilar vocal-track lengths, or differences related to variations in spoken phonetic units due to dialect or idiolect [15:305].

A speaker-dependent system would also be practical. For example, Air Force pilots could carry a plastic card imprinted with their voice patterns or acoustic sounds that represented all the phonetic sound units present within the recognizer's vocabulary. The pilot could place the card into the generic Voice Interactive System available in every aircraft thereby making the system unique for that pilot. If the aircraft developed problems prior to take-off, the pilot could switch aircraft and utilize its generic recognizer. This technology could be used with other machines, including office computers and land vehicles.

Typically, recognition systems process a single feature or representation of the speech signal. Methods, that are used to represent speech, attempt to preserve the relationship between time and frequency inherent within every spoken word. Processing methods usually use pattern-matching techniques in determining the correct word or utterance. The recent introduction of neural networks in speech research is an attempt to mimic the human recognition process.

Problem Statement

The primary purpose of this thesis is the development of a recognition system using a Kohonen neural network, dynamic programming, and multi-feature fusion. The recognition system will be speaker-dependent, and will be capable of recognizing both isolated and continuous speech.

Several features or representations of the speech signal are used in determining the correct word or utterance. Features are processed using a neural network and a word classifier based on dynamic programming. The Kohonen network organizes the speech data onto a two-dimensional grid. Dynamic programming then fuses together the two-dimensional output of the Kohonen network and the speaker's formant frequencies to produce the desired output.

The response of a Kohonen neural network to speech input is not well known. In order to optimize the accuracy of the recognition system, a significant amount of this thesis effort is spent testing the various parameters that affect the response of the Kohonen network to speech.

Scope of This Research

Initially, the response of the Kohonen neural network to speech data is ascertained. Parameters are tested that affect the ability of a Kohonen network to group sounds into discrete regions on its surface. Various gain reduction methods are evaluated along with different levels of conscience. The size of the two-dimensional Kohonen network grid remains constant along with the range of the initial weight values for the Kohonen network.

The vocabulary of the final speech recognition system consists of the 70 word, F-16 command vocabulary shown in Table 1. All the words are included in the recognition system. Also, all 70 words are used to test the final recognition system. The system is speaker-dependent.

Table 1. F-16 Cockpit Commands

1. Advise	25. Five	48. Profile
2. Affirmative	26. Flares	49. Radar
3. Aft	27. Forward	50. Range
4. Air-to-Air	28. Four	51. Report
5. Air-to-Surface	29. Foxtrot	52. Rhaw
6. Alpha	30. Frequency	53. Search
7. Arm	31. Fuel	54. Select
8. Backspace	32. Gun	55. Seven
9. Bearing	33. Heading	56. Six
10. Bravo	34. Hundred	57. SMS
11. Cancel	35. Knots	58. South
12. Chaff	36. Lock-On	59. Station
13. Change	37. Map	60. Strafe
14. Charlie	38. Mark	61. Tail
15. Channel	39. Miles	62. Target
16. Clear	40. Minus	63. Thousand
17. Confirm	41. Missile	64. Threat
18. Degrees	42. Negative	65. Three
19. Delta	43. Nine	66. Two
20. East	44. North	67. Waypoint
21. Echo	45. Nose	68. Weapon
22. Eight	46. One	69. West
23. Enter	47. Point	70. Zero
24. Fault		

Assumptions

All speech data are digitized by a Digital Sound Corporation (DSC) Analog to Digital convertor. To preclude the effects of aliasing, this research assumes the DSC signal convertor correctly low-pass filters and samples the data at or higher than the Nyquist frequency before the analog-to-digital conversion process, otherwise the effects of aliasing would corrupt the data and invalidate the test results.

Standards

This research effort uses the F-16 command vocabulary. Except for Captain Peter Kim's thesis work, which included research on merged templates using the F-16 command vocabulary [14], other thesis efforts at the Air Force Institute of Technology (AFIT) considered only the spoken digits zero through nine as possible vocabulary words. The digits are a subset of the F-16 vocabulary.

Barmore used this subset of the F-16 vocabulary in his Kohonen based speech recognition system [4]. As a means of gauging the performance characteristics of the Kohonen network during gain reduction and network conscience testing, the vocabulary of the recognition system in Phase I of this thesis consists of the digits zero through nine only. These digit sounds or utterances consist of the same data set used in the Barmore thesis. Using Barmore's speech files in Phase I allows for easy comparison between the various gain reduction methods evaluated in this thesis and the reduction method used in Barmore's system.

The entire F-16 command vocabulary is included within the recognition system during Phase II. Although Kim's system contained a 70 word vocabulary, only 20 of the total 70 words were used to test the final recognition system. This thesis effort, however, uses all 70 words to test the final recognition system.

Approach

The project is divided into two phases. Improvements or capabilities realized in Phase I are incorporated into Phase II testing. The first phase investigates methods to optimize the performance of the Kohonen network used by Barmore [4]. The vocabulary consists of the isolated digits zero through nine from Barmore's speech files. Both isolated and connected speech data are used. The areas covered in Phase I include:

- Investigating different gain reduction methods as applied to Kohonen neural network training of speech data.
- Determining the effect of varying conscience on the output performance of a Kohonen neural network.
- Investigating the effectiveness of a Kohonen network to organize speech data by determining the minimum training cycle time required for adequate speech recognition.
- Investigating the effect of average subtraction on the output performance of a Kohonen neural network.

The entire F-16 command vocabulary is added to the system in Phase II.

Materials and Equipment

The following materials and equipment were used:

- Digital Sound Corporation (DSC) Analog to Digital convertor.
- VaxWorkstation II running the VMS operating system.
- MicroVax III running the VMS operating system.
- VAX 'C' Language Complier using the Graphics Kernel System (GKS).

- Symbolics 3600 Lisp Machine running SPIRE software.
- Noise Reducing Microphone (SHURE Model SM54).

Sequence of Presentation

Chapter II provides background information about the variable nature of speech and design approaches used in currently available speech recognitions systems. Dynamic programming and Kohonen artificial neural networks, as integral parts of this thesis effort, receive additional coverage.

Chapter III describes the hardware and software environment required to develop the recognition system.

Chapter IV describes the system used to test the Kohonen network in Phase I and the speech recognition system in Phase II. Additional coverage is given to Kohonen network structure and dynamic programming.

Chapter V discusses the results of testing different configurations of Kohonen network structure and also testing of the final speech recognition system.

Chapter VI provides the conclusion and recommendations for further application of the recognition system developed in this thesis.

II. Literature Review

Introduction

The computational ability of the digital computer has given researchers the tool needed to process many of the unique features of speech. As a result, a number of speech recognizers have been developed over the last several years. The early hope, however, of approximating human speech perception has become an elusive goal. A glance at the session headings of the 1988 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)(New York City, April 1988) is a good indicator of our current ability to recognize speech by machine. Three sessions representing over 30 papers were devoted solely to isolated speech recognition. Clearly, much still remains to be learned about the structure of speech.

This section starts by discussing some of the difficulties encountered in recognizing speech sounds by machine. Various kinds of speech variability are described along with their effects on the speech signal. Next, the two main approaches to the problem of speech recognition are presented. Starting with the Conventional or Traditional approach, both dynamic programming and Hidden Markov Models are described with emphasis given to dynamic programming. After briefly looking at the biological evocation of artificial neural networks, attention is then focused on the two primary artificial neural networks used in speech recognition research today. The Kohonen neural network along with thoughts on human speech processing in the brain by Dr. Teuvo Kohonen is delineated. Finally, speech recognition research using artificial neural networks at AFIT is reviewed.

Variability of Speech

Variability within the speech signal is the primary reason why current machines cannot decode speech as well as people. This signal is inherently noisy and

is produced by a part of the human anatomy with a primary function of transfer of air and consumption of food [9:48]. Additionally, other factors listed below compound the problem of practical speech recognition:

- Within-speaker variability: Our own speech normally varies in pitch, loudness, and quality depending on our mood, health and stress level. Also, we pronounce words only to the extent necessary for recognition. For example, words with the suffix "ing" are often pronounced with the "g" sound omitted [15:304].
- Cross-speaker variability: Speech is composed of several speaker-dependent variables that range from differing voice box characteristics to varying regional dialects. For example, a person from Massachusetts would pronounce the word "car"quite differently than would a person from Illinois [15:304].
- Coarticulation: Coarticulation is the process in which the speech organs start the transition to the next sound while still producing the current sound. This mutual corruption of individual sounds is subject to considerable variation and tends to be speaker-dependent [15:304][25:341].
- Environment: The brain appears to decode and process speech sounds differently from nonspeech sounds. This ability helps people interpret speech sounds within noisy surroundings [25:75]. Machines, however, process the entire signal (speech and noise) without regard to its origin [15:304].

Current speech recognizers cannot cope with these variables inherent in every speech signal. An inadequate understanding of these combined effects is probably the single most important factor inhibiting the development of a reliable speech recognition system [15:304].

Conventional Speech Recognizers

The most successful speech recognizers employ either pattern-matching techniques or stochastic modeling of speech. Pattern based recognizers may use words, phonemes, or other features characteristic of the speech signal as the input pattern. A word input pattern is often called a template. The input pattern is identified by finding the closest match between this pattern and all the stored patterns in the vocabulary file of the recognizer [20:5]. Kim gives an excellent review of speech recognition research using pattern based recognizers at AFIT over the last decade [14:2-1:2-21].

Dynamic Programming The combined temporal-spectral nature of speech hindered the development of pattern-matching algorithms for many years. Speech consists of the continuous flow of phonetic sound units. Unfortunately for the recognition problem, the time duration of individual sound units is not fixed, but varies between each recorded utterance. This temporal aspect of speech complicates the comparison of the spectral features that are unique to each word or template. Some type of time normalization is therefore required for accurate recognition of an unknown utterance. One of the first techniques involved simple compression or expansion of the unknown utterance until it was the same length as the library reference template. Even with accurate end-point alignment, the method failed to accurately time align individual sounds. Alignment of maximum energy points within the unknown utterance was also considered [26:354]. The introduction of dynamic programming effectively solved the problem of time normalization in isolated-word recognizers.

Dynamic programming or dynamic "time warping" (DTW) is the method of choice in pattern-matching machines today. With this method, the individual spectral features of the unknown utterance are brought into alignment with the reference template by nonuniformly distorting or warping its time axis with respect

to the reference time axis. The input vectors for both the unknown and reference words to DTW represent some unique feature of speech. Typical features include signal amplitude, formants, or linear predictive coefficients (LPC) [25:297]. The attractiveness of dynamic programming is due to the simultaneous execution of nonlinear time alignment and recognition of the unknown word.

Hidden Markov Models The use of the hidden Markov Model (HMM) to stochastically model speech has gained favor in recent years. Recognition accuracies are now similar to pattern based systems with additional improvements anticipated as the model is further refined. A HMM recognizer will build a Markov model for each word in its vocabulary. An input word matches a vocabulary word whose Markov model gives the highest probability that the Markov model gave rise to the input word. Hidden Markov Modeling, however, requires a detailed knowledge of the segmentation rules within a given language [24:263]. As with pattern based systems, HMM recognizers perform well as isolated-word, speaker-dependent systems but perform poorly when tested with connected speech from multiple speakers [25:307- 310][27:13].

Both pattern based and HMM recognizers by themselves lack the robustness necessary to deal with the variability present in human speech. As a result, a different approach to recognition modeling centers on how humans decode speech.

Biological Neural Networks

Information processing in the brain involves the interaction of large numbers of neural processing elements connected in a highly parallel fashion. Biological neural networks appear to process information in a manner that is different from digital computers. At the cell level, each element acts as an analog processor of thousands of inputs. "Threshold" levels within neurons are time variable and not adequately stable enough to define functions that characterize the digital world.

At the network level, this instability precludes programming (learning) in an algorithmic fashion where processing is recursive in nature [16:4-6]. Teuvo Kohonen, who developed the Kohonen artificial neural network, concludes that a biological neural network is probably programmed (i.e. *learns*) in one of two ways:

- The structures of interconnections between processing elements are altered.
- The strengths or "signal transmittances" of these interconnections are changed.

How interconnections are altered is unclear and not easily quantified. Each signal strength change between processing nodes, however, is a straightforward process found in many artificial neural models [16:4-6].

Artificial Neural Networks

Two different neural network models have received the most interest within the speech recognition community. They are the Perceptron and the Kohonon net. Although both models are different in design implementation, they share the concept of varying interconnection signal strengths between processing elements. Each model is an adaptive structure; present weighted strength values are dependent upon past weighted strength values and the input pattern. Applying an input pattern over many iterations represents "learning."

The learning or training phase of a neural net is accomplished in either a *supervised* or *unsupervised* fashion. In *supervised* mode, the network receives guidance from outside the network in determining new weighted interconnection signal strength values. In *unsupervised* mode, the network makes its own decisions concerning new signal strength values [22:3].

Perceptron

Single-Layer Perceptron (SLP) Initial research involving single-layer perceptrons dates back 30 years [29]. The model is linear, uses a novel learn-

ing algorithm, and is simple in parallel computational design [23:231-232]. The linearity[1] of single-layer perceptrons, however, restricts its use to recognition of simple patterns where the input can belong to one of only two decision regions or classes. A hyperplane separates the two decision regions [20:13]. Since the decision space is partitioned by linear boundaries, a single-layer perceptron cannot compute the exclusive-OR function. This restriction probably prohibits the use of the single-layer perceptron in speech recognizers unless other types of auxiliary neural networks are also present. The exclusive-OR problem was finally solved with the multi-layered perceptron.

Multi-Layered Perceptron (MLP) Multi-layered perceptrons allow for the solution of a large class of problems that map as either bounded or unbounded complex regions. This capability is provided by non-linear boundaries introduced by the hidden layers during training of the network.

Training of a multi-layered perceptron is accomplished in *supervised* mode by setting the output nodes to a desired output pattern. For a given input, individual nodes are activated at the input layer. Activation of nodes flows through the hidden layers to the output layer with individual node values determined by the weighted interconnection signal strengths.

The activations that propagate through the network are then compared with the desired output pattern. Error terms are computed at each output node. These error terms are then used to determine new weighted interconnection signal strength values that will tend to minimize the output error on the next training iteration. Depending upon the complexity of the input pattern and size of the network, learning may require thousands of iterations [20:15-18][11:158].

[1] Linearity refers to the solution of linearly separable sets in the output decision space.

Kohonen Network The Kohonen network reflects Teuvo Kohonen's perception of how programming or learning may take place in humans. He writes:

> It is indeed very difficult to imagine how such an enormous network could be 'programmed' at all. One possibility, especially relating to the sensory subsystems could be that the system structure, or the dynamical process defined by it, in some way directly tends to image the sensory experiences or other occurrences. One does not thereby mean any photographic models, static representations of the environment, or metrically faithful copies of signals; 'imaging' must be understood in a more general and abstract sense, for example, that certain feature dimensions of the observations are imaged, or that there appear events in the behavior of the network, the temporal, or logic, relationships between some events of the exterior world or its history. [16:4-5]

Biological neurons do not respond to external stimuli in isolation. Interactions between neighboring neural elements that excite close neighbors and inhibit far neighbors helps to explain the relationship between the Kohonen network and biological responses to speech input. Kohonen describes this type of neural coupling as the "Mexican hat function." These Mexican hats or "bubbles" form local clusters in locations solely dependent upon the input stimulus [17:11].

The "imaging" surface of the Kohonen network (see figure 1) is a two-dimensional array of output nodes. Each node represents a neuron that is connected to every input node with a interconnected weight value [20:18-19][22:6-7]. An input stimulus (feature of speech) applied to the network will produce a response at each output node that depends on the summation of the individual input signal values and their associated weights:

$$y_i = \sum_{j=1}^{N} m_{i,j} x_j \tag{1}$$

where y_i represents the i^{th} output node, $m_{i,j}$ is the weight connecting the j^{th} component of the input to the i^{th} output node, and x_j is the input stimulus value of the j^{th} component. The winning neural output node with index $i = c$ is the

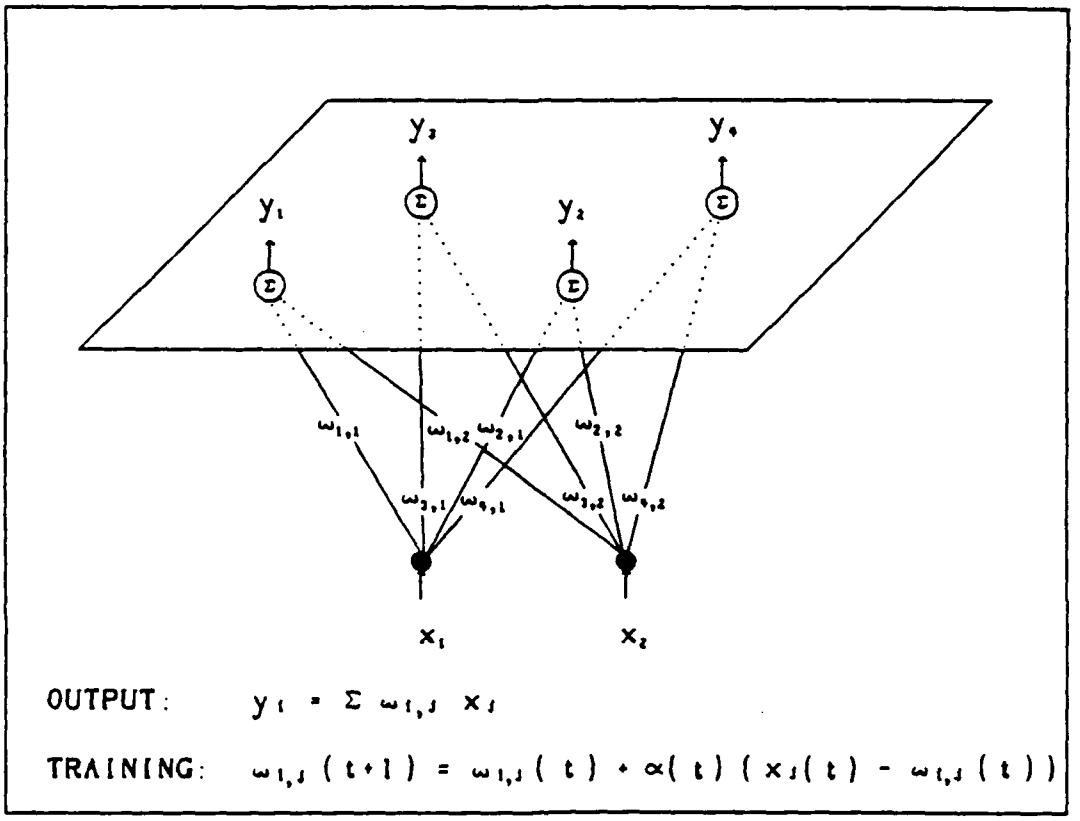

Figure 1: Kohonen Neural Network 4:2-7

absolute valued minimum attained from equation 1. In terms of distance, the *winning* node has a smaller distance between the *winning* node's weight vector and the input vector than between any other node's weight vector and the input vector [4:3-7].

The "bubble", or neighborhood N_c, centered on output node c is then trained according to the equation:

$$m_{i,j}(t+1) = m_{i,j}(t) + \alpha(t)(x_j(t) - m_{i,j}) \tag{2}$$

if i is contained in N_c. A node's weight value remains unchanged if i is outside of N_c. α is a positive scalar gain factor that initially starts with a value near unity. As training progresses, α decreases gradually toward zero. During each training iteration, equation 1 and equation 2 are applied alternately as new interconnection signal strength values are computed within the neighborhood [17:13-17]. In short, the learning algorithm for a Kohonen network is a two-step process. For each iteration:

- Locate the *winning* node.
- Increase the match at this node and its topological neighbors with the set of input vectors.

Eventually, a unique pattern is obtained for each word or phoneme [20:18-19].

AFIT Speech Research with Neural Networks

In 1988, two AFIT theses were completed using neural networks to model speech [4,3,22]. Captain Gary Barmore used a Kohonen network that created trajectory patterns for the spoken digits zero through nine. Recognition was speaker-dependent and both isolated and connected words were analyzed. The output

of two-dimensional coordinate vectors from the Kohonen net was fed into a conventional dynamic time warping (DTW) algorithm for word recognition. Testing was also conducted with a second Kohonen network for recognition of the test utterance.

With DTW, isolated and connected recognition rates were 99.17% and 90.7% respectively. Word recognition rates were lower when the recognizer used the second Kohonen net for word recognition. The results with a second Kohonen net were 96% for isolated words and 81% for connected words [4:5-1].

Using the same vocabulary, Captain Mark Lutey developed an isolated-word recognition system using a Kohonen network and a multi-layer perceptron. Three speakers (two male and one female) recorded speech digits for both training and testing of the network. Recognition rates were highest when a network was tested using a second utterance from the original speaker and when the starting and ending locations of a word were known. The recognition rates, however, decreased sharply when the starting and ending points of a word were unknown and when utterances between speakers were compared. Overall recognition rates ranged from 32% to 80% for the three speakers.

Conclusion

This brief review of the speech signal and of the methods used in the recognition process highlights the difficulty in developing adequate and reliable speech recognition systems. Indeed, research in this field has continued for more than 40 years.

This research effort merges a Kohonen neural network and dynamic programming into a hybrid system similar to the recognition system designed by Barmore. The next chapter provides a discussion of the programming environment for this system.

III. System Environment

Introduction

The purpose of this chapter is to describe the hardware and software environment used to design the recognition system. There are three sections. The first section describes the method used to obtain and digitize speech sounds. The next section describes a dedicated speech processing tool used to extract the appropriate features of speech. The last section describes the software environment for the recognition system.

Sound Generation

Because human sounds are analog in nature, initial processing must convert this analog signal into a digital format required by a computer. This conversion was performed by a DSC-200 Analog to Digital converter at a 16 KHz rate. This system accurately provides the necessary signal processing environment required for conversion of speech signals.

The DSC-200 was connected to a VaxWorkstation II via UNIBUS interface. With a noise reducing microphone, a user can easily generate and convert the speech sounds necessary for training and testing the recognition system. A total of five sets of 70 isolated words were processed along with several connected utterances. The DSC-200 has a maximum processing buffer size of 20 seconds.

Each recorded word buffer was usually one or two seconds in duration while each actual isolated utterance was approximately 0.7 seconds in length. A larger recording buffer guaranteed that the start and end of each word was captured for further processing. Large recording buffers, however, required considerable amounts of computer memory. All word buffer files were then sent to the Speech

and Phonetics Interactive Research Environment (SPIRE) for further processing which included trimming off excess recording buffer on either side of each word.

SPIRE

SPIRE is a software program which allows a user to process and evaluate speech signals. The capability also exists to extract the multiple features of speech required in the recognition process. SPIRE was developed by the Speech Research Group at the Massachusetts Institute of Technology (MIT), and it is available by license through the MIT patent office. In addition to the processing ability of SPIRE, the system provides excellent graphical representations of all recognition features of speech. A list of graphical displays available from SPIRE is given in Figure 4 and represents the default displays available from the system. Other representations are possible. For example, energy values in the formant range were required for this thesis effort. A separate software function was written within SPIRE for this purpose.

SPIRE's graphical capabilities are illustrated in Figure 5 which shows the utterance **target** and Figure 6 which shows the utterance **select gun strafe charlie**. Each figure displays the original waveform, Linear Predictive Coding (LPC) spectrum slice, formants, and energy content for the respective utterance.

Files for each of the seventy words in the vocabulary are referenced numerically. Table 2 lists the numbering scheme used in naming sound files. The same numbering scheme was used in the displayed output from the recognition system.

In Figure 5, the word **target** is the forty-eighth word in the vocabulary. The original waveform (top graph) shows the normalized amplitude of the utterance plotted against time on the horizontal axis. The time axis has a duration of 0.65 seconds. SPIRE allows evaluation of speech parameters at specific points in time which are specified by the user. In Figure 5, a time slice at approximately 0.4 seconds was chosen (see vertical line in top graph). The LPC Spectrum slice *at* 0.4

Table 2. F-16 Cockpit Commands (Numerical Cross-Reference Listing Used in File Names, Display Output of SPIRE and Recognition System)

0. Zero	24. Tail	48. Target
1. One	25. Clear	49. Weapon
2. Two	26. Profile	50. West
3. Three	27. Report	51. Air-to-Surface
4. Four	28. Affirmative	52. Charlie
5. Five	29 Frequency	53. Fault
6. Six	30. Waypoint	54. Gun
7. Seven	31. Advise	55. Knots
8. Eight	32. Bearing	56. Miles
9. Nine	33. Cancel	57. North
10. (Space)	34. Degrees	58. Range
11. Aft	35. Foxtrot	59. Select
12. Arm	36. Lock-on	60. Forward
13. Chaff	37. Missile	61. Alpha
14. Delta	38. Radar	62. Change
15. Flares	39. Station	63. Channel
16. Hundred	40. Threat	64. Confirm
17. Negative	41. Bravo	65. Heading
18. Air-to-Air	42. Echo	66. Minus
19. Backspace	43. Enter	67. Rhaw
20. Thousand	44. Mark	68. SMS
21. East	45. Nose	69. Strafe
22. Fuel	46. Search	70. Point
23. Map	47. South	

seconds is shown in the second graph from the top. This graph, which plots an LPC spectral slice against frequency, shows an envelope generated by LPC coefficients. The peaks within the envelope correspond to resonant peaks of the vocal track.

The third graph shows the formant tracks for the first four formants plotted against time. The value of each formant is displayed at the position specified by the user (0.4 seconds). The value of the first formant at this position is $468.8Hz$.

The formants, which represent the resonant frequencies of the vocal track, characterize the shape of the oral cavity during sound production. The most basic sounds are called *phonemes.* The phonemes in the English language[1] are shown in Figure 2. The vowels are produced by a combination of vocal cord vibration and fixed cavity shape. For each vowel, the tongue, jaw, and lips are positioned to allow the unrestricted flow of air through the oral cavity. There is a definite relationship between each vowel sound and the formant frequencies. The vowel formant frequencies are shown in Figure 3.

The areas of the formant graph in Figure 5 with vertical spikes represent consonant-type sounds (*fricatives, stops* etc.) within the spoken word. The vocal track is constricted at some point during the production of these sounds. The location of the constriction determines which sound is produced [28:43-54]. The constricted air flow creates enough broadband energy to cause erratic formant tracking. These areas contain unreliable formant information.

The bottom graph shows the energy of the signal contained in the frequency region from $250Hz$ to $3000Hz$. The energy value specified at 0.4 seconds is - 63.58 dB. A threshold value of - 75.0 dB[2] was used in this thesis effort. Formant data was zeroed out if the energy value for the corresponding time slice was *below* - 75 dB.

[1]There is not universal agreement as to the total number of phonemes in the English language.
[2]An internal SPIRE defined energy reference.

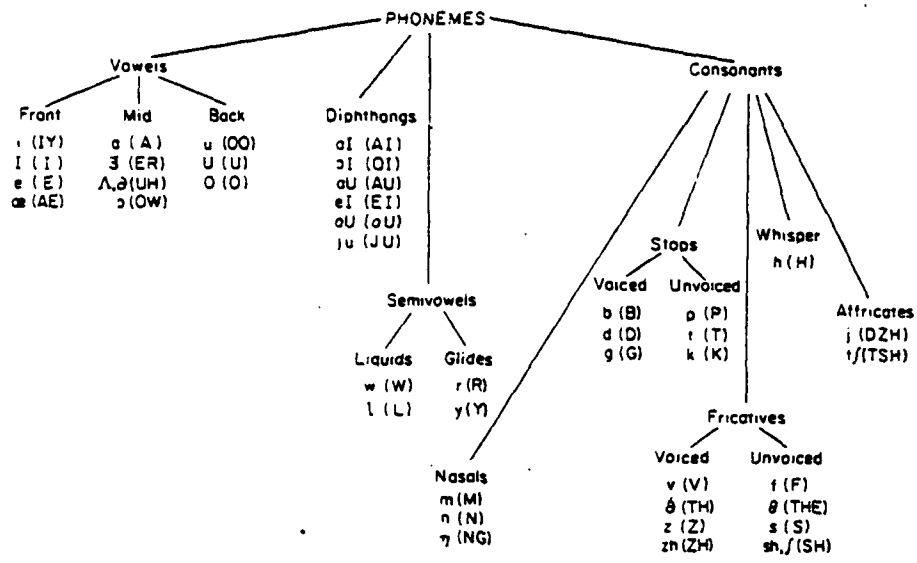

Figure 16: Phonetic and Orthographic Representation of American English Phonemes 28:43

FORMANT FREQUENCIES FOR THE VOWELS					
Typewritten Symbol for Vowel	IPA Symbol	Typical Word	F_1	F_2	F_3
IY	i	(beat)	270	2290	3010
I	ɪ	(bit)	390	1990	2550
E	ɛ	(bet)	530	1840	2480
AE	æ	(bat)	660	1720	2410
UH	ʌ	(but)	520	1190	2390
A	ɑ	(hot)	730	1090	2440
OW	ɔ	(bought)	570	840	2410
U	ʊ	(foot)	440	1020	2240
OO	u	(boot)	300	870	2240
ER	ɜ	(bird)	490	1350	1690

Figure 17: Average Vowel Formant Frequencies 28:45

SPIRE requires specialized hardware for correct operation. At AFIT, SPIRE is run on a Symbolics 3600 Lisp machine. The Symbolics 3600 provides the necessary interactive and graphical environment required by SPIRE software. Further information on SPIRE is available from [12,13,1,2].

Software Development

Except for the SPIRE software program which is written in LISP, all software is written in the 'C' programming language on a MicroVax III system running the VMS version 5.1 operating system. All programs are written in a modular fashion with adherence to software engineering principles. Emphasis is placed on adequate documentation of the code for easy maintainability.

Several modules from Barmore's thesis effort are used in modified form. Also, the file naming convention used is the same one that Barmore used in his thesis. All unprocessed sound files are denoted as **.snd* files and all processed sound files are denoted as **.trn* files.

Programs were normally run in batch mode on the MicroVax III using a *.com* file. A batch run is initiated by the 'SUBMIT' command. Figure 7 gives an example .com file that executes a dynamic programming program. A *.com* file requires a '$' before any command while lines without a '$' are used for data or other input requests. Note in Figure 7, the last line initiates another batch run when the current batch job is complete.

Conclusion

This chapter described the software and hardware operating environment required for system development. The next chapter describes in detail the mythology used to realize the complete system.

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

Figure 4. Standard SPIRE Displays

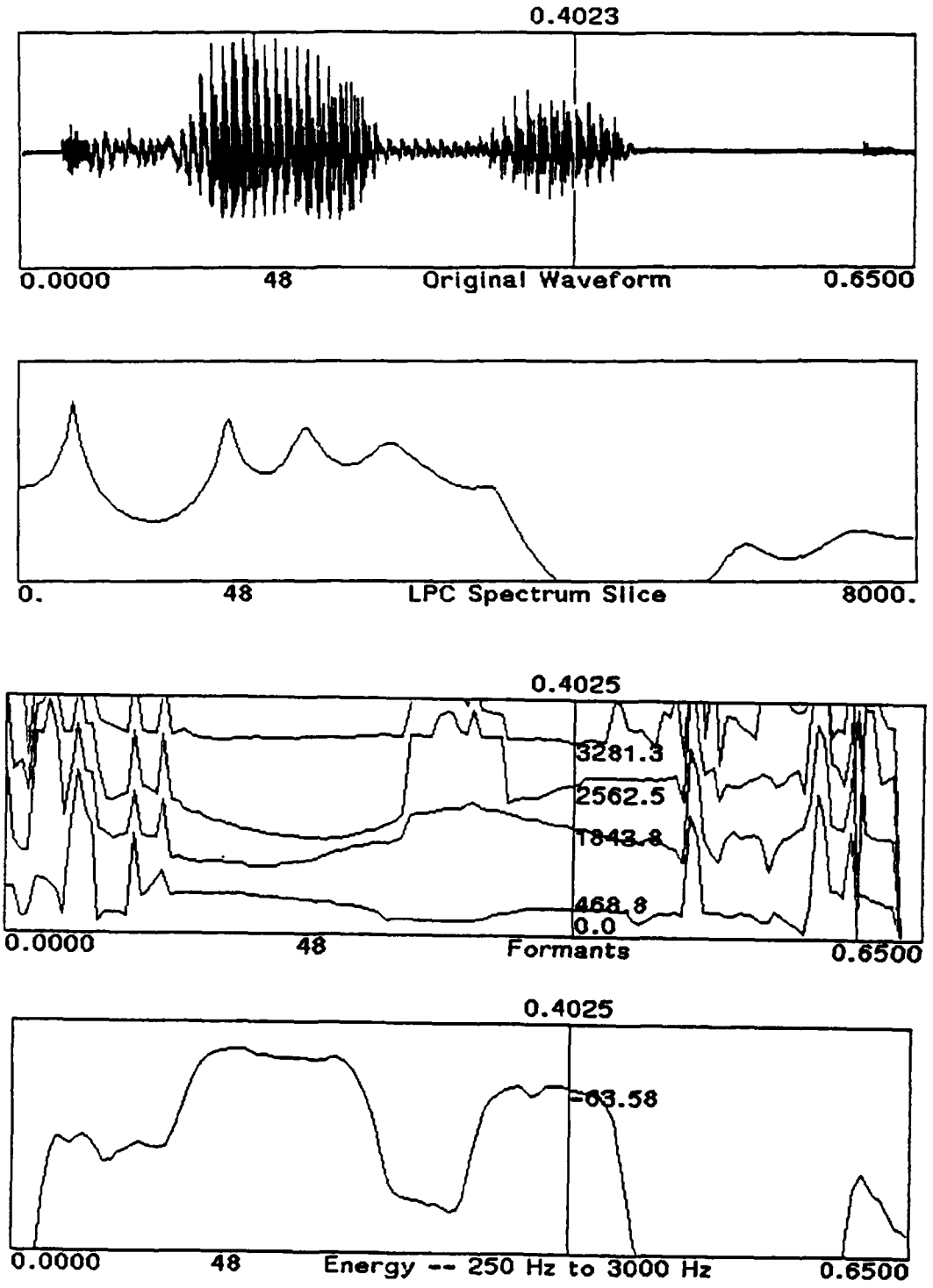

Figure 5. SPIRE Display for Utterance 'TARGET'

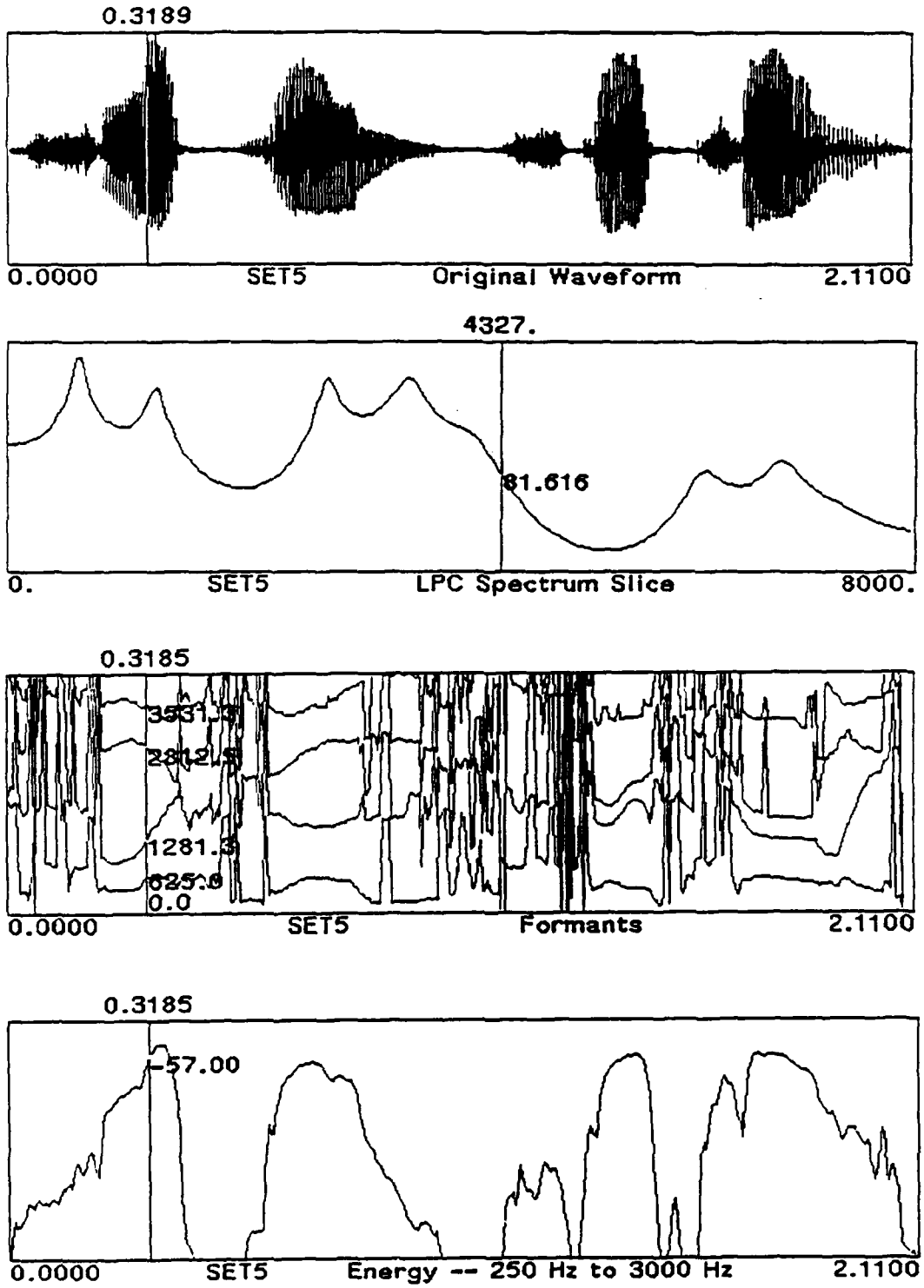

Figure 6. SPIRE Display for Utterance 'SELECT GUN STRAFE CHARLIE'

```
$ on warning then exit
$ run autodtw20b
lib70lfa
b70a_lf
.75
.75
.65
.65
3
.2
0
0
v32a
v2a
$ submit/noprinter/notify/restart/nolog_file dv4a
```

Figure 7. Sample .COM File for Batch Processing

IV. System Design

Introduction

By themselves, the conventional techniques described in chapter II have proven inadequate to the task of mimicking human speech perception. Neural networks show great promise, but further refinement of existing algorithms may be required to keep track of the temporal aspects of speech. The multi-layered perceptron (MLP) falls in this category. New research is attempting to introduce some form of temporal reasoning within a multi-layered perceptron [21:15]. A breakthrough in this area would allow use of the MLP in connected speech recognizers.

A Kohonen neural network offers the advantage of significantly reducing the complexity of the input problem space. Indeed, the recognition system described here reduces the input speech signal from a 15-dimensional vector space to a two-dimensional vector space; therefore, a Kohonen network will reduce the computational complexity of any post-processing system. Dynamic programming is capable of discerning and separating word sounds within connected speech, but the algorithm is computational intensive. A Kohonen neural network and dynamic programming may compliment each other in a hybrid system.

This research effort will attempt to combine a Kohonen neural network with dynamic programming. The Kohonen network will create trajectory patterns for each word. The output of the Kohonen network will then feed into a one-stage dynamic time warping algorithm for both isolated and connected speech.

The system designed in Phase I is identical with the Barmore recognizer. This design provides the vehicle for learning more about the Kohonen structure and its response to speech signals. The Phase II design is the outgrowth of knowledge gained during Phase I testing along with the use of multiple feature sets.

This chapter is divided into five sections corresponding to the main areas of the recognition system. The basic system found in both phases includes the preprocessor, the Kohonen network, and the dynamic programming output section. Areas peculiar to each phase are noted. The last two sections describe functions available only in the final recognition system in Phase II. Section four describes the fusion routine where the LPC and formant based dynamic programming outputs are merged. The rule-based system described in section five is needed in the final system developed in Phase II, since the processing of speech features is asynchronous[1].

Preprocessor

Digitized Speech Processing

Phase I Phase I uses Barmore's original pre-digitized sound files. Each word file is first Hamming windowed using a 3:1 overlapping scheme. Each window is 16 *ms* in duration and begins 5.3 *ms* after the start of the previous window [4:3-3]. A Hamming window is chosen to minimize side lobe distortion and to reduce a smearing effect of the processed frequencies called *leakage* when a Fourier Transform is performed on a data sequence of finite length. Leakage is minimized because the envelope of the Hamming window tapers to a small non-zero value at either side of the Discrete Fourier Transform (DFT) processed sequence of data. Leakage is reduced, but so is the gain envelope at both ends of the Hamming window. Overlapping of windows provides the effect of a more uniform gain term.

Phase II In Phase II, Linear Predictive Coefficients (LPC) instead of a Fast Fourier Transform are used to process the digitized speech. A DFT is a general type of processing tool. Linear Predictive Coefficients, which are based on linear

[1]The processing of speech features is asynchronous in the sense that fusion is performed on features that were processed at different times.

predictive analysis [28:396], are used to predict the next value of a quasi-periodic signal by analysis of a linear combination of past samples. LPC processing is therefore an excellent tool for speech signals. The use of the LPC model assumes the source of the data modeled (the vocal track) is an all-pole linear system. This assumption is generally valid for the vocal track except for nasals and some fricatives [25:138].

Formant Processing The formant frequencies needed to be processed before entering the dynamic programming routine. As seen in Figure 5 and Figure 6, the formant plot shows areas with multiple peaks. These areas represent unvoiced *fricatives* or *stops*. (See Table 2.) The SPIRE formant routine experiences difficulty tracking the formants in these regions because of the broadband fricative energy present in the signal. As a result, inaccurate formant transitions are present in these areas. Although no feature of speech always signals when these regions are encountered, the amount of energy present at the formant frequencies usually is a good indicator of voiced speech. Using an energy frequency range of $250Hz$ to $3000Hz$, the energy content of the signal is usually high during voiced speech and lower during unvoiced speech. A energy value of $-75\ dB$ was chosen as a gate for the formant frequencies. If the energy in the signal was below $-75\ dB$, the formant data were replaced with zeros. For energy values above $-75\ dB$, the formant data were preserved and processed by the dynamic programming routine. An example of formant processing using an energy gate of $-75\ dB$ is shown in Figure 8 for the connected utterance **select gun strafe charlie**. The top plot shows the unprocessed formant data. The middle plot shows the processed formant data using the energy present in the signal. The energy present at the formant frequencies is shown in the bottom plot.

Software routines were written to reduce both $\mathcal{F}1/\mathcal{F}2$ and $\mathcal{F}1/\mathcal{F}2/\mathcal{F}3$ formant data files. The user is prompted for a formant threshold factor, formant range

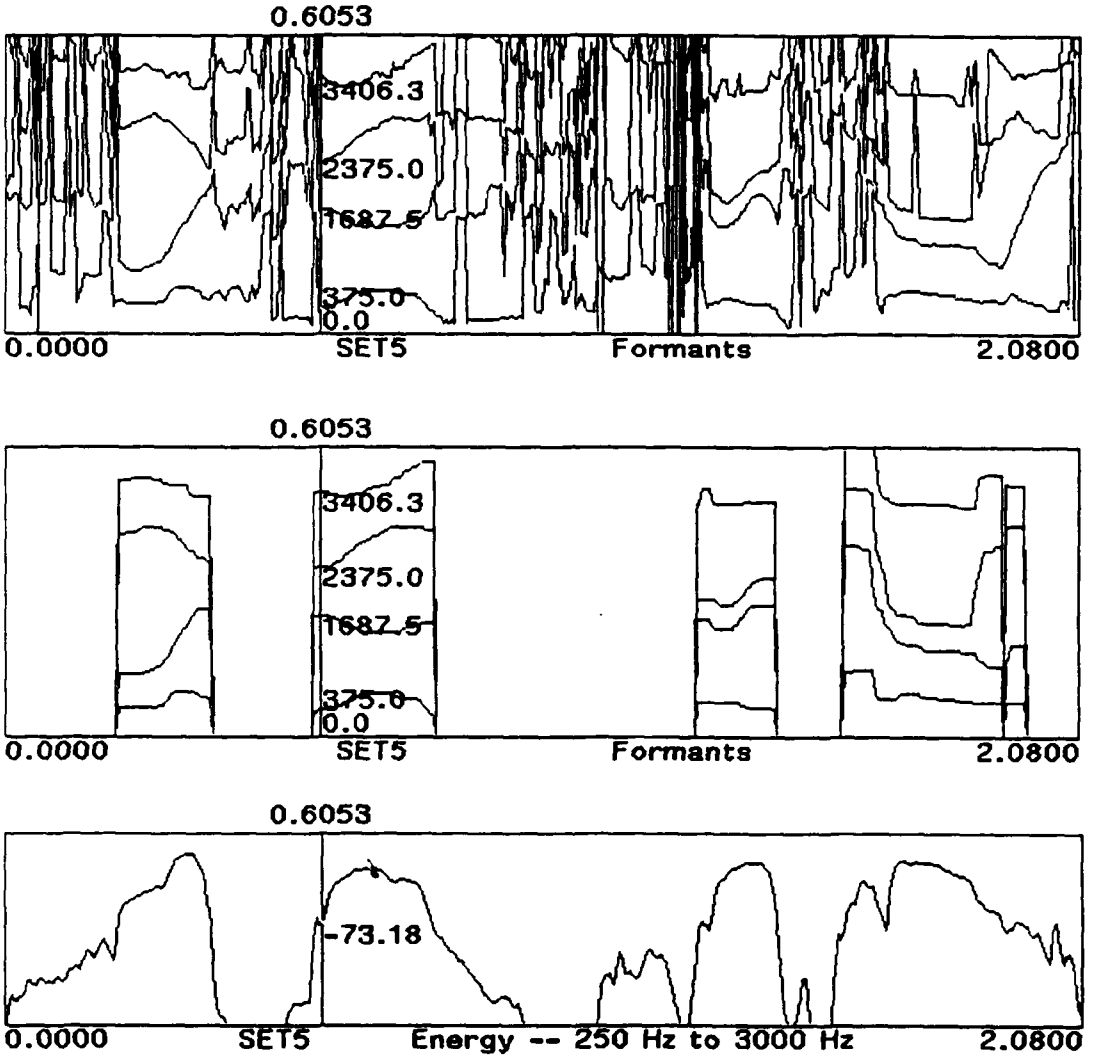

Figure 8: Formant Processing Using Energy Gate for Utterance 'SELECT GUN STRAFE CHARLIE'

factor, and the number of times (number of passes) the reduction routine will process the formant data file. The formant threshold factor refers to the number of data points in a formant data file. The formant range factor refers to the separation in Hertz between two adjacent formant data points within the same harmonic number. The reduction routine evaluates adjacent data points only in formant data files that have a larger number of data points than the formant threshold factor. Each formant is evaluated separately. Within the reduction routine, if the distance between two adjacent data points is less than the formant range factor, the routine deletes one of the data points. The choice of which data point to delete is arbitrary provided the formant range factor is small compared to the distance between adjacent vowel sounds (See Figure 3). Values between $1Hz$ and $25Hz$ were evaluated.

In general, the formant recognition accuracy decreased when formant reduction was used. The reason for the reduced accuracy is probably related to the amount of useful formant information available for each word after initial processing using the energy threshold. Often, only half the word contained useful formant information. In this case, the actual formant file would be half the size of the original data file. Formant reduction deleted additional useful information needed to distinguish individual words. The formant reduction routines may prove valuable if the raw formant data files are processed more efficiently. The routines are available in the final software package.

An attempt was made to process the formant data files with a Kohonen neural network before dynamic programming. Two different normalization schemes were tested. The recognition accuracies were *very* low. The reason for the poor performance was twofold. The range of measured formant data were large, but the important formant data were concentrated within a small range around the first two formants. Normalization decreased the resolution between these two formants because of the wide overall range of the measured formant data. Secondly,

both relative and absolute formant values uniquely determine a vowel-type sound. Normalization has difficulty preserving both attributes. Considering the above research, all formant data files were processed without formant reduction and with dynamic programming only.

Frequency Reduction The resulting 128 frequency components from FFT or LPC processing are then reduced to 15 component slices by frequency reduction. Components of speech at the lower frequencies are accentuated. The rationale is that there are fewer relevant components of speech at the higher frequencies. The pseudo-logarithmic reduction shown in Figure 9 placed only a few low frequency components in the first few bins thereby increasing their overall effect.

Average Subtraction A sequence of 15 component normalized vectors actually represents a trajectory through a 15-dimensional hypersphere [4:3-6]. Because each component is positive, only the positive portion of the hypersphere is used. Subtraction of the average value of individual slices produces both positive and negative vector components within each slice; therefore average subtraction utilizes the entire hypersphere.

Energy Normalization Individual component slices or frames are normalized with respect to their energy content. Energy normalization cancels the effect of gain variations in recording conditions and word volume from one recording session to another.

Kohonen Network

Size The same square shaped Kohonen structure is used in both test phases. The Kohonen net is a two-dimensional surface that consists of 225 output nodes. There are 15 input nodes with each input node connected to every output node via a variable weight.

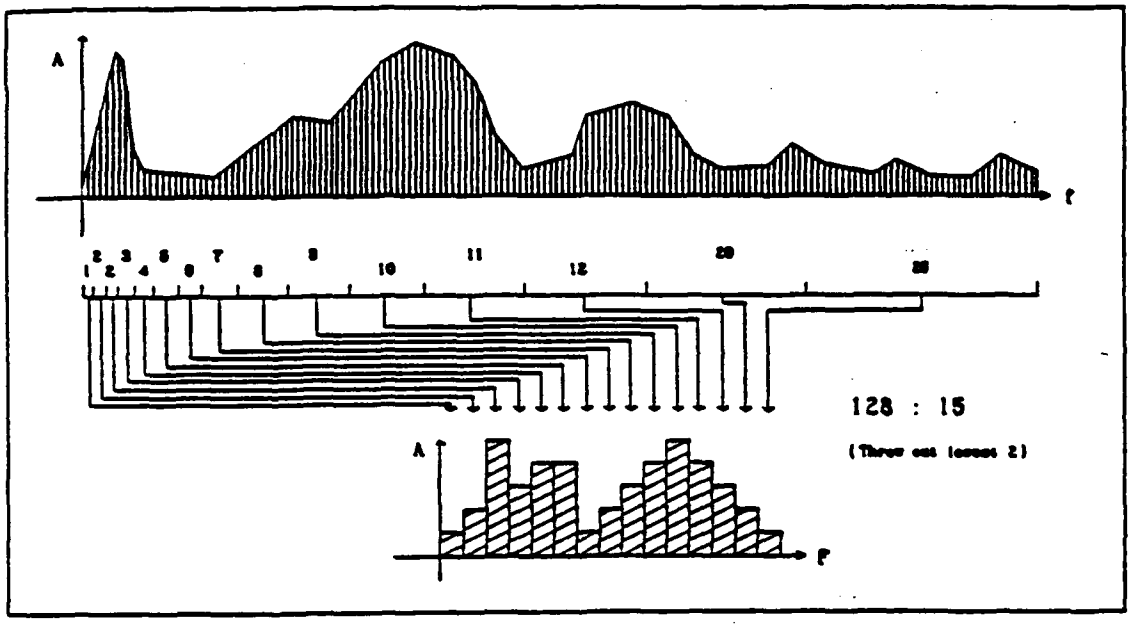

Figure 9: Frequency Reduction 4:3-5

Initialization The same range of randomized weight values are used in both test phases. Weight values are uniformly distributed between -0.05 to $+0.05$ with the actual range and mean dependent on the initial seed value. Barmore used an arbitrary initial seed value[2] of #33 for the published maximum performance results of 99.1% isolated and 90.7% connected speech. Except for test runs starting with a seed value of #33 to verify and compare results in Phase I testing, all tests performed in this thesis effort use an initial seed value of #72.

Training Cycle Times

Phase I A training iteration represents the application of a 15 component slice or frame to the Kohonen network. The network weights within a specified neighborhood are adjusted in response to the input vectors. The network stabilizes and another input is applied to the network. This input represents the next training iteration. Barmore used between 90,000 and 150,000 iterations to train his networks [4:3-8].

A major part of Phase I testing involved determining the minimum iteration time required to train a Kohonen network. The initial assumption was that the recognition accuracy of a Kohonen network would gradually rise to a maximum value determined by the gain reduction method and input data stream presented to the network. Each test in Phase I is run on a series of 13 Kohonen networks. Table 3 lists the number of iterative steps for each network in the series. The 13 networks listed in Table 3 actually represent one network trained for different iteration times, because the same seed value and sequence of training vectors is used.

[2]The pseudorandom number generator within the VAX C language was used in both research efforts.

Phase II The training cycle times in Phase II are a function of the size of the input data file used to train the neural networks. After preprocessing of speech data using the LPC algorithm and pseudo-logarithmic reduction, each data file will consist of a sequence of 15 component frames. The concept of a *multiple*, which is used for Phase II testing, refers to the number of iterations required for the Kohonen network to *see all* of the 15 component frames in the data file used to train the network. The number of iterations required for the Kohonen network to *see* all of the data in a file *once* would be *one multiple*. The total data file size divided by the number of Kohonen input nodes (15) represents the number of iterations required for the network to *see* the entire 70 word data file.

Two 70 word data files are used in Phase II. The 70 word data file used to train the neural networks in Set#1, Phase II testing is 1.2 Mbytes in size and has 139,380 data values. One multiple, (139,380/15), represents 9,292 iterations. Neural nets trained with this data file use multiples or half-multiples of 9,292. The data file used in Set#2, Phase II testing has a multiple of 9,454.

Neighborhoods The concept of a neighborhood is unique to Kohonen networks. While every node weight is updated after each training iteration in a multilayered perceptron, only a small region or neighborhood of nodes centered on the *winning* node receives updating in a Kohonen network. Neighborhood size is variable and time dependent. Neighborhoods start out large (relative to total net area), and generally diminish in size as training progresses. Often neighborhood reduction is coupled to the type of gain reduction employed. All five methods are tested in Phase I while the Exponential method only is used in Phase II.

Gain Reduction The gain schedule or rate of decay of $\alpha(t)$ during each iteration affects how much a node's corresponding weight is adjusted. Kohonen states that many scheduling laws can be used with his network including linear, piecewise-linear, and exponential reduction [18]. Questions arise as to how each reduction

method performs with speech input and dynamic programming programming as a word classifier. Research needs to quantify the performance of the different gain reduction methods, and determine if there is a best gain reduction method for the recognition system used in this thesis. Current literature offers little guidance in this area.

The starting α value of a scheduling method also needs consideration. Kohonen suggests that α start at a value close to unity with a gradual reduction over time to a value less than 0.01 [19]. Again, published research to date does not help answer the question about the effect of α on recognition accuracy. Another major goal of Phase I is to evaluate the performance of various gain scheduling methods.

A total of five methods are evaluated in Phase I. Two of the five reduction methods are linear reduction methods. In this thesis, they are referred to as Linear Type #1 and Linear Type #2. The Linear Type #1 and Piecewise-Linear methods were adapted from the Barmore thesis. Barmore used the Piecewise-Linear gain reduction method with starting gains of 0.1/0.01 in his final recognition system. He achieved recognition accuracies of 99.1% with isolated words, and 90% recognition accuracy with connected speech. The Linear Type #2, Exponential, and Central-Adaptation methods were adapted from research presented at the 1989 International Conference on Acoustics, Speech and Signal Processing (Glaslow, Scotland)[6]. Because of similar neighborhood reduction scheme, the Linear Type #1 and Piecewise-Linear methods are discussed separately.

Linear Type #1 and Piecewise-Linear For these two methods, node weights within the neighborhood are updated according to the following equations:

$$\begin{aligned} m_{i,j}(t+1) &= m_{i,j}(t) + \alpha(t)(x_j(t) - m_{i,j}) && \text{if Node i} \in N_c(t) \\ m_{i,j}(t+1) &= m_{i,j}(t) && \text{if Node i} \notin N_c(t) \end{aligned} \tag{3}$$

where:

m : weight connecting output node i to the j_{th} component of input x

α : gain factor

t : current iteration time

for the two reduction methods:

$$\text{Linear Type \#1} \begin{cases} N_x(t) &= NX_{min} + [(NX_{max} - NX_{min} + 1)(1 - t/T)] \\ N_y(t) &= NY_{min} + [(NY_{max} - NY_{min} + 1)(1 - t/T)] \\ \alpha(t) &= C(1 - t/T) \end{cases} \tag{4}$$

$$\text{Piecewise-Linear} \begin{cases} if\ t &< t_m \\ N_x(t) &= NX_{min} + [(NX_{max} - NX_{min})(1 - t/t_m)] \\ N_y(t) &= NY_{min} + [(NY_{max} - NY_{min})(1 - t/t_m)] \\ \alpha &= \alpha_s(1 - t/t_m) \\ \text{else} & \\ N_x(t) &= NX_{min} \\ N_y(t) &= NY_{min} \\ \alpha &= \alpha_m(1 - t/T) \end{cases} \tag{5}$$

where:

NX_{min} : minimum neighborhood size in x direction

NX_{max} : maximum neighborhood size in x direction

NY_{min} : minimum neighborhood size in y direction

NY_{max} : maximum neighborhood size in y direction

C : constant set to 0.99, 0.5 or 0.1

α_s : start gain value of 1^{nd} linear portion

α_m : start gain value of 2^{nd} linear portion

t_m : start cycle time of 2^{nd} linear portion

T : total number of iterations

A graph showing a typical Linear Type #1 gain reduction of α is shown in Figure 10. Piecewise-Linear gain reduction is a special case of Linear reduction. It consists of two linear curves with the second curve starting after the first curve ends. Kohonen asserts that training of the net or *learning* is a two step process [18]. In the first step, topological ordering of the network takes place. This initial iterative period lasts from "a few hundred" to the "first 1000 steps or so" [19]. Kohonen believes that α should stay above 0.1 during this period. The final or convergence step is when the network is "fine tuned" [18]. This last step is of long duration. The Piecewise-Linear reduction method is designed to accommodate the two-step learning process. Figure 11 shows an example of Piecewise-Linear gain reduction where the second linear curve starts when 80% of the total cycle time remains ($t = 0.8T$). This thesis effort will test initiation times of 20% and 80% for the second linear curve.

The neighborhood for both methods is square in shape with a starting horizontal and vertical node radius specified by the user. Each neighborhood decreases in a linear fashion to a size specified by the user. In all cases, the starting neighborhood radius is "7 by 7", and the final neighborhood size is "1 by 1." The final neighborhood size contains nine nodes centered about the last *winning* node. These start and end neighborhood sizes were used by Barmore [4:3-9]. They are used here for easier comparison of results.

Three series of 13 Kohonen networks are trained for both Linear Type #1 and Piecewise-Linear reduction methods. A different starting gain value is used for each of the three series. The initial gain values used are 0.99, 0.5 and 0.1.

Linear Type #2, Exponential and Central-Adaptation The Linear Type #2 and Exponential methods use the same gain reduction equation with a sample gain reduction graph shown in Figure 12. These two methods are differentiated only by the schedule used to reduce the neighborhood size. In each case, the initial gain

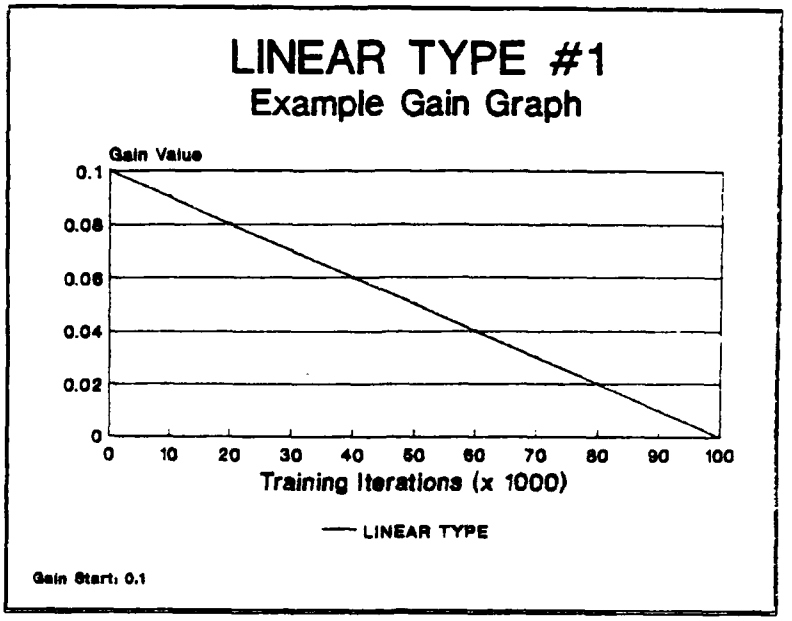

Figure 10. Linear Type #1 Gain Reduction

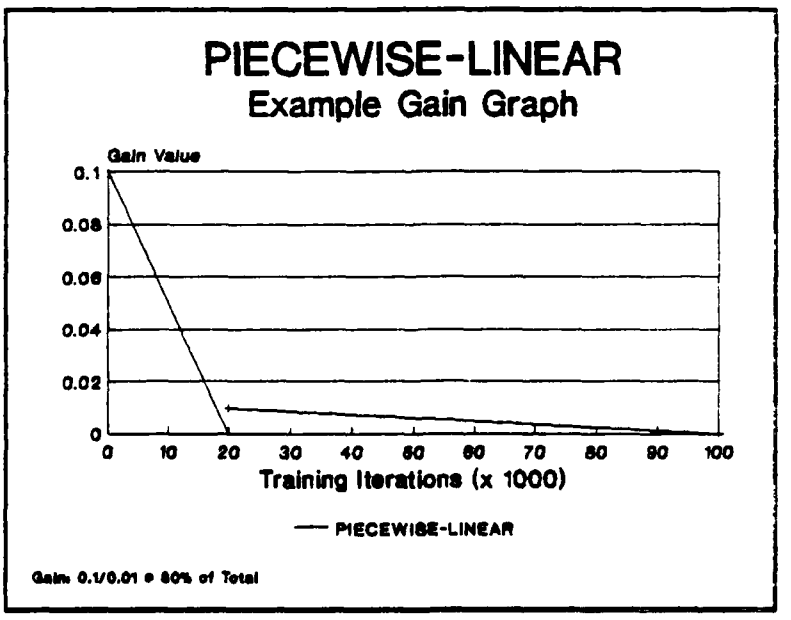

Figure 11. Piecewise-Linear Gain Reduction (2nd Gain Start: 80% of Total)

value is 0.1. Figure 13 graphically shows how each neighborhood is reduced in size.

In the previous methods, the gain value applied during the updating of each node in a neighborhood is held constant. A node next to the *winning* node and the nodes at the periphery of the neighborhood use the same gain multiplier. In the Central-Adaptation method, the amount a node is updated depends on its distance from the *winning* node. All nodes are eligible since the neighborhood remains at a constant size.

For each of these three methods, the node weights are updated according to equation 3 if:

$$r_{\nu,J} \leq R(t) \tag{6}$$

where:

$$r_{\nu,J}^2 = (x_\nu - x_j)^2 + (y_\nu - y_j)^2 \tag{7}$$

and for each of the three methods:

$$\text{Linear Type \#2} \begin{cases} R(t) &= R_d + (1 - R_d)t/T \\ \alpha &= C(1 - t/T) \end{cases} \tag{8}$$

$$\text{Exponential} \begin{cases} R(t) &= R_d^{(1-t/T)} \\ \alpha &= C(1 - t/T) \end{cases} \tag{9}$$

$$\text{Central-Adaptation} \begin{cases} R(t) &= R_d \\ \alpha &= 2C(1 - t/T)e^{-r_{\nu,J}[1.4/R_d + t/T(5.6 - 1.4/R_d)]} \end{cases} \tag{10}$$

where:

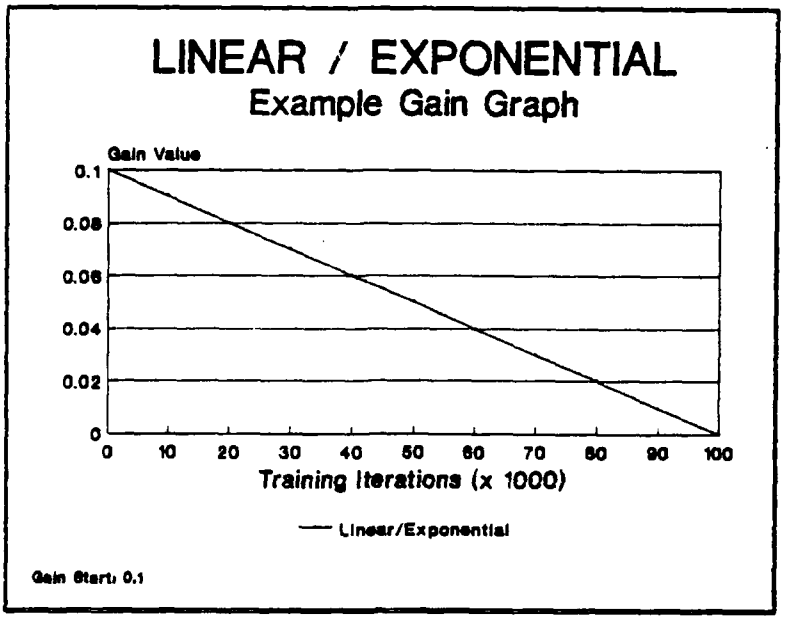

Figure 12. Linear Type #2 and Exponential Gain Reduction

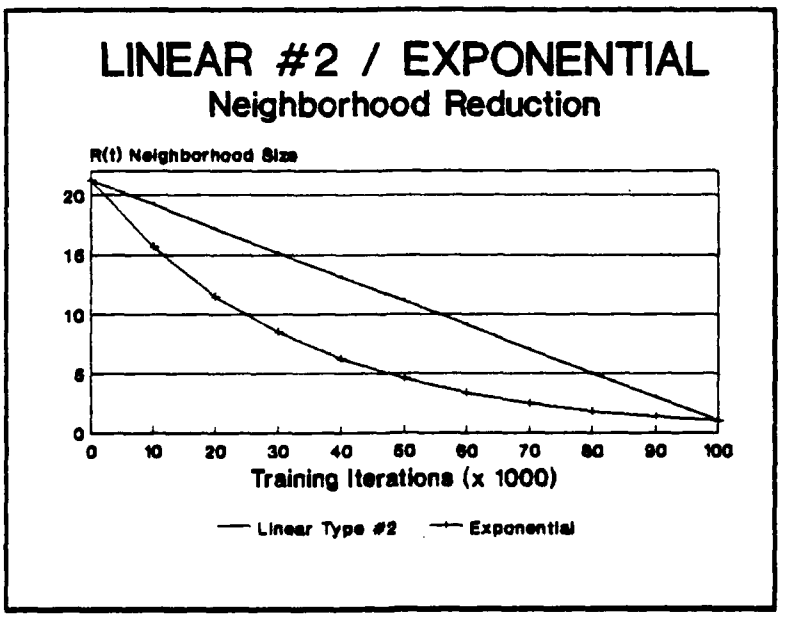

Figure 13. Linear Type #2 and Exponential Neighborhood Reduction

$$\begin{aligned} C &: constant = 0.1 \\ R_d &: \sqrt{x^2 + y^2} \end{aligned} \tag{11}$$

All three methods use a surface constant R_d which will vary only if the size of the network changes. With a 15 $\times$ 15 size neural network surface, R_d is 21.21 from Equation 11. The neighborhood cannot be varied by the user with any of these methods. It is time dependent on the current iteration count for the Linear Type #2 and Exponential methods and equal to R_d in the Central-Adaptation Method.

Conscience During each training iteration, the *winning* node is defined solely by the distance between the node's weight vector and the input vector computed during that cycle. The number of times a particular node may have won during past iterations is not considered. Depending on the input data and initial node weight values, it is possible for a group of nodes to dominate as *winning* nodes during the course of a training session. Other nodes may rarely if ever win during the same session causing underutilization of the Kohonen surface. Underutilization of the Kohonen nodes can cause insufficient resolution of adjacent sounds. Any output recognition stage may not be able to differentiate these sounds.

To alleviate the problem, the concept of conscience (the memory of the number of past wins by a particular node) was introduced. There are different methods to select the conscience threshold [4,8]; the method used in this thesis is the same method used by Barmore. When training a Kohonen network with conscience, before a node is allowed to be a *winning* node, the number of times the node has won in the past must be less than the conscience value according to the following equation [4:3-10]:

$$c_i(t) < \beta t/n \tag{12}$$

where c is the number of times node i has won prior to the current iteration t, n is the total number of nodes on the Kohonen surface, and β is the conscience factor. As the conscience factor β decreases, network conscience increases. Barmore found that a conscience factor of 1.5 gave the best performance, and a conscience factor much above 1.5 caused the network not to implement any conscience [4:3-10].

Barmore compared results using conscience factor values of 1.1, 1.5, and no conscience [4:4-9]. The concept of conscience is intriguing, and, because of its effect on dividing the input vector space into discrete regions within the Kohonen surface, further study is warranted. Research needs to investigate the response of a Kohonen neural network to different values of conscience, and determine if a conscience factor other than 1.5 would provide better recognition accuracy.

A major part of Phase I was to determine the response of a Kohonen network to different values of conscience. In the process, an attempt was made to find an optimum conscience factor value for the speech recognition techniques used in this thesis effort. Only two gain reduction methods were tested due to research time constraints. The Piecewise-Linear method was evaluated using a seed value of #33 and a 90K/18K training schedule, because this combination was used by Barmore in his thesis. The Exponential gain reduction was evaluated using a seed value of #72, because this combination was used in the final speech recognition system in Phase II of this thesis.

Trajectory Reduction An individual word is composed of approximately one hundred[3] 15 component frames with each frame represented by a *winning* node on the Kohonen surface. A word trajectory pattern represents the connection of each *winning* node in time sequence. Typically, the trajectory will represent a transition through each sound type within the word from beginning to end. Also, the trajectory will pause at major sounds in the word depending on the length

[3]Assume an average word length of 500 *ms*.

of the sound. For example, there would be an extended pause in the area of the Kohonen surface representing the *e* sound in the word *three*.

In practice, a unique trajectory is observed for each word as the word transitions through its sound types. Superimposed on this pattern, however, is a random one or two frame wander of the trajectory to different parts of the Kohonen surface. Kohonen observed this phenomenon in the development of his Phonetic Typewriter [18]. Barmore developed a method that deletes the random wander within the trajectory pattern. Each vector set representing a trajectory is passed through a two-step reduction algorithm. In the first step, a point on the trajectory, not within two node units of another point on the trajectory within two time slices, is eliminated. In the second step, all points on the trajectory which are not part of three consecutive points are eliminated [4:3-12].

Dynamic Programming

The final stage in the recognition system is the word classifier using dynamic programming or dynamic time warping. The algorithm used in this thesis is a one-stage dynamic programming algorithm adapted from Ney [24]. Ney's paper is tutorial in nature. He describes the original dynamic programming algorithm formulated by T. K. Vintsyuk in 1971, and various derivatives of the basic algorithm. Ney's one-stage algorithm is the same as Vintsyuk's original algorithm except for a more efficient use of computational resources. The algorithm is optimized for connected speech; it is the same algorithm used in several previous thesis efforts at AFIT [7,14,4].

Dynamic programming is essentially a rate or time normalization operation, in which an unknown word is stretched or compressed in time until it is the same length as the reference or library word [25:297]. In describing the operation of dynamic programming in graphical form, the library or reference words are usually put along the vertical axis; and the unknown test utterance is situated along the

horizontal axis. Using Ney's notation, the library of words are referenced by the index $k = 1, \ldots, K$, where K is the total number of words in the recognition library. Each word in the library is referenced by the index $j = 1, \ldots, J(k)$ where k is the length of library word k. The unknown utterance consists of N time frames equal to the reduced vector trajectory output of the Kohonen network, where the pattern is referenced by the index $i = 1, \ldots, N$ [24:264].

For purposes of discussion, the simplest case has one unknown test word compared with two library words as shown in Figure 14. Dynamic programming computes a distance function $\mathcal{D}$ as each point i in the test word is compared with each point j in each library word where $j = 1, \ldots, J(k)$ and $k = 0$ or 1. The distance function $\mathcal{D}$ is the absolute valued difference between the library and unknown word. For each word, the algorithm keeps track of all previous values of the distance function from the start point S to the end point E. The minimum distance function, min $\mathcal{D}$, represents the best match between the test word and both library words. The library word associated with the minimum distance function is the library word output from the recognition system. The above discussion can easily be extended to K library words where $K = 70$ in this thesis effort.

Ney's algorithm extends dynamic programming in the isolated-word case described above to connected speech. The algorithm determines the sequence $q(1), \ldots q(R)$ of library words which represents the minimum distance function for the test utterance consisting of R words [24:264]. Dynamic programming of connected speech is shown graphically in Figure 15 with a library set of ten words and an input pattern consisting of seven words.

The algorithm uses two types of transition rules for recognition of each word within a multi-word sequence. In the word's interior, there are within-template transition rules, and at word boundaries there are between-template transition rules. The algorithm allows the distance function to travel either up, to the right,

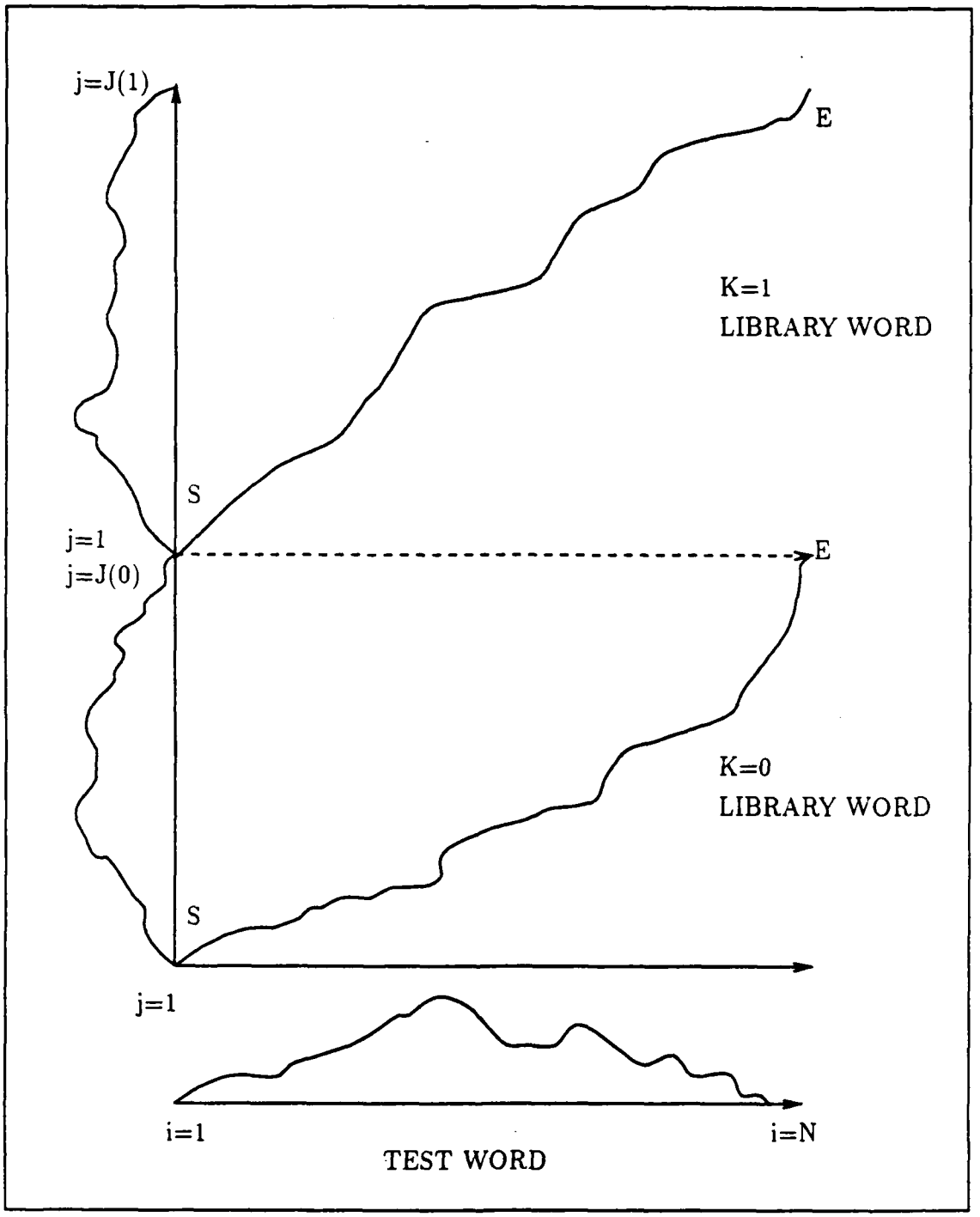

Figure 14. Dynamic Programming Example (Isolated Speech)

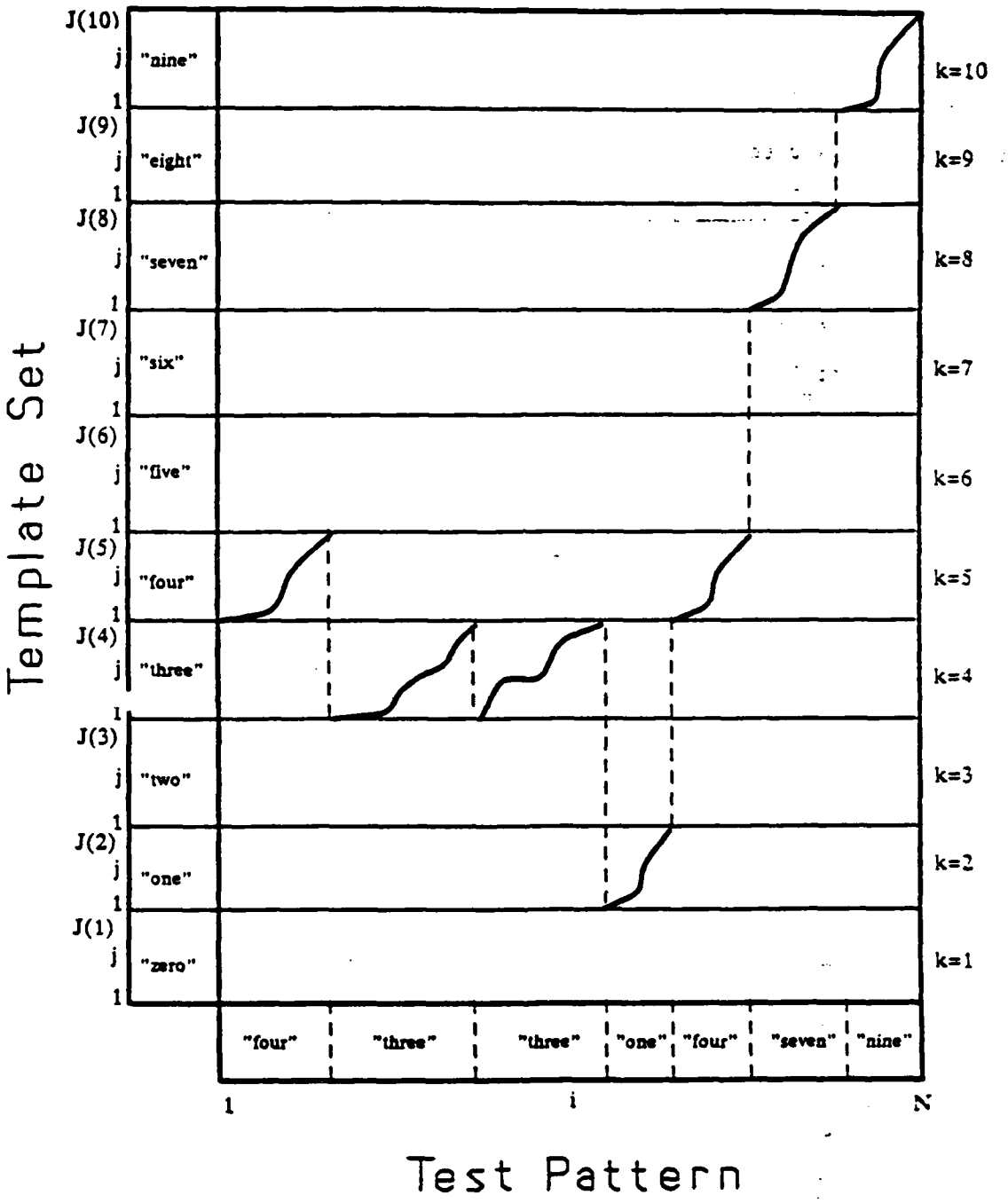

Figure 16: Dynamic Programming Example (Connected Speech) 6:3-12

or diagonally to the right from its present position. The distance function can never travel to the left or down. For isolated word recognition or within a word interior, (see Figure 16), the within-template transition rules apply. For connected word recognition at the word boundaries, the point J(1) at the boundary of the new word as seen in Figure 17 can be reached from the ending frame of any library word k^* including k itself [24:265].

Ney introduced time distortion penalties if the distance function traveled to the right (utterance is longer than the library word at point i), or the distance function traveled up (utterance is shorter than the library word at point i). The computed distance function $\mathcal{D}$ is multiplied by the appropriate distortion factor. Barmore used penalty factors of 0.75 for both vertical and horizontal movement of the distance function, and a factor of one for diagonal movement. The distortion factors used in this thesis were 0.75 for dynamic programming using LPC data, and 0.65 using formant data.

Ney determined that only two arrays of data were needed to recover the unknown word or words within the input sequence. For each point i in the test utterance, one array keeps track of the library word that has the minimum cost function computed at point i. The other array is a backpointer that marks when this library word began by keeping track of when the between template transition rules were used for that word. In the isolated case, the second array always begins at $i = 0$.

The input to the dynamic programming routine can be any attribute of speech. In this thesis, two types of speech features underwent dynamic programming. They are the reduced vector trajectories from the Kohonen network trained with LPC processed speech and raw formants from the utterance under test and library words. Both $\mathcal{F}1/\mathcal{F}2$ and $\mathcal{F}1/\mathcal{F}2/\mathcal{F}3$ formant sets are tested using dynamic programming.

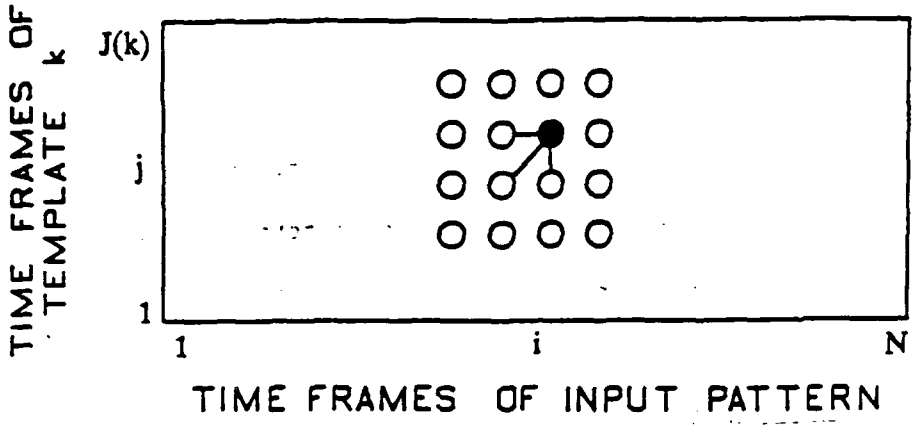

Figure 17: Within Template Transition Rules 6:3-13

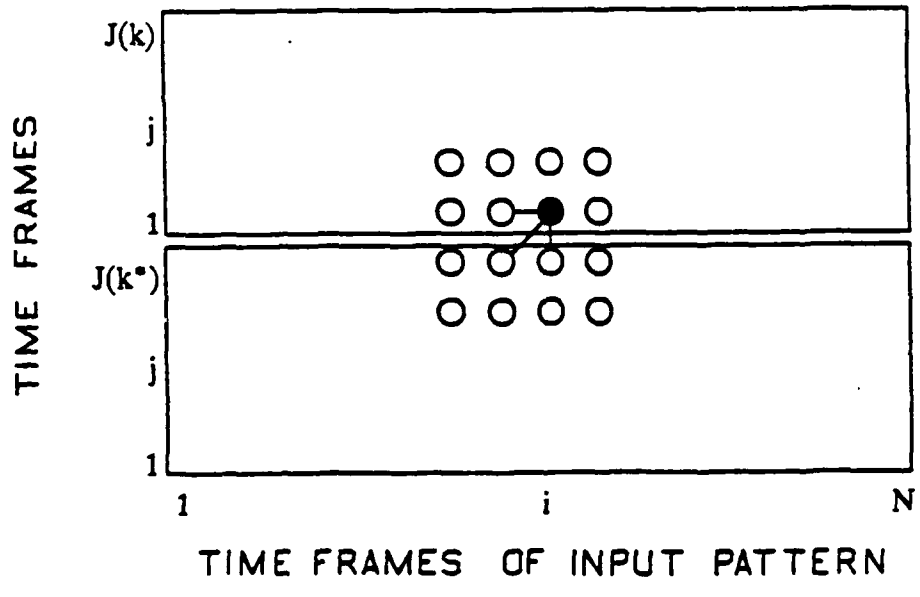

Figure 18: Between Template Transition Rules 6:3-13

Feature-Fusion Section

Two distinct outputs were produced from the dynamic programming section for each utterance. They corresponded to the LPC-based output and the formant-based output, where dynamic programming of each feature was performed at separate times (asynchronously). Either the LPC or formant feature of speech may have performed better for a particular word depending upon the sounds within the utterance. Words are never spoken the same way each time. The word spoken may not have the minimum distance function $\mathcal{D}$, because the word was spoken in a slightly different manner than the way in which the template words were spoken; in this case, the actual distance metric for the word was usually close to the minimum. The fusion routine found the best match between the two features of speech (LPC and formant) at any point i in the utterance that dynamic programming had specified as the end of a word. The best match from the feature-fusion routine was the final system output.

The feature-fusion routine treated the LPC and formant data with equal weight. It was not possible to just add together the two dynamic programming results because the LPC data were in processed form before dynamic programming, and the formant data were inputed to the dynamic programming routine in raw unprocessed form. The actual distance metric produced by the dynamic programming section for each feature could vary by a factor of 20 or more which would preclude simple summation.

The fusion process was performed in two steps. First, at each point i of the utterance that dynamic programming specified as the end of a word using the backtracking function, the measured distance for each library word at that point was divided by the minimum distance at that point. As a result, within each array the minimum distance factor was then normalized to 1.000 with all other distance values greater than 1.000. N arrays corresponding to N words from the test utterance were then available for fusion. Fusion took place in the second step.

In step two, the normalized distances of each corresponding LPC and formant word were multiplied. The end result was a fused LPC/Formant distance metric for each word.

Fusion was not performed on every word. The reason was two-fold. A *winning* word with an LPC dynamic programming distance that was much smaller than the other words in the library was probably the correct answer. Fusion was not required in that case. Fusion could produce the incorrect answer if the corresponding formant dynamic programming distance used in the fusion process was high relative to the other formant distances. Secondly, it was possible that the LPC based dynamic programming distances would be close in absolute value to the *winning* word. In that case, the fusion results were based solely on the formant results. If the correct word had a relatively high formant dynamic programming distance, the recognizer would output the wrong word. The LPC results were given added emphasis by allowing only words close in LPC dynamic programming distance to the *winning* word into the fusion process. The recognition system would occasionally output the correct word solely due to that *windowing* effect.

A threshold value was used to produce the desired size *window* or group of words allowed into the fusion process. In this thesis, three fusion thresholds were tested. They were 1.5, 2.0 and 2.5. For example, when the 1.5 fusion threshold was specified, each normalized distance metric in the LPC normalized array between 1.000 and 1.500 was fused with its matching normalized formant distance metric.

An example using the word **degrees** is shown in Figure 18. The following notation is used in the recognition system display output.

- For *.trn* files, each file has three items of information.

  1) The first three letters specify the type of file.

  LPC for lpc file and FMC for formant file.

  2) The #2 in **wr2** denotes the second of five series of

words. Each series contains 70 words.

3) The last number in the prefix of the file name references a particular word. (i.e. wr2_**34** for the word **degrees**) See Table 2 for a complete listing.

- The result from the recognition system and the correct word are given numerically according to Table 2.
- The *best choices* displayed by the recognition system provide the following information:

  1) Word.

  2) Distance value computed from dynamic programming (DP).

  3) Distance factor (DP distance / minimum DP distance).

- **LPC-F1/F2** denotes feature-fusion of the first two formants with the LPC feature.
- **LPC-F1/F2/F3** denotes feature-fusion of the first three formants with the LPC feature.
- **The # of words less than** represents the number of words eligible for feature-fusion.

The displayed output for the word **Degrees** is shown in Figure 18. For each word, there is an LPC and formant result determined and output to the screen. The percentage of correct words for each feature are continually updated. Within the feature-fusion output area, three results are continually updated corresponding to the three feature-fusion thresholds. In addition to the best match output result, the system provides the top n choices, where n is a value specified by the user or the position of the actual word, whichever is larger. For each word, its actual dynamic programming distance and its normalized distance is output to the screen.

```
 lpc7_wr2_34.trn is:  34
   Should be:  34
    Best choices are:
     DEGREES had a lpc distance of 90.25 and a distance factor of 1.0000
     SPACE had a lpc distance of 242.50 and a distance factor of 2.6870
     EAST had a lpc distance of 276.25 and a distance factor of 3.0609
   correct = 1.000    cum_correct = 0.971
   Total wrong lpc words are 1.00 from a total of 34.00 words

fmc7_wr2_34.trn is 174 vectors long
 fmc7_wr2_34.trn is:  34
   Should be:  34
    Best choices are:
     DEGREES had a formant distance of 6614.00 and a distance factor of 1.0000
     THREE had a formant distance of 28602.80 and a distance factor of 4.3246
     EAST had a formant distance of 29239.25 and a distance factor of 4.4208
   correct = 1.000    cum_correct = 0.824
   Total wrong formant words are 6.00 from a total of 34.00 words

The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
The computed utterance string is:  34
    Should be:  34
    The # of words less than 1.5 are 1
    Best choices are:
       DEGREES had a formant-lpc distance factor of 1.0000
   correct = 1.000    cum_correct = 1.000
   Total wrong fusion words are 0.00 from a total of 34.00 words

The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
The computed utterance string is:  34
    Should be:  34
    The # of words less than 2.0 are 1
    Best choices are:
       DEGREES had a formant-lpc distance factor of 1.0000
   correct = 1.000    cum_correct = 0.971
   Total wrong fusion words are 1.00 from a total of 34.00 words

The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
The computed utterance string is:  34
    Should be:  34
    The # of words less than 2.5 are 1
    Best choices are:
       DEGREES had a formant-lpc distance factor of 1.0000
   correct = 1.000    cum_correct = 0.971
   Total wrong fusion words are 1.00 from a total of 34.00 words
```

Figure 18. Feature Fusi· ·f Speech 'DEGREES'

Rule-Based Section

Dynamic programming on the LPC and formant features of each word are performed asynchronously. Sometimes the number of words determined from LPC based dynamic programming is different than the number of words determined from formant based dynamic programming. This word count mismatch is found more often with connected speech, but occasionally it is found with multisyllabic words spoken in isolation. The rule-based system was developed to align the two dynamic programming results prior to the feature fusion process, because the feature-fusion routine requires an equal number of words from each feature.

The coding scheme for the rule-based system is extensive because it must anticipate every possible mismatch between the LPC and formant based outputs. To preclude a rule-based system of infinite size, a bound was set as to the number of words the rule-based system can process. The rule-based system will not process utterances with lengths greater than seven. For unequal word count outputs from dynamic programming greater than seven, feature fusion is not performed. If the word count is the same for the two features, fusion is performed for any number of words in the test utterance.

The rules within the rule-based system are specific and unique for the speech recognition environment created in this thesis effort. These rules were determined after evaluating thousands of dynamic programming results. The rule-based system, which evaluates the LPC and formant results, is shown in pseudo-code format below.

- **Initial Rules Prior to First Feature-Fusion Calculation.**
- Perform feature-fusion only if:

  (1) LPC words == Formant words OR

  (2) If LPC words not equal Formant words

then only if:

number LPC words AND number Formant words are less than 7.

- If number of LPC words == 2 AND number Formant words == 1

  then:

  If one of the LPC words matches the Formant word,

  then:

  1) delete other LPC word.

  2) set number LPC words == 1.

- If number of LPC words == 2

  then:

  If one of the LPC words is a *space (pause)*

  then:

  If DTW distance of *space* is greater than 200 AND

  other LPC distance are less than 120

  then:

  1) delete *space* as an LPC word.

  2) set number LPC words == 1.

- If LPC and Formant features are equal.

  then:

  perform feature-fusion.

  else:

  continue rule-based evaluation of features.

- **Isolated-Word Mismatch Rules.**

- If number LPC words == 1 AND

  number Formant words == 2 OR 3.

Divide all Formant DTW distances except the minimum Formant DTW distance by 2.5. Determine if the minimum Formant distance is still minimum when compared to the other Formant distances. (If yes, the minimum Formant DTW distance is probably the correct word.)

If yes:

1) Delete the other Formant word(s).

2) Number LPC words == 1 AND number Formant words == 1.

3) Perform feature-fusion on equal number of words.

If no:

1) Add normalized Formant DTW distances for each word.

2) Perform feature-fusion.

- If number of Formant words == 1

  AND number LPC words == 2 OR 3.

  Evaluate the LPC DTW distances. If the smallest DTW distance is less than 120 and the other DTW distance(s) is(are) greater then 120, then:

  1) Delete the other LPC word.

  2) Number LPC words == 1.

  3) Perform feature-fusion.

  If not true:

  1) Add LPC normalized DTW distances for each word.

  2) Perform feature-fusion.

The remaining rules cover connected-word mismatches. The general rule assumes the first and last words between both features of speech are a good match, because the effects of coarticulation are less for the beginning and ending words. Matching the inner words within a connected utterance was much more

difficult, because there were not any guidelines to follow. The decision rules for each connected-word case is given in general form below.

- **Connected-Word Mismatch Rules.**
- 1) Delete pauses from LPC results if number LPC words is greater than number of Formant words.
  2) Match first word of each feature.
  3) Match last word of each feature.
  4. Match number of Formant words to number of LPC words if the word mismatch is less than 3.
  5. If word mismatch between features is greater than 3, determine a best guess for the actual number of words which is somewhere in between the two word counts.
  6. Working from the beginning of the utterance and the end of the utterance, match the feature with the smaller number of words to the feature with the larger number of words by creating and distributing duplicate words.
  7. The end result will have features with an even number of words.
  8. Perform feature-fusion.

The rule-based system represents a first attempt at matching two features. It is a baseline system. Refinements may be necessary as testing continues.

Conclusion

This chapter discussed in detail the five main parts of the speech recognition system. The basic system consists of the preprocessor, the Kohonen neural network, and the dynamic programming word classifier routine. This system is used

in Phase I. The final recognition system tested in Phase II consists of the basic system, plus the feature-fusion and rule-based routines. The next chapter provides the results of testing both systems.

Table 3. Phase I Training Schedule

Net #	*Training Iterations*
1	5,000
2	10,000
3	20,000
4	30,000
5	40,000
6	50,000
7	60,000
8	70,000
9	80,000
10	90,000
11	100,000
12	110,000
13	120,000

V. Test Results

Introduction

This chapter presents the test results from Phase I and Phase II. Although the goal of each phase was different, the testing philosophy was the same. In accordance with that philosophy, all tests were performed using at least a basic recognition system consisting of a preprocessor section, Kohonen neural network, dynamic programming word classifier, and a large representative group of test words. In each phase, the words used to train the Kohonen network were different than the words used to test the network.

As with any test, it is important to change only the variables under test. For this thesis, parts of the system not directly under test were held constant. Parts of the system unchanged during both test phases, except where specifically noted, include:

- the initial seed value for the Kohonen network.
- the range of initial weight values for the Kohonen network.
- the use of average subtraction in the preprocessor section.
- the LPC frequency reduction scheme in the preprocessor section.
- the size of the Kohonen network (15×15).
- the presentation of data to the Kohonen network (sequential).
- the same post-Kohonen network trajectory reduction method.
- the LPC vertical and horizontal distortion penalties (0.75).
- the Formant vertical and horizontal distortion penalties (0.65).
- the same grading method used by Barmore.

This chapter is divided into two main sections. The first section discusses the tests performed in Phase I. The goal of Phase I was to learn more about the Kohonen neural network, and in the process, to optimize the recognition system for Phase II. The last section discusses Phase II testing of the final recognition system.

Phase I

Introduction In Phase I, ten sets of ten digits were used in each test. These words are listed in Table 4. Table 5 lists the connected utterances used in Phase I. As previously noted, these are the same words used in the Barmore thesis.

Table 4. Phase #1 Test Words (Isolated Speech)

zero	five
one	six
two	seven
three	eight
four	nine

Gain Reduction A large number of tests were run to find the response of a Kohonen network to the five different gain reduction methods. Starting with the Linear Type #1 method, the results of each method are presented along with a discussion of the results. A conscience value of 1.5 was used for all gain reduction tests.

Linear Type #1 Gain Reduction Three series of 13 neural networks were trained using the Linear Type #1 reduction method. The training iteration cycle times are listed in Table 3. Each series started with a different α value.

The results for isolated and connected speech using this method are shown in Figure 19 and Figure 20 respectively. Each graph shows the percentage of words

Table 5. Phase I Test Words (Connected Speech)

two eight two eight two eight
two four six eight
two eight three one eight
zero one two three four five six seven eight nine

correctly identified for each of the trained neural networks.

The results show that as the initial starting gain is decreased from 0.99 to 0.1, the recognition accuracy increases significantly. An initial starting gain of 0.99 gave the worst recognition accuracy. These results appear to contradict Kohonen's assertion that the initial starting gain value should be near unity [19:17]. An initial starting gain of 0.1 gave the best recognition accuracy for both isolated and connected speech. Table 6 shows the range and average percentage of words correctly identified for each of the initial gain values. In general, networks using Linear Type #1 gain reduction that trained below 70,000 iterations gave better recognition accuracy.

Table 6. Linear Type #1 Recognition Accuracy

Starting Gain	*Range (%)*		*Average (%)*	
	Iso	*Con*	*Iso*	*Con*
0.10	84-99	76-92	92	86
0.50	76-94	72-92	86	86
0.99	64-86	64-96	78	83

Piecewise Linear Gain Reduction Piecewise-Linear gain reduction introduces two variables not present with linear reduction. They are the value and iteration start time of the second gain value. Normally, the initial value for the second gain is much less than the initial value for the first gain. A initial second gain value of one-tenth the value of the first gain value was chosen. For example,

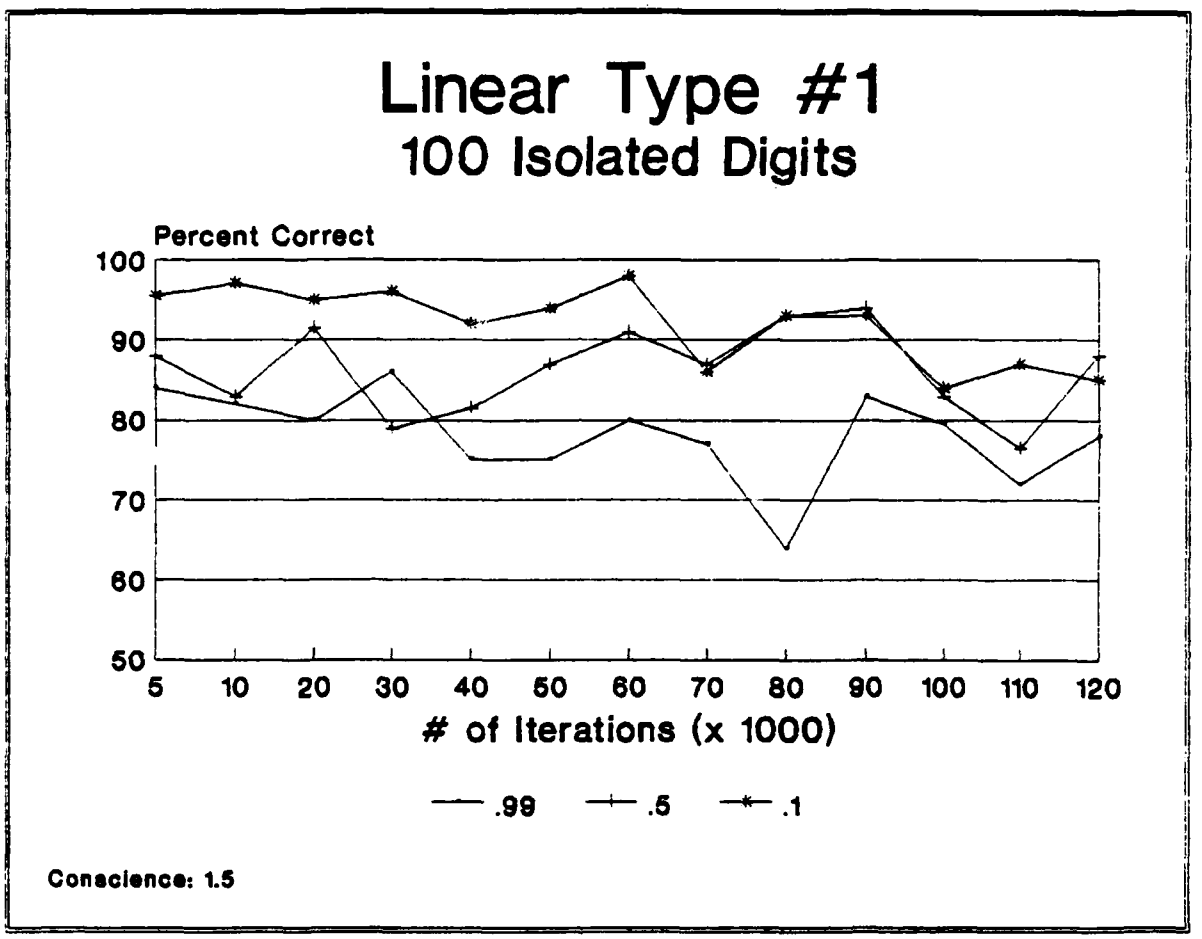

Figure 19. Linear Type #1 Test Results (Isolated Digits)

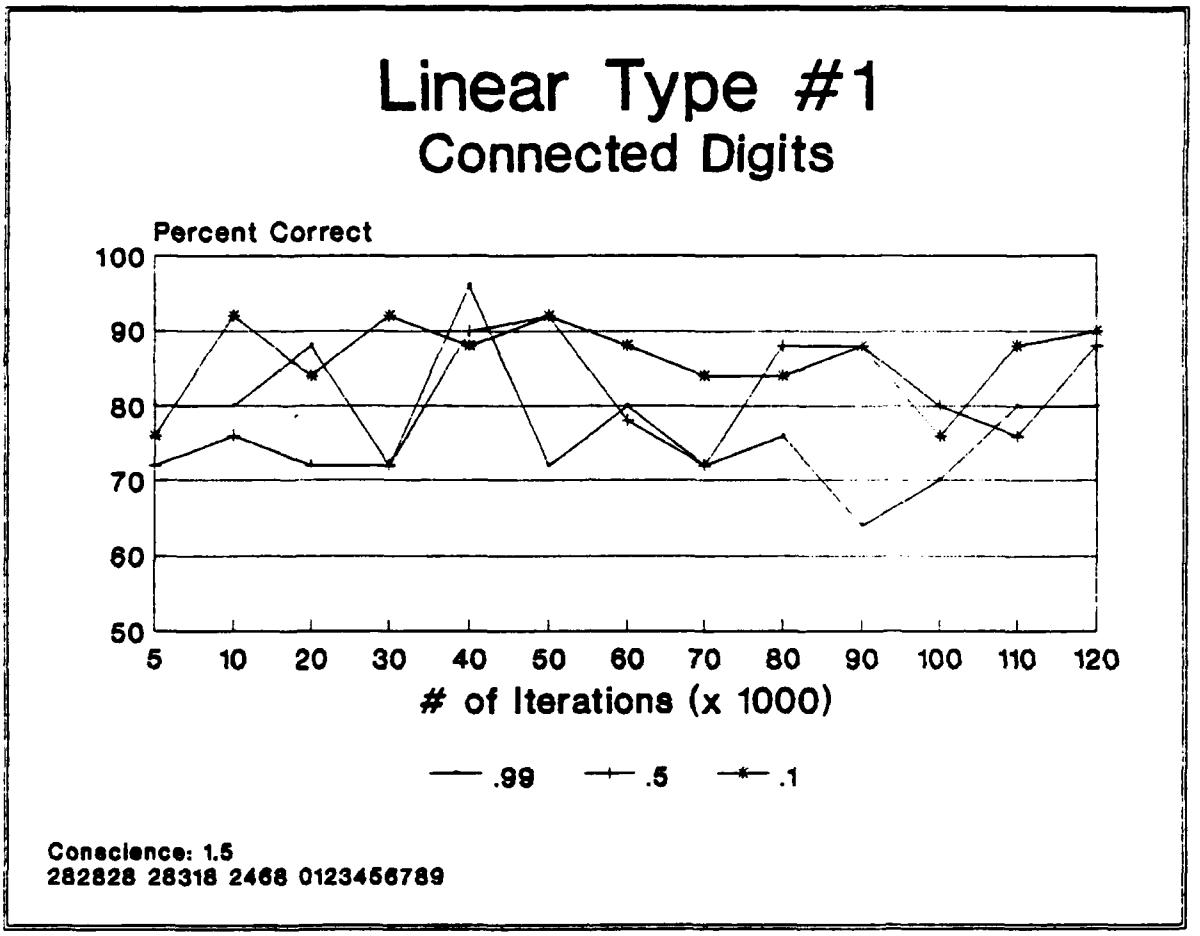

Figure 20. Linear Type #1 Test Results (Connected Digits)

with an initial first gain value of 0.5, the initial start gain value of 0.05 was used for the second gain.

Two sets consisting of three series of 13 Kohonen neural networks were tested using Piecewise-linear gain reduction. The first set had the second gain starting at 20% of the total iteration count; and the second set had the second gain starting at 80% of the total iteration count. For example, a neural network trained for a total of 100,000 iterations will have the second gain start at 80,000 iterations for the 80% case. Table 7 shows the gain start times for both sets of tests using Piecewise-Linear gain reduction.

Table 7. Piecewise Linear Gain Reduction Schedule

Test Number	*Training Iterations*		
	Start First Gain	*Start Second Gain 20% of Total*	*Start Second Gain 80% of Total*
1	5,000	1,000	4,000
2	10,000	2,000	8,000
3	20,000	4,000	16,000
4	30,000	6,000	24,000
5	40,000	8,000	32,000
6	50,000	10,000	40,000
7	60,000	12,000	48,000
8	70,000	14,000	56,000
9	80,000	16,000	64,000
10	90,000	18,000	72,000
11	100,000	20,000	80,000
12	110,000	22,000	88,000
13	120,000	24,000	96,000

The set of tests using an initial start for the second gain of 20% of the total iteration training time are discussed first. Referring to Figure 21 and Figure 22, the results show that using a value of of 0.1 for the first gain consistently gave excellent recognition results. The excellent accuracy obtained for an initial first gain value of 0.1 is clearly evident in both graphs. Neural networks trained using

a start gain value of 0.99 gave the worst recognition accuracy. Using a gain value of 0.5 gave slightly better test results when compared to a gain value of 0.99. The average accuracies for both 0.5 and 0.99 were below 70%

All three gain values gave better recognition accuracies when the second starting gain began at 80% of the total iteration count. The results are shown in Figure 23 and Figure 24 for the isolated and connected word case. Table 8 lists the range and average values for the Piecewise-Linear tests. The results show that recognition rates will improve if the first gain term decreases quickly and the second gain term is allowed to decrease slowly in value. The best accuracy rates are possible if a starting gain close to 0.1 is used.

Table 8. Piecewise-Linear Recognition Accuracy

Starting Gains	*Start 2nd Gain (% of Total)*	*Range (%)*		*Average (%)*	
		Iso	*Con*	*Iso*	*Con*
0.10/0.01	20%	83-93	76-88	87	82
0.10/0.01	80%	83-95	72-92	90	85
0.50/0.05	20%	50-86	48-80	70	69
0.50/0.05	80%	75-95	68-92	83	81
0.99/0.01	20%	51-74	54-88	68	67
0.99/0.01	80%	60-85	68-88	74	77

The astute observer will notice in Figure 21 and Figure 22 that Barmore's final recognition accuracies were not duplicated when the Piecewise-Linear gain method was evaluated. There are three reasons why recognition accuracies of 99% for isolated and 90% for connected speech were not observed. The primary and most obvious reason is the use of a different seed value. All gain tests were run using a seed value of #72 while Barmore used a seed value of #33. The second reason is number of words used to test the system. Barmore used 110 isolated words and eight sets of connected words whereas in Phase I, 100 isolated words and four sets of connected words were used to test the system. Four sets of connected words were

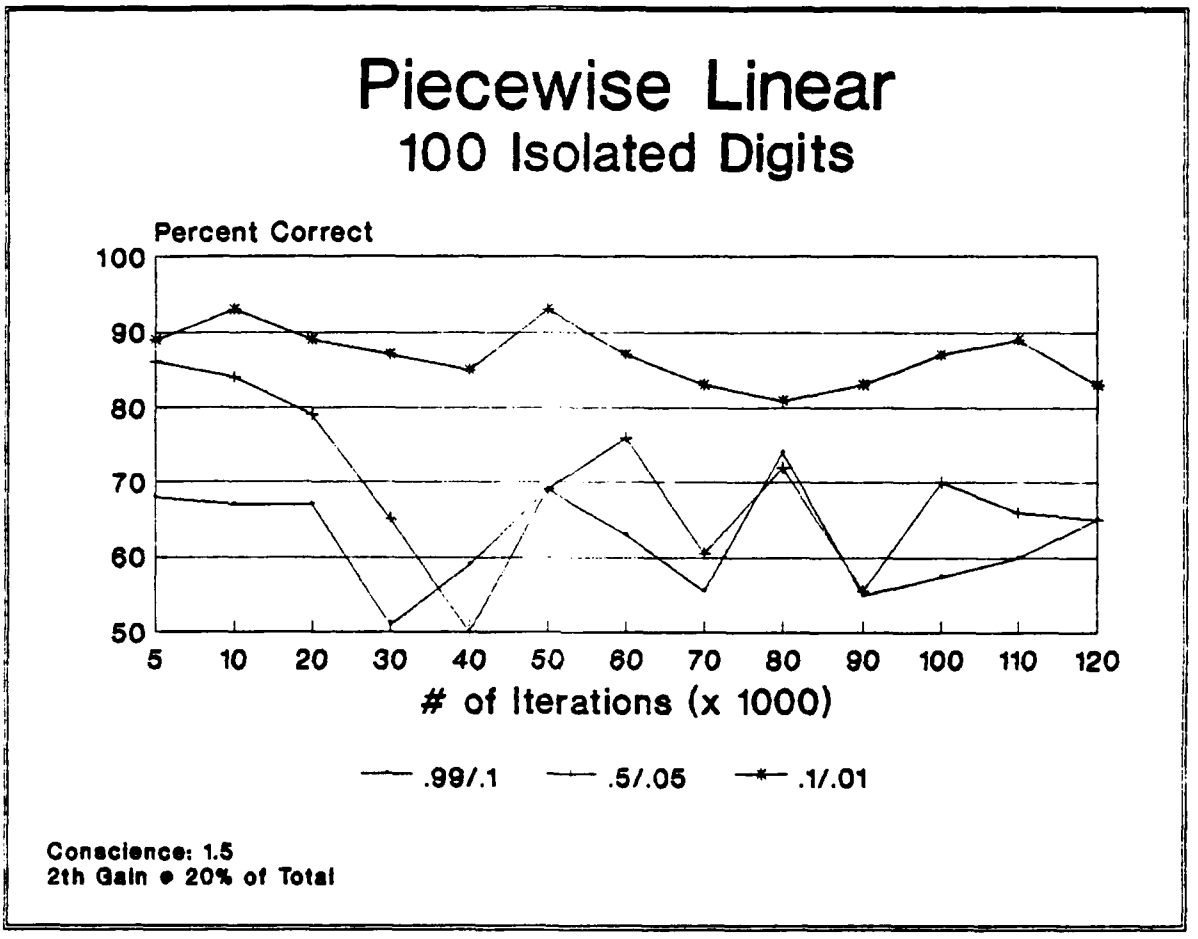

Figure 21. Piecewise-Linear Test Results (2nd Gain 20%) (Isolated Digits)

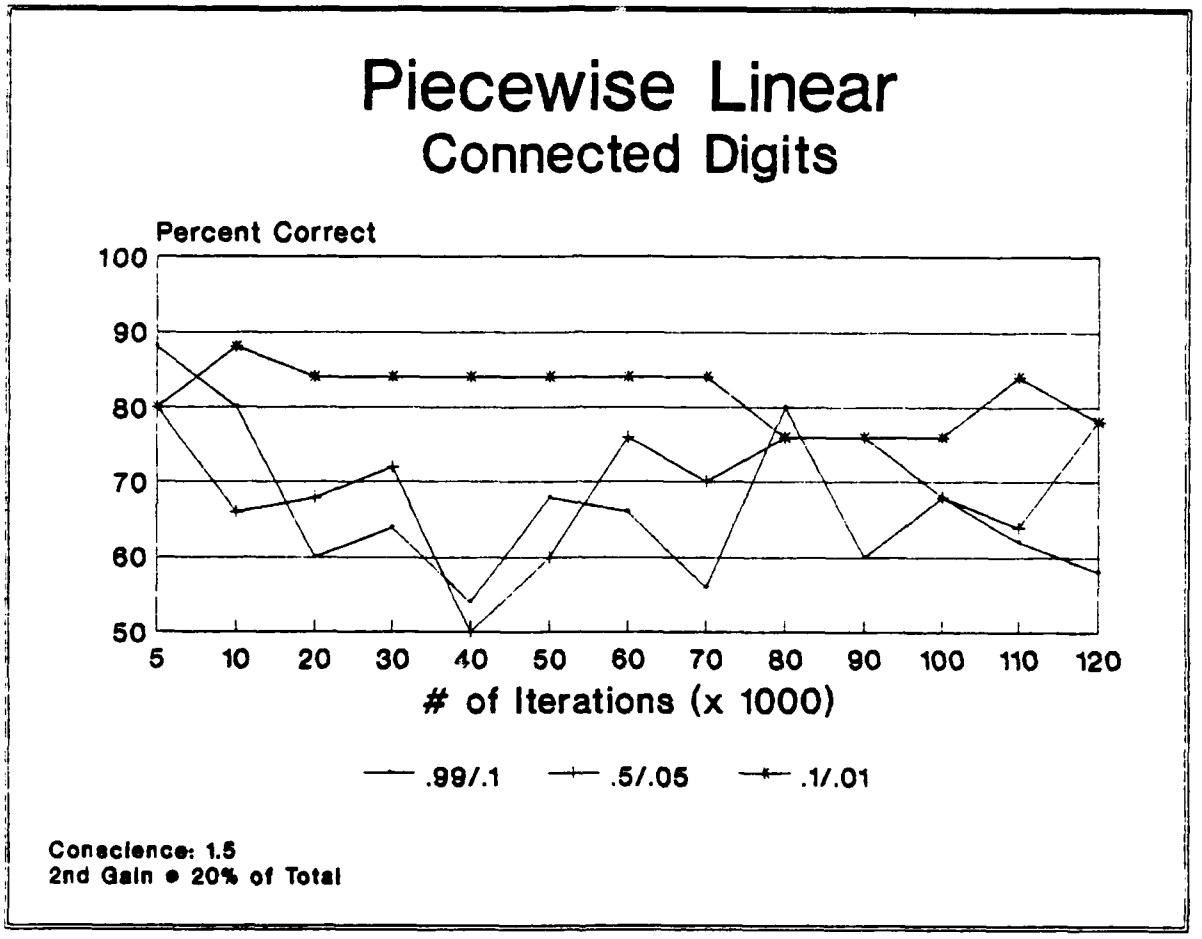

Figure 22. Piecewise-Linear Test Results (2nd Gain 20%) (Connected Digits)

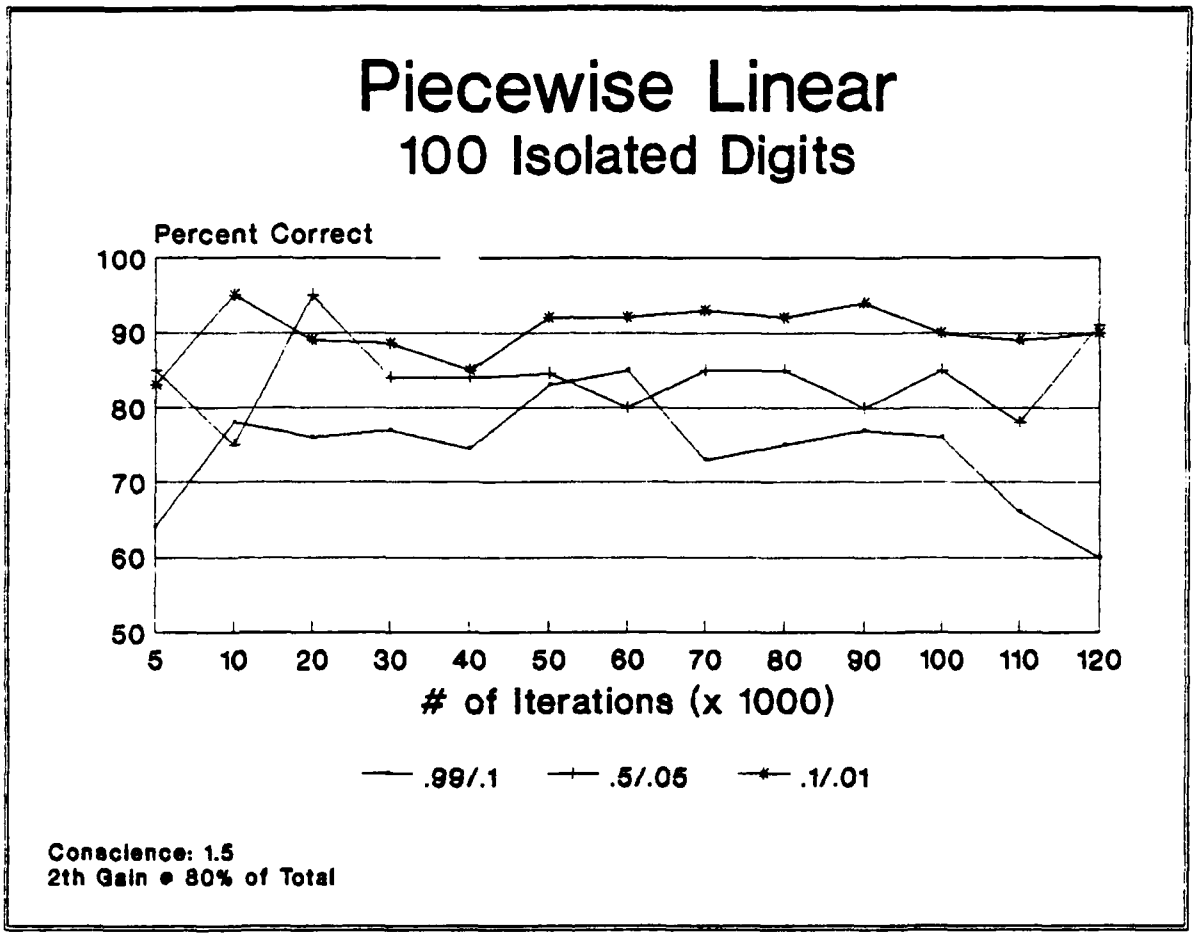

Figure 23. Piecewise-Linear Test Results (2nd Gain 80%) (Isolated Digits)

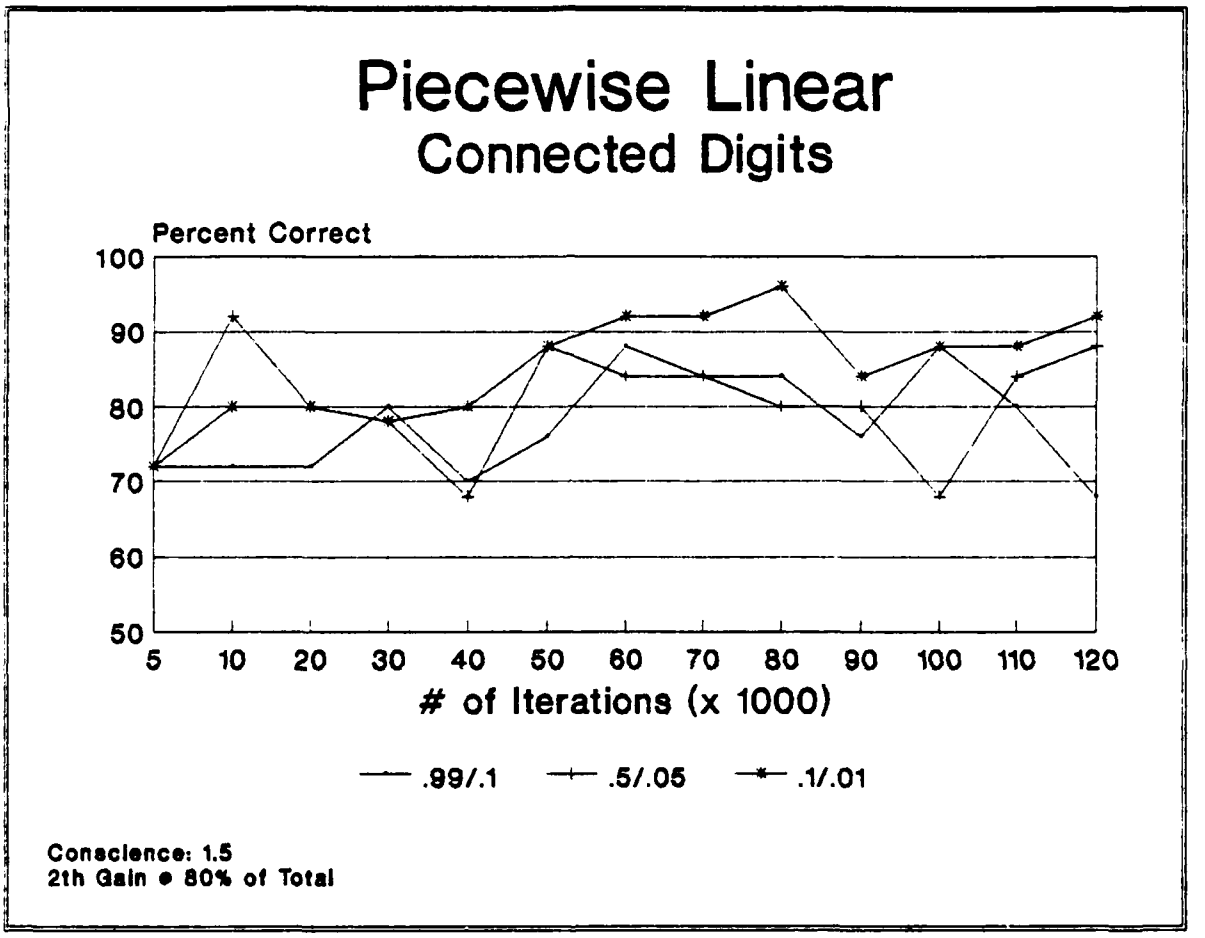

Figure 24. Piecewise-Linear Test Results (2nd Gain 80%) (Connected Digits)

adequate in testing the different methods in Phase I. Note that 210 isolated words and 10 sets of connected words were used in Phase II testing of the recognition system.

The last reason is subtle but dramatic in its effect. Barmore, using Piecewise-Linear reduction in his final recognition system, trained his Kohonen network for a total of 90,000 iterations and started the second gain curve at 20,000. The test was trained using a 90K/20K training schedule. A value of 20,000 represents 22.2% of the total iteration count. In this thesis, a value of 20% was used for the start of the second gain curve. Twenty-percent computes to an actual iteration count of 18,000, (90K/18K training schedule), giving a difference in second gain start times of 2,000 iterations. Another test was conducted to show the effect of varying the start of the second gain term by 2% and the initial seed value.

Three other neural networks were trained so that the seed values and difference in second gain start times could be compared. Two networks were trained using a seed value of #33 and gain reduction schedules of 90K/18K and 90K/20K. Another network was trained using a seed value of #72 and a schedule of 90K/20K. The results are shown in Table 9.

Table 9. Effect of Varying Start of 2nd Gain Curve and Seed Value

Initial Seed	*Training Schedule*	*Results (%)* *Iso*	*Con*
72	90K/18K	83	76
72	90K/20K	89	72
33	90K/18K	99	96
33	90K/20K	94	84

It was expected that the recognition rates would vary when a different seed value was used, but the large observed variation in recognition accuracy was not expected. The recognition rates varied by 16 percentage points for isolated speech and 20 percentage points for connected speech with different seed values using the

90K/18K training schedule. For the 90K/20K training schedule, the difference was 5 percentage points and 12 percentage points for isolated and connected speech. The other significant finding was the variation in recognition rates solely due to a 2% change in the start of the second gain reduction curve. Using a seed value of #72, the recognition rates varied by 6 percentage points for isolated speech and 4 percentage points for connected speech. With a seed of #33 and a training schedule of 90K/20K found in Barmore's final system, the recognition rates observed were 99% and 96%. Changing the start of the second gain curve by 2% gave recognition rates of 94% and 84%. This decrease in overall accuracy consequent to a seemingly innocuous parameter change could potentially make the recognition system unacceptable for aircraft applications.

Linear Type #2 and Exponential Gain Reduction The Linear Type #2 and Exponential reduction methods used the same linear gain reduction schedule. The difference between the two methods is how the neighborhood decreases as training proceeds (see Figure 12 and Figure 13). The results of training Kohonen networks using these two methods is shown in Figure 25 for isolated speech and Figure 26 for connected speech. Table 10 summarizes the results for both types of speech.

Table 10. Linear Type #2 and Exponential Recognition Accuracy

Reduction Method	*Range (%)*		*Average (%)*	
	Iso	*Con*	*Iso*	*Con*
Linear Type #2	84-94	68-100	90	87
Exponential	83-98	80-96	91	88

Both gain reduction methods gave high recognition accuracies. Each had isolated word recognition accuracies that averaged about 90% for the series, and connected word accuracies that averaged in the high 80% range. Exponential reduction performed slightly better with recognition accuracies in the high 90% range

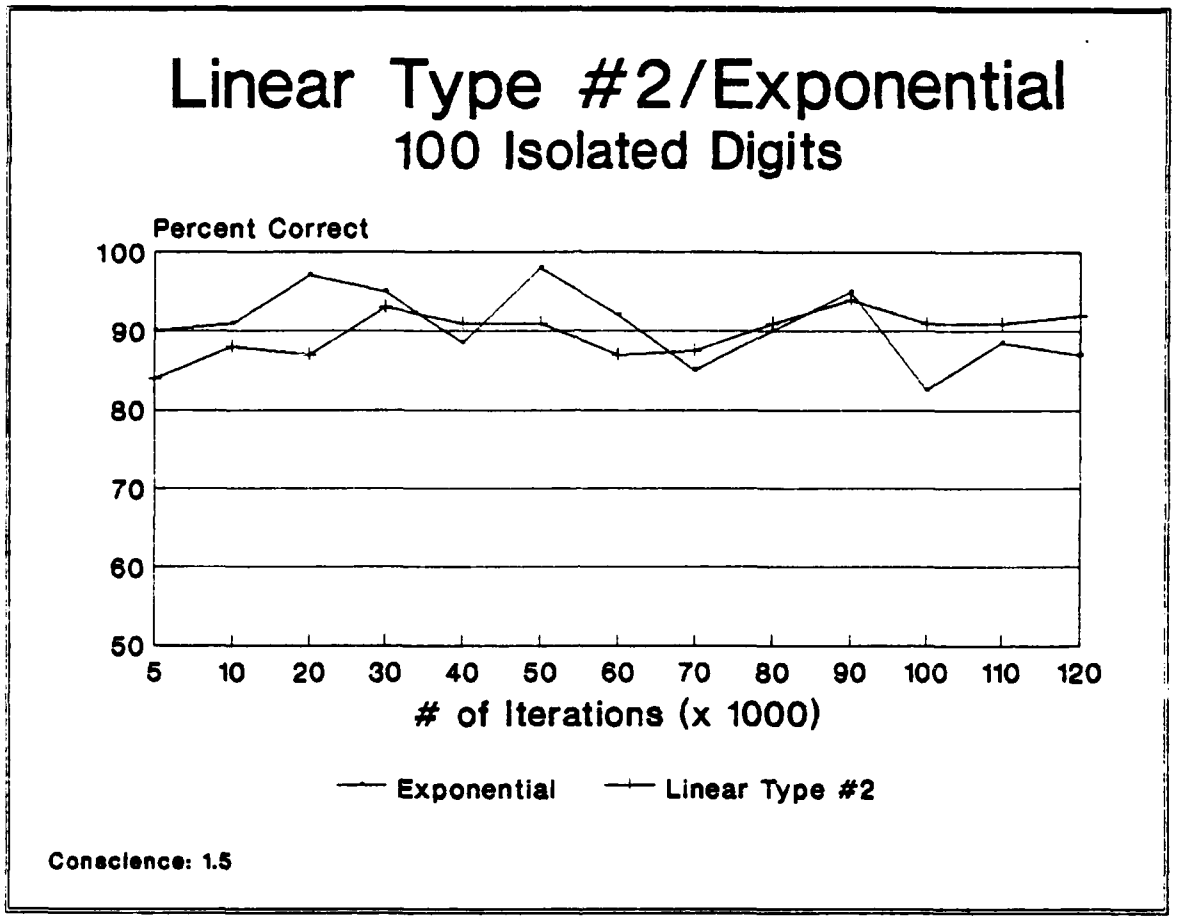

Figure 25. Linear Type #2 and Exponential Reduction Test Results (Isolated Digits)

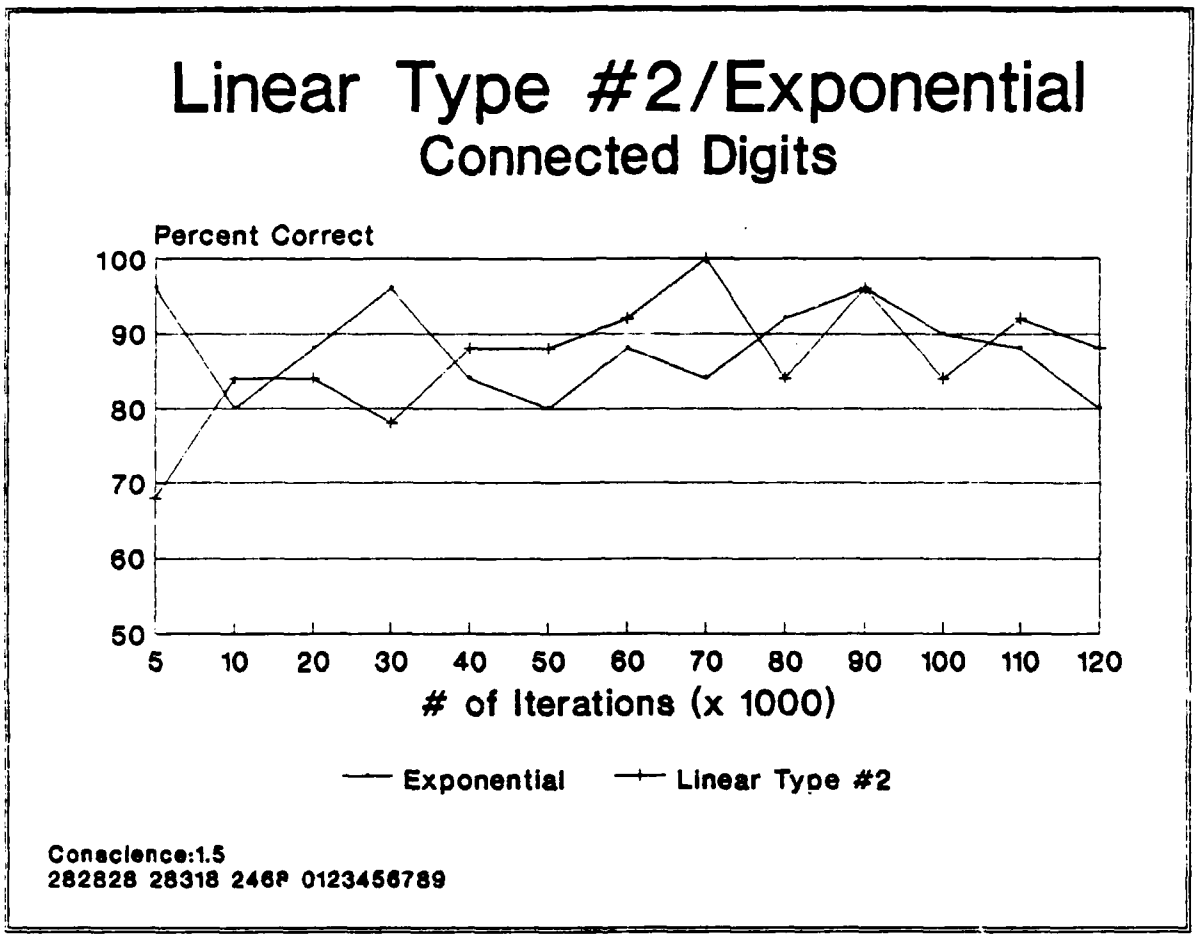

Figure 26. Linear Type #2 and Exponential Reduction Test Results (Connected Digits)

for iteration cycle times less than 60,000. The sharper decrease in neighborhood size used with exponential reduction appears to improve the performance of the Kohonen network.

Central-Adaptation Gain Reduction The last gain reduction method evaluated was Central-Adaptation. With this method, the neighborhood size is held constant while the gain factor varies with distance from the winning node. Except for a good recognition accuracy of 91% at 10,000 iterations using isolated words, the Central-Adaptation gain reduction method did not provide the proper mix of gain and neighborhood reduction necessary for speech applications.

The Central-Adaptation gain formula was evaluated for possible improvement. The gain formula, which is reproduced from Chapter 4, has an exponential argument with two terms within the brackets.

$$\alpha = 2C(1 - t/T)e^{-r_{\nu,j}[1.4/R_d + t/T(5.6-1.4/R_d]} \tag{13}$$

The value of t/T in the second term of the argument determines which of the two terms will dominate while the network is training. During the early part of training when t/T is small, the first term $(1.4/R_d)$ dominates. As the training progresses and $t/T \rightarrow 1$, the second gain term dominates.

A low iteration factor (LIF) term and high iteration factor (HIF) term was added to the basic Central-Adaptation gain equation as seen below. Note that the LIF and HIF equal one in the basic equation. (See Equation 13).

$$\alpha = 2C(1 - t/T)e^{-r_{\nu,j}[1.4/(R_d * LIF) + t/(T * HIF)(5.6-1.4/R_d]} \tag{14}$$

Two additional series of tests were run using different iteration factors in an effort to improve the performance of the Central-Adaptation method. The first series of tests used a LIF of 2.5 and a HIF of 5.0. The second series used a LIF of 2.5 and

a HIF of 10.0. The results for all three series are shown in Figure 27 for isolated words and Figure 28 for the connected word case. Table 11 gives the overall range and average recognition rates for all three series of tests.

Table 11. Central-Adaptation Recognition Accuracy

Low Iteration Factor	*High Iteration Factor*	*Range (%)*		*Average (%)*	
		Iso	*Con*	*Iso*	*Con*
1.0	1.0	74-91	60-84	80	74
2.5	5.0	79-95	70-92	87	79
2.5	10.0	75-88	54-88	79	74

The best recognition accuracy was found using iteration factors of 2.5 and 5.0. The results, however, were not as good as the other gain reduction methods tested. It is possible that the right mix of iteration factors could be found to significantly improve recognition accuracy.

One final series of tests were conducted using Central-Adaptation. As previously noted, the neighborhood is held constant during training with the Central-Adaptation method. The last series of tests combined the Central-Adaptation gain reduction equation with the exponential reduction in neighborhood size. The results are shown in Figure 29. No significant improvement was observed using exponential neighborhood reduction when compared to a constant neighborhood size.

The series of tests using the five gain reduction methods showed that the Exponential gain reduction method gave the best overall recognition accuracy for both isolated and connected speech. Linear Type #1 and Piecewise-Linear (2nd gain start at 80% of total) with an initial gain of 0.1 also provided excellent recognition results. Linear and Piecewise-Linear reduction starting with a gain near unity gave the worst overall recognition accuracy. A major disappointment was the response of the Central-Adaptation method in the various test configurations.

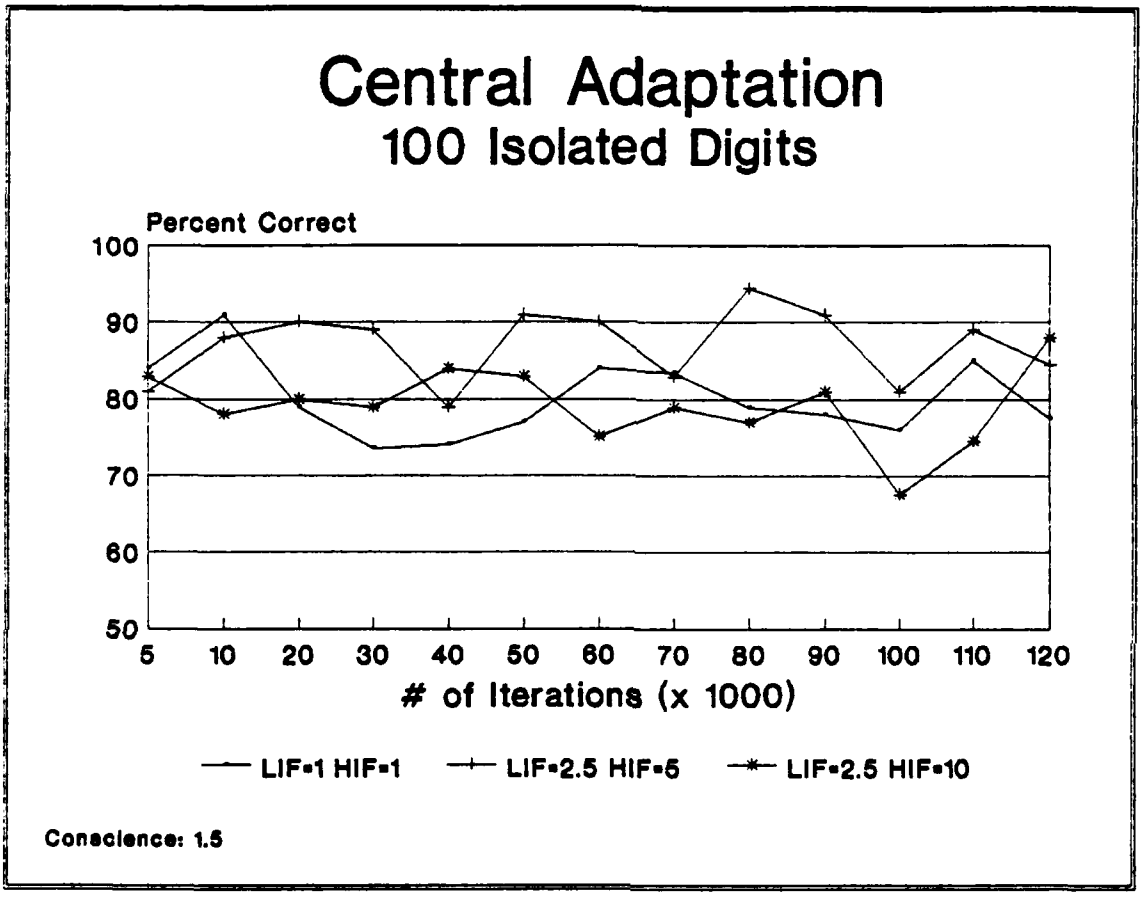

Figure 27. Central-Adaptation Reduction Test Results(Isolated Digits)

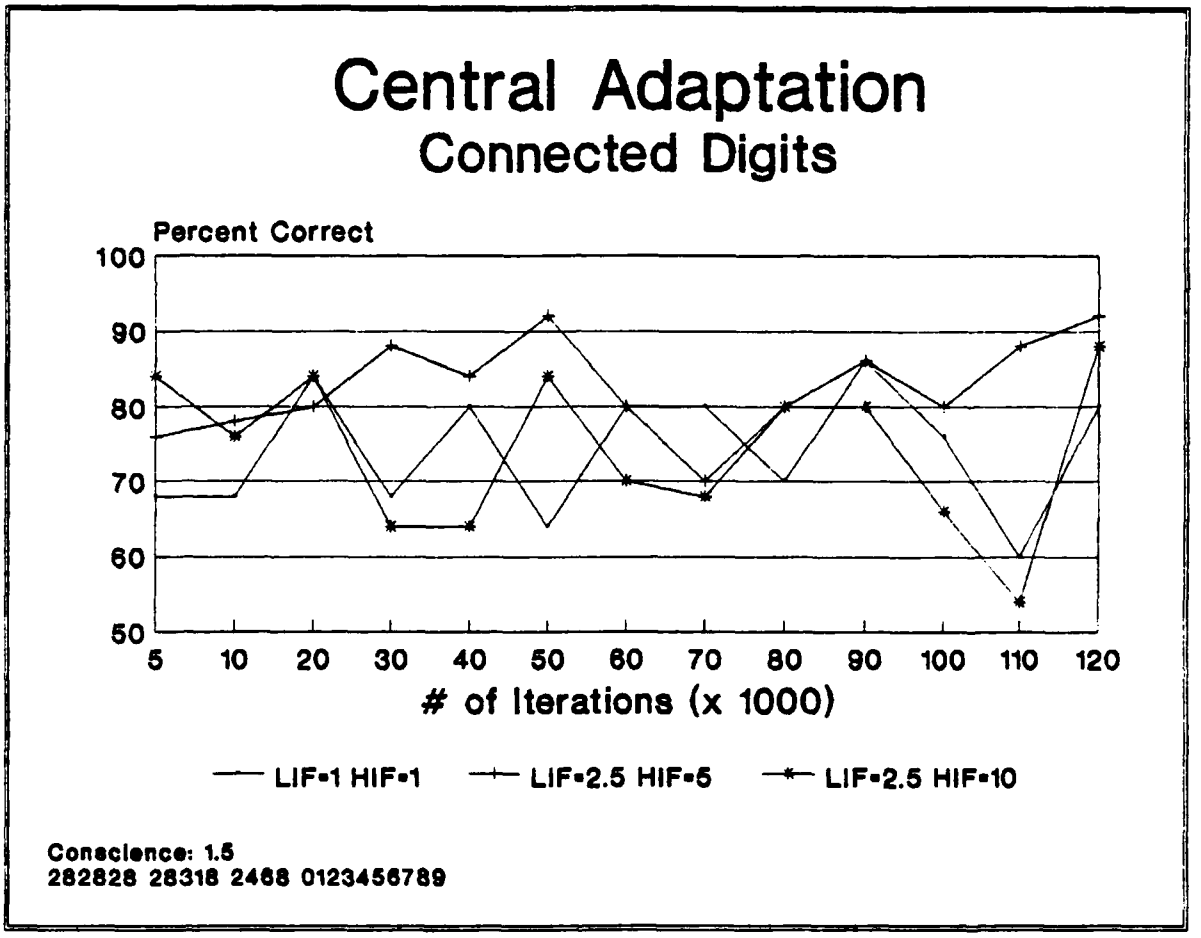

Figure 28. Central-Adaptation Reduction Test Results(Connected Digits)

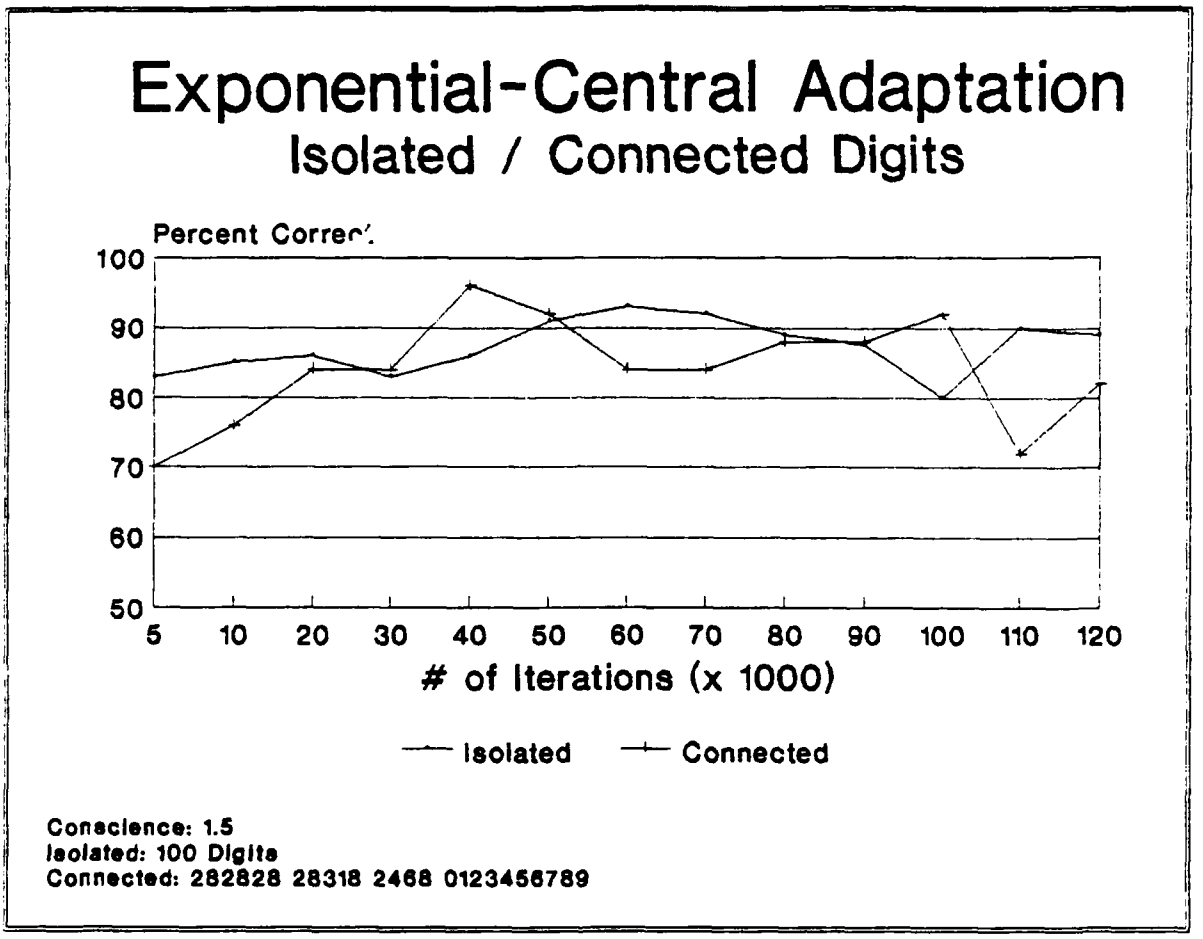

Figure 29. Exponential-Central-Adaptation Reduction Test Results

Conscience There was less information available concerning the effect of conscience on a Kohonen network than the effect of various gain methods on a Kohonen network. Two data points, (1.1 and 1.5), were evaluated by Barmore [4:4-9]. Additional information was needed on conscience to understand its effect on Kohonen network training, and to optimize the performance of the speech recognition system.

Conscience testing using a wide range of conscience factors is time and computer intensive. Given these constraints, a series of tests were conducted on the sequence $000, 001, \cdots, 111$ and different conscience factor values. Each 15×15 network was trained with this sequence for $2,000$ iterations to determine the range of conscience factor values that affected network training.

Two measures of Kohonen network usage were used for these tests. The Maximum Node Hit Value is the maximum number of times any one node on the Kohonen surface becomes the *winning* node. Node Utilization is a measure of the percentage of nodes that *win* at least *once* during a complete training session. Conscience factors between 1.1 and 20.0 were evaluated with the results shown in Figure 30 and Figure 31.

The results show that node utilization is near 100% for a conscience factor of 1.1, (maximum conscience for this test), and gradually declined to 68% for a conscience factor of 20.0 (minimum conscience for this test). A test conducted without any conscience had a node utilization of 63%. These results show that conscience had an effect on network training at least up to a conscience factor value of 20.0. This result dictated the range of conscience factor values used to test the speech recognition system.

An in-depth analysis of conscience was performed on two of the gain reduction methods. Time constraints precluded evaluation of every reduction method. The Piecewise-Linear and Exponential gain reduction methods were evaluated using the conscience factor values listed in Table 12. Each conscience factor was run on

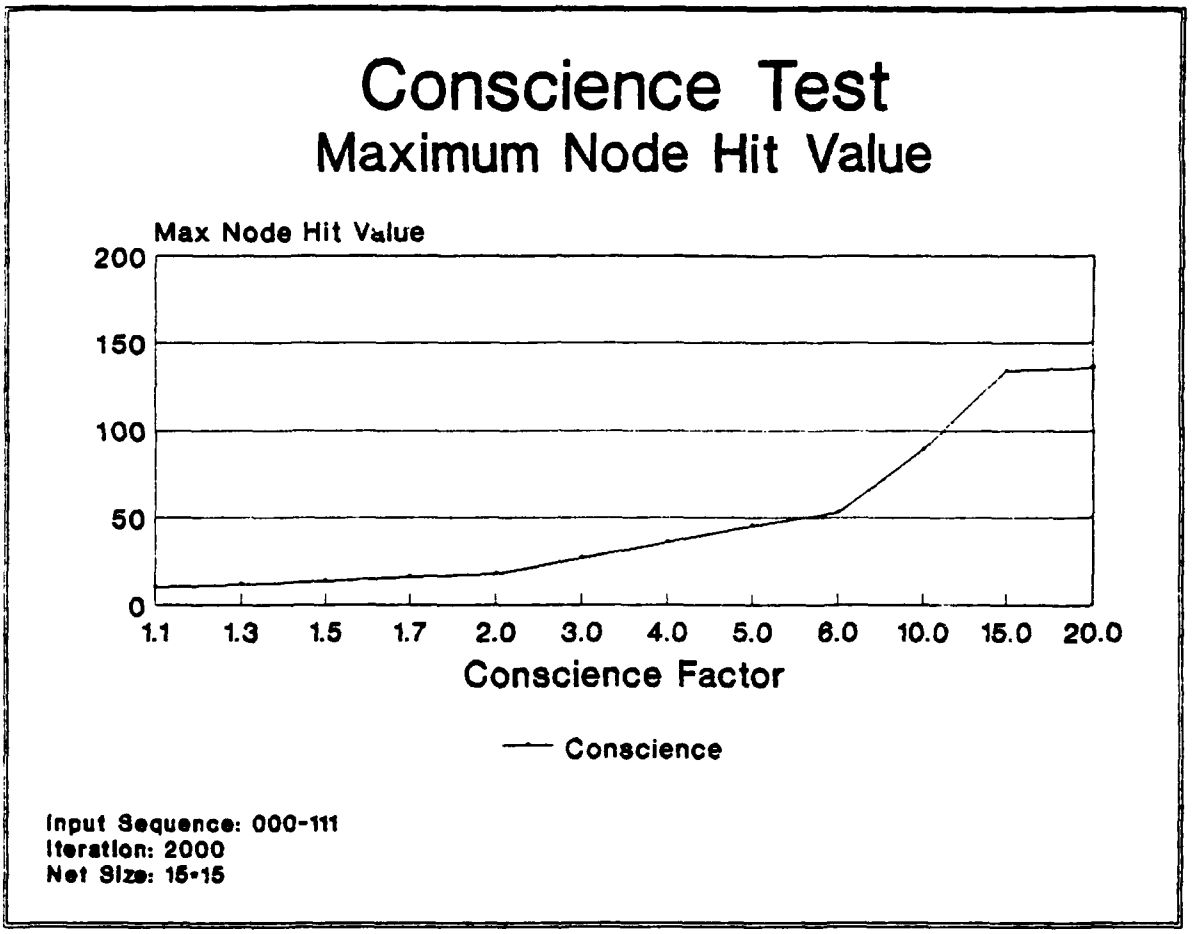

Figure 30. Maximum Node Hit Value Test Results

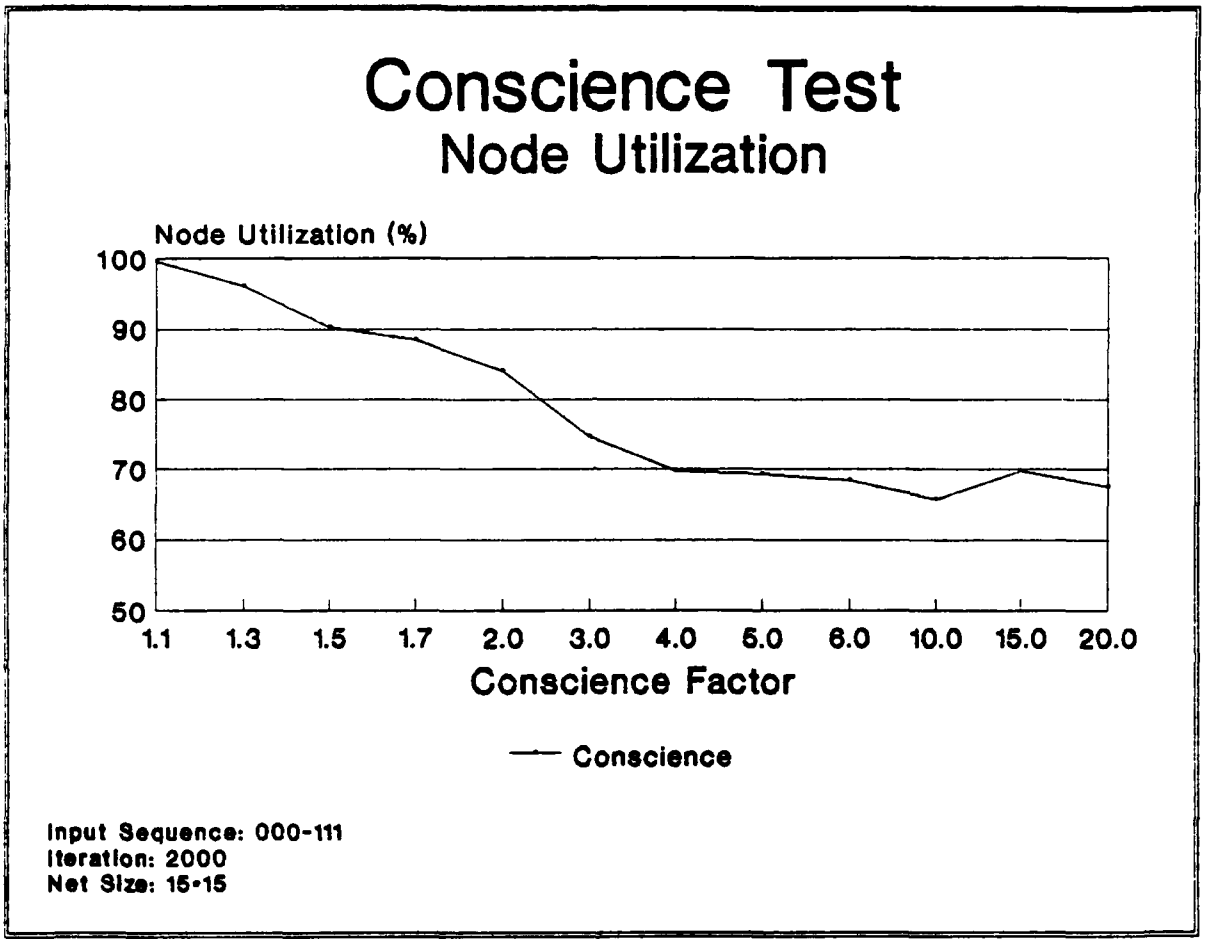

Figure 31. Node Utilization

Table 12. Conscience Factor (β) Values

Piecewise-Linear	*Exponential*
1.0	1.0
1.25	1.25
1.5	1.5
2.0	1.7
3.0	1.9
4.0	2.3
5.0	2.7
6.0	3.0
7.0	4.3
8.0	5.0
9.0	6.0
10.0	7.5
12.0	8.0
14.0	9.0
16.0	10.0
	12.0
	14.0
	16.0
no conscience	*no conscience*

the series of 13 Kohonen networks listed in Table 3. The Piecewise-Linear method had the same initial gains and training schedule used by Barmore in his thesis. A three-dimensional surface was generated for each gain method showing recognition accuracy on 100 isolated words plotted against network training iteration size and conscience factor. The plots for the two methods represent more than 44,000 dynamic programming calculations.

There are two three-dimensional plots for each of the gain methods representing different orientations of the Kohonen-Dynamic Programming Recognition Surface. See Figure 32 through Figure 35. In each plot, a conscience factor value of 17 represents a slice through the recognition surface with no conscience. Note that conscience is maximum at a conscience factor of 1.0.

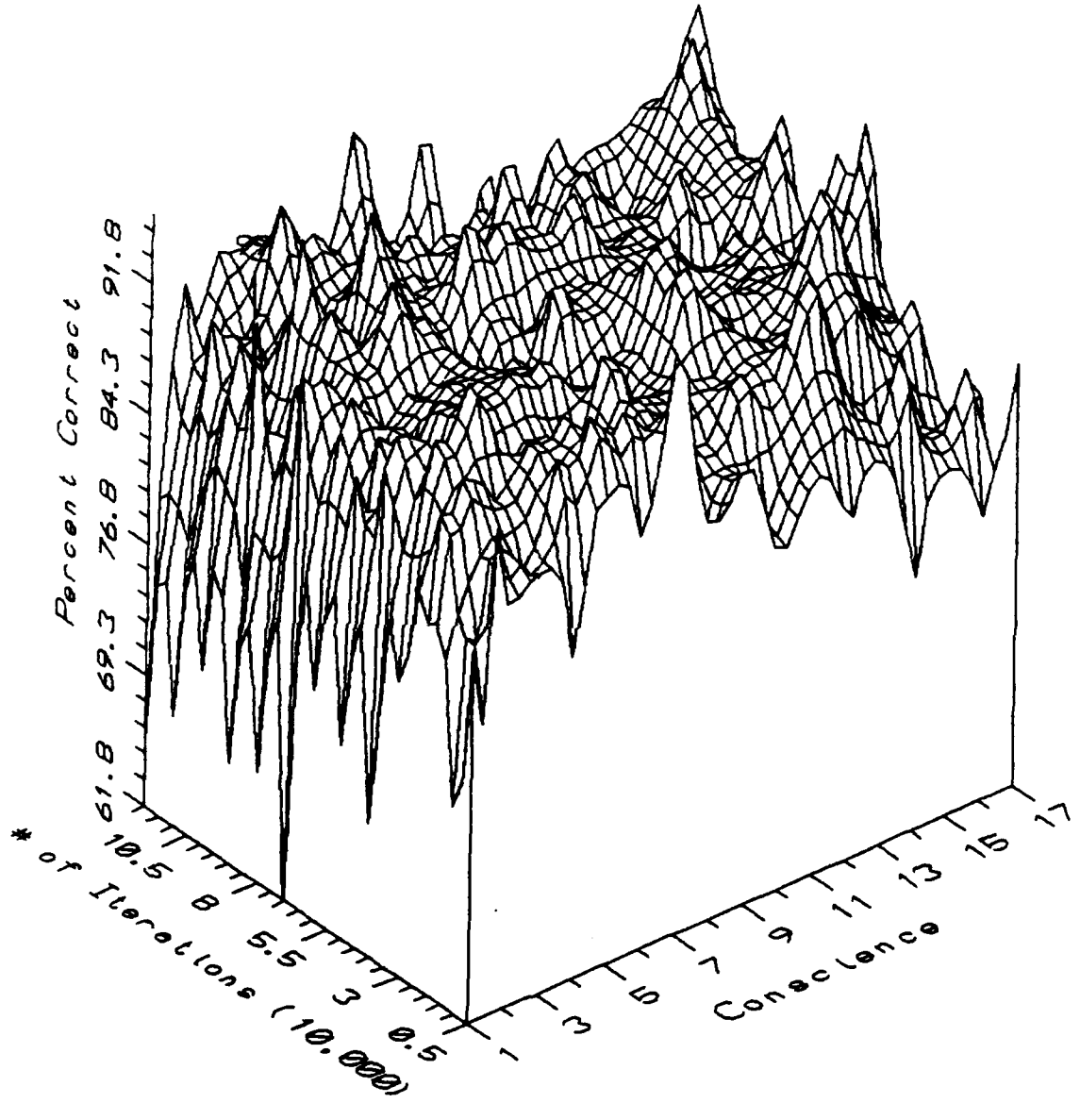

Figure 32. Kohonen-Dynamic Programming Recognition Surface: Piecewise-Linear (Front View)

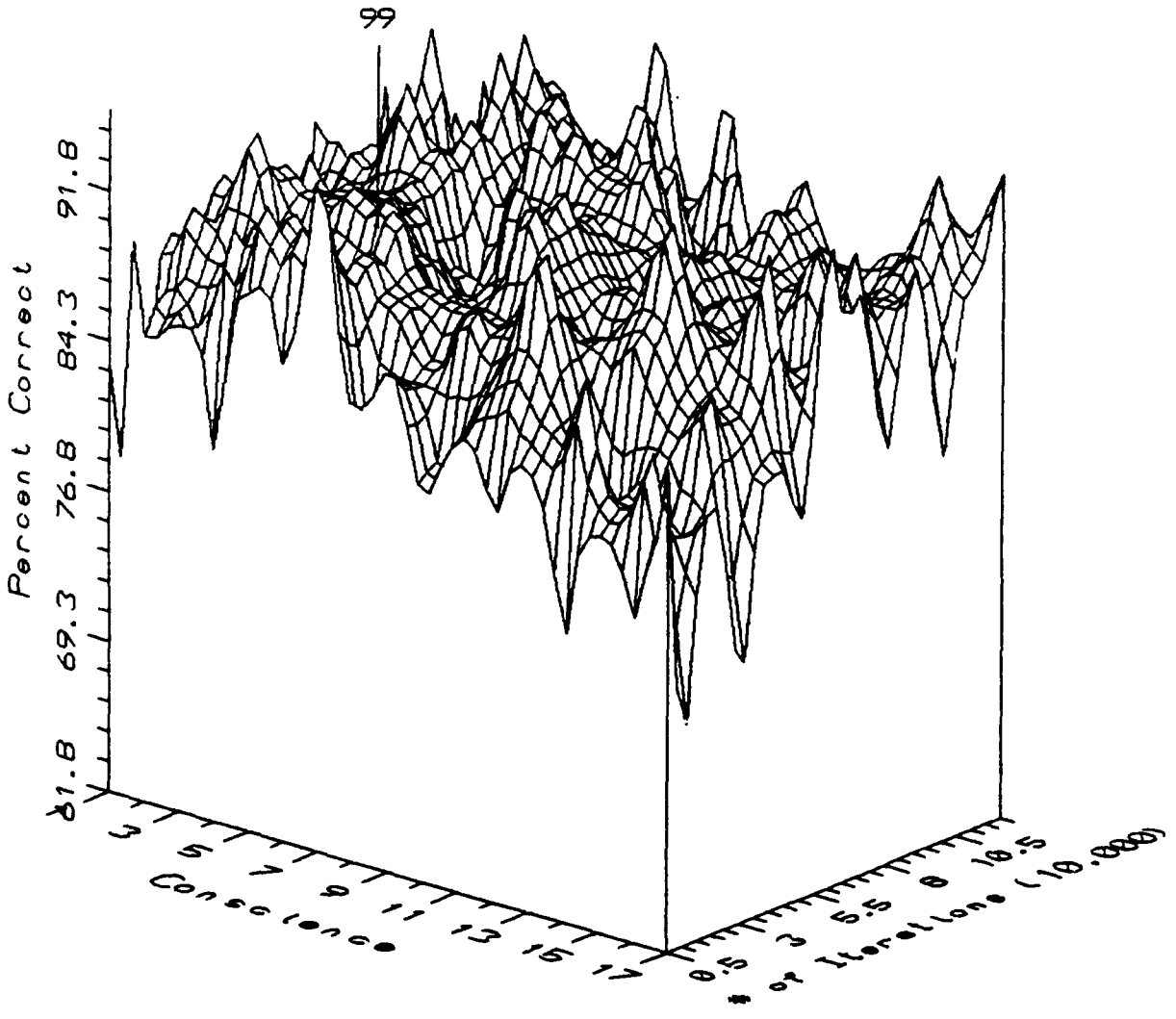

Figure 33. Kohonen-Dynamic Programming Recognition Surface: Piecewise-Linear (Back View)

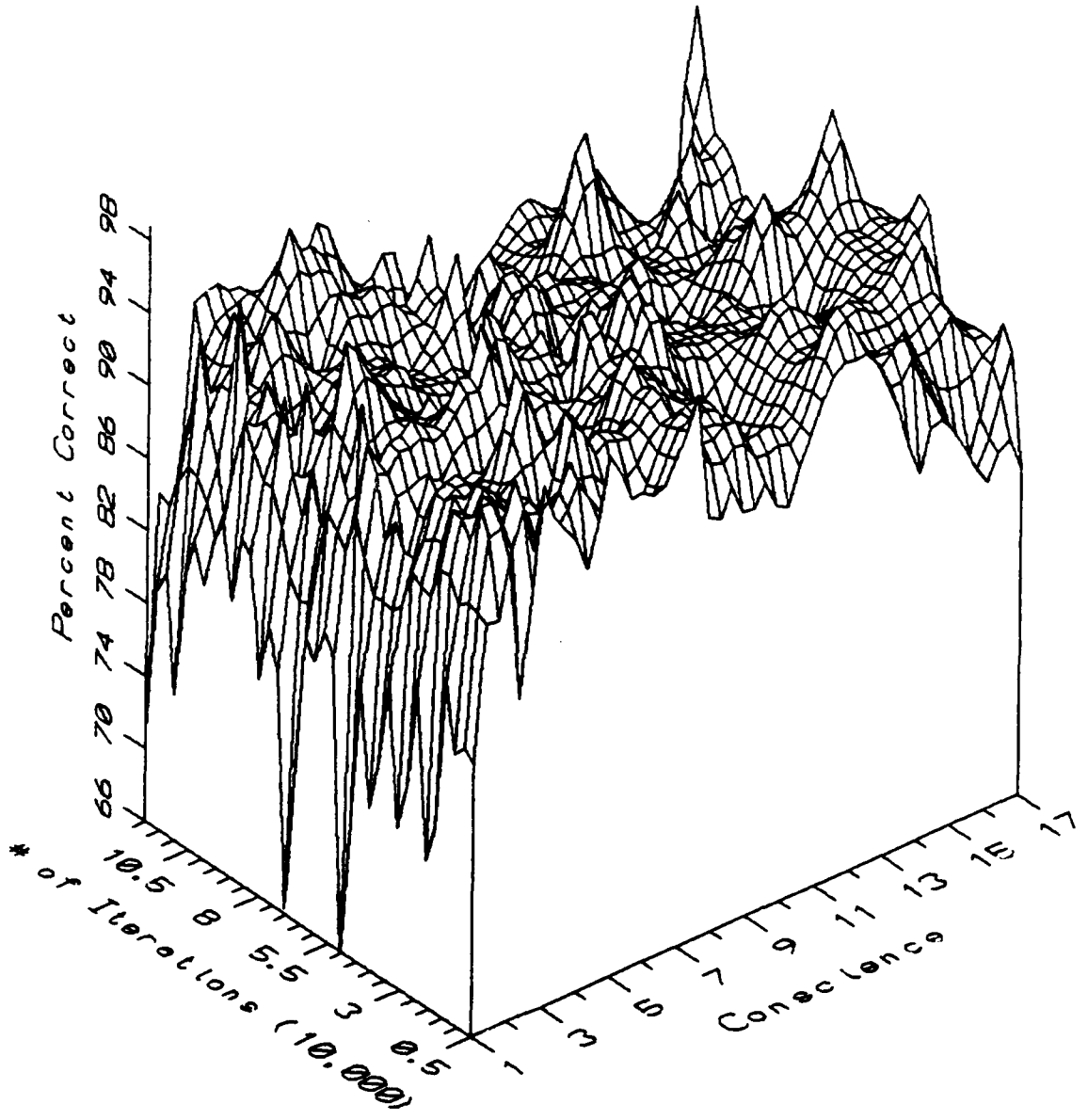

Figure 34. Kohonen-Dynamic Programming Recognition Surface: Exponential (Front View)

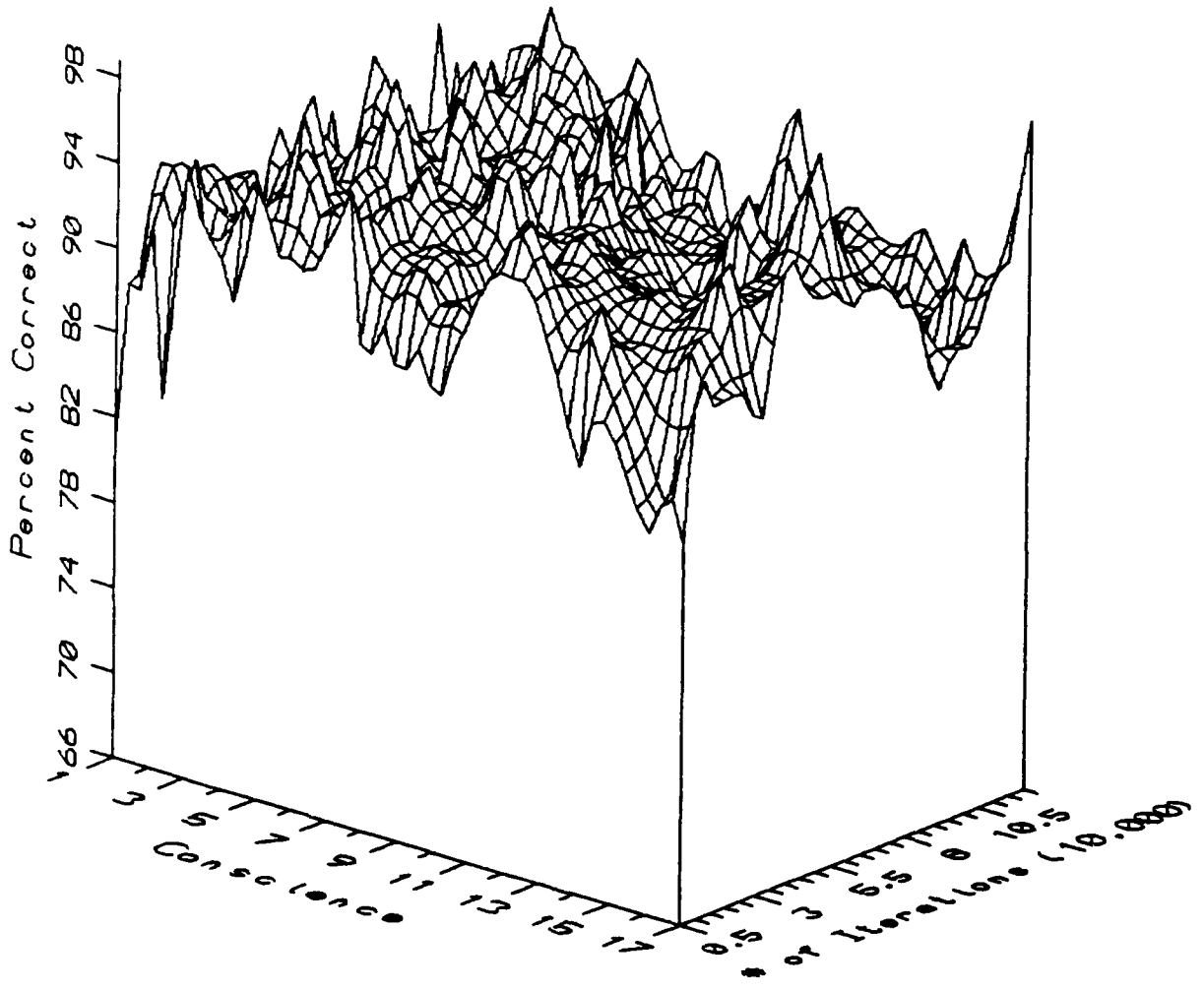

Figure 35. Kohonen-Dynamic Programming Recognition Surface: Exponential (Back View)

The different orientations of the recognition surface provide valuable information about a Kohonen network's ability to segment speech sounds and about the major role conscience plays in the recognition process. Some observations from the three-dimensional plots include:

- Figure 32 and Figure 34 show that the worst recognition accuracy is found at a conscience factor of 1.0 (maximum conscience). The recognition surface drops sharply in both plots at this value. The poor recognition accuracy is probably due to the inability of the Kohonen network to properly segment the sounds into clearly defined regions. Too large a conscience forces the network to determine a *winning* node primarily based on past wins rather than on the relationship between the input data and node weight vectors.
- Kohonen networks tend to organize quickly. The networks trained at 5,000 iterations, along the conscience plane in the three-dimensional plots, already had high recognition rates. Plots from both methods show that conscience helps a Kohonen network organize speech quicker than when conscience is not present. Other than a conscience factor of 1.1, the lowest recognition rates for networks trained for 5,000 iterations occurred with networks trained without conscience.
- The recognition surface does not slope upward in a continuous manner as training iteration size increases. There is no correlation between the training time and recognition accuracy.
- Conscience does affect a Kohonen network's ability to organize speech sounds for conscience factor values much higher than 1.5. The effect of conscience on the network is similar to the results obtained with the sequence $000, 001, \cdots, 111$ on a Kohonen network of the same size. These plots show that the optimum conscience factor value for speech recognition is in the five to seven range.

- The overall recognition surface is much higher using the Exponential gain reduction method when compared to the Piecewise-Linear reduction method. The higher recognition surface means that the Exponential method is superior to the Piecewise-Linear method for speech recognition applications.

Based on the research conducted in Phase I, the recognition system designed in Phase II uses Exponential gain reduction and a conscience factor of 5.0. Figure 36 shows a two-dimensional slice from the three-dimensional plot at this conscience factor using Exponential gain reduction. Results from Phase I also show that a Kohonen network organizes quickly. The number of iterations chosen to train a Kohonen network must not be arbitrary. Some standard measure is required. The standard used for Phase II testing is the size of the data file used to train the network. The size of the data file divided by the number of inputs represents the number of training iterations required for the network to *see* all of the data once. This method applies regardless of the size of the data file; therefore, it standardizes the process of determining the appropriate training cycle time. An experiment using multiples of the data file size is shown in Figure 37. These results show that after the network has *seen* the data only a few times, the recognition accuracy on ten digits was above 90%. This finding is significant. Kohonen neural networks may need to be trained in terms of thousands of iterations rather than tens of thousands of iterations! A complete set of neural networks will be trained up to about 90,000 iterations in Phase II to confirm these findings.

Average Subtraction The range of data in processed sound files was much larger when the average-subtraction routine was used in the preprocessor section. By itself, the average-subtraction routine should not alter the range of the data values, since this routine merely sums 15 component frames, divides the sum by the frame size, and subtracts the resultant sum value from each component value in the frame. In the preprocessor section, the data was processed by the average-

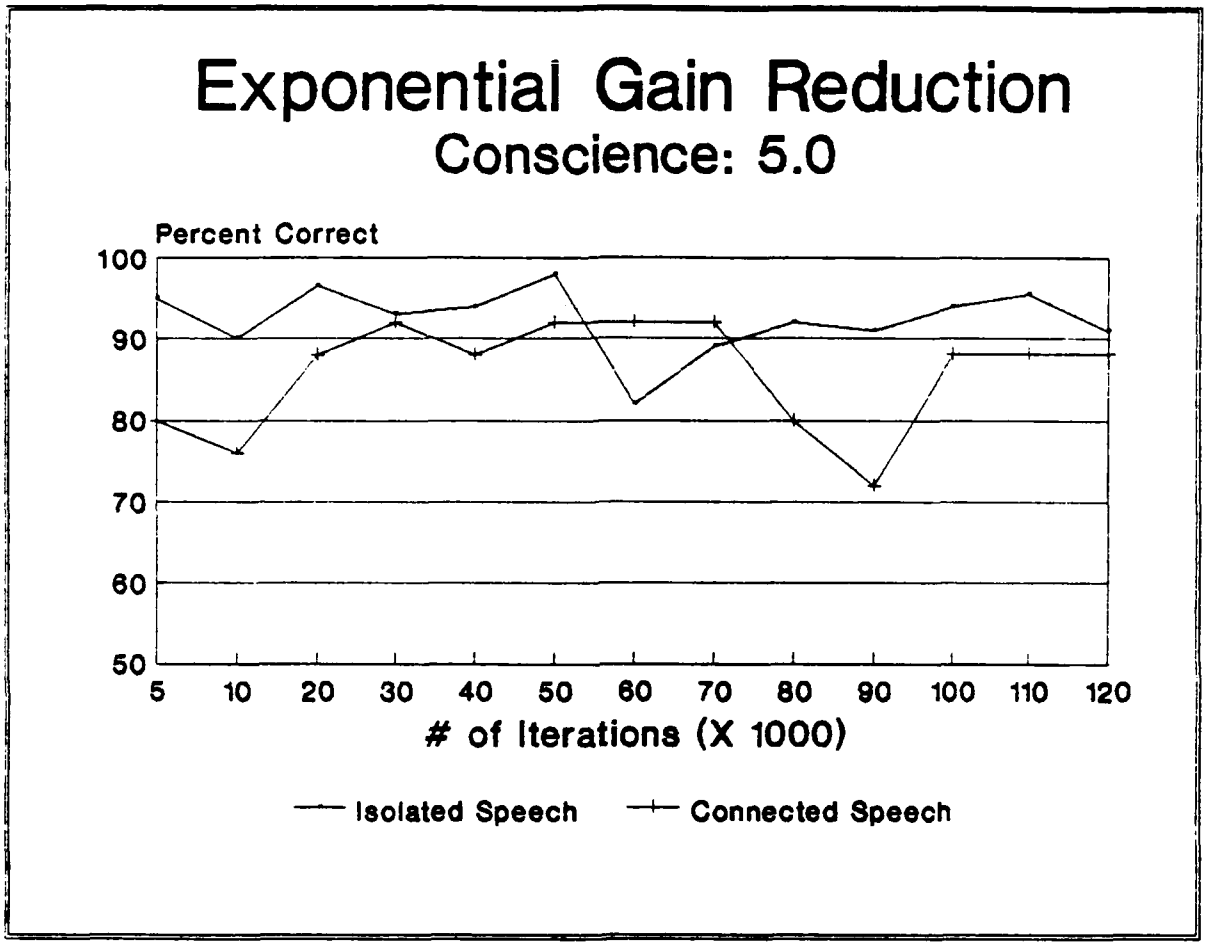

Figure 36. Exponential Gain Reduction (Conscience: 5.0)

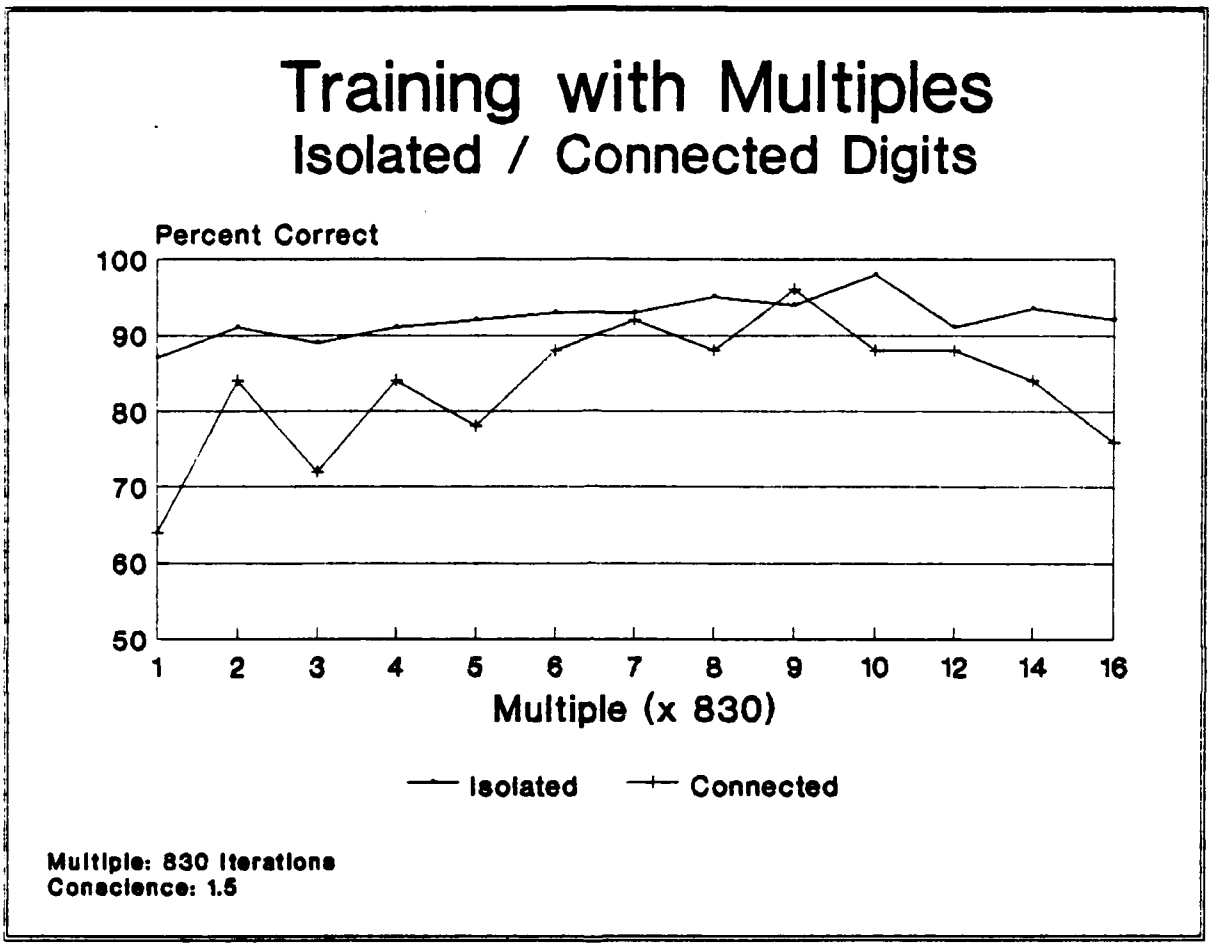

Figure 37. Network Training Using Multiples of Input Data File

subtraction routine, and then the data was normalized. A combination of these two routines working in this sequence created the change in the range of data values. It is possible that altering the position of the individual vectors in the hyperspace could disrupt the delicate interrelationship between the different regions of sound on the Kohonen surface. A series of neural networks were trained to investigate the affect of average subtraction of data on a Kohonen network.

Using the exponential gain reduction method, tests were conducted at conscience factor values of 1.5 and 5.0. Eleven networks were processed at each conscience value with average subtraction, and eleven networks processed without average subtraction. The range of initial weight values for the Kohonen network was changed to a range of zero to one without average subtraction, since the data values were within this range.

The Kohonen networks for this test were trained using multiples of the input data file size divided by the frame size. This value was 1,122. Odd multiples of 1,122 from 1 to 21 were used. The corresponding iteration cycle times of the Kohonen networks were 1,122 to 23,562. Results of isolated and connected speech are shown in Figure 38 and connected speech in Figure 39 for a conscience factor of 1.5. Figure 40 and figure 41 show the results for a conscience factor of 5.0.

The results show that average subtraction did improve somewhat the recognition rates of the system described above. As a result of these tests, the average-subtraction routine was employed in the final recognition system in Phase II. Note that the recognition accuracies obtained during average-subtraction testing support the claim that a Kohonen network does train quickly using speech data.

Phase II

Introduction The recognition system developed in Phase II uses the entire F-16 command vocabulary listed in Table 1. Three sets of these 70 words were used to test the ability of the system to recognize isolated speech. Eleven connected

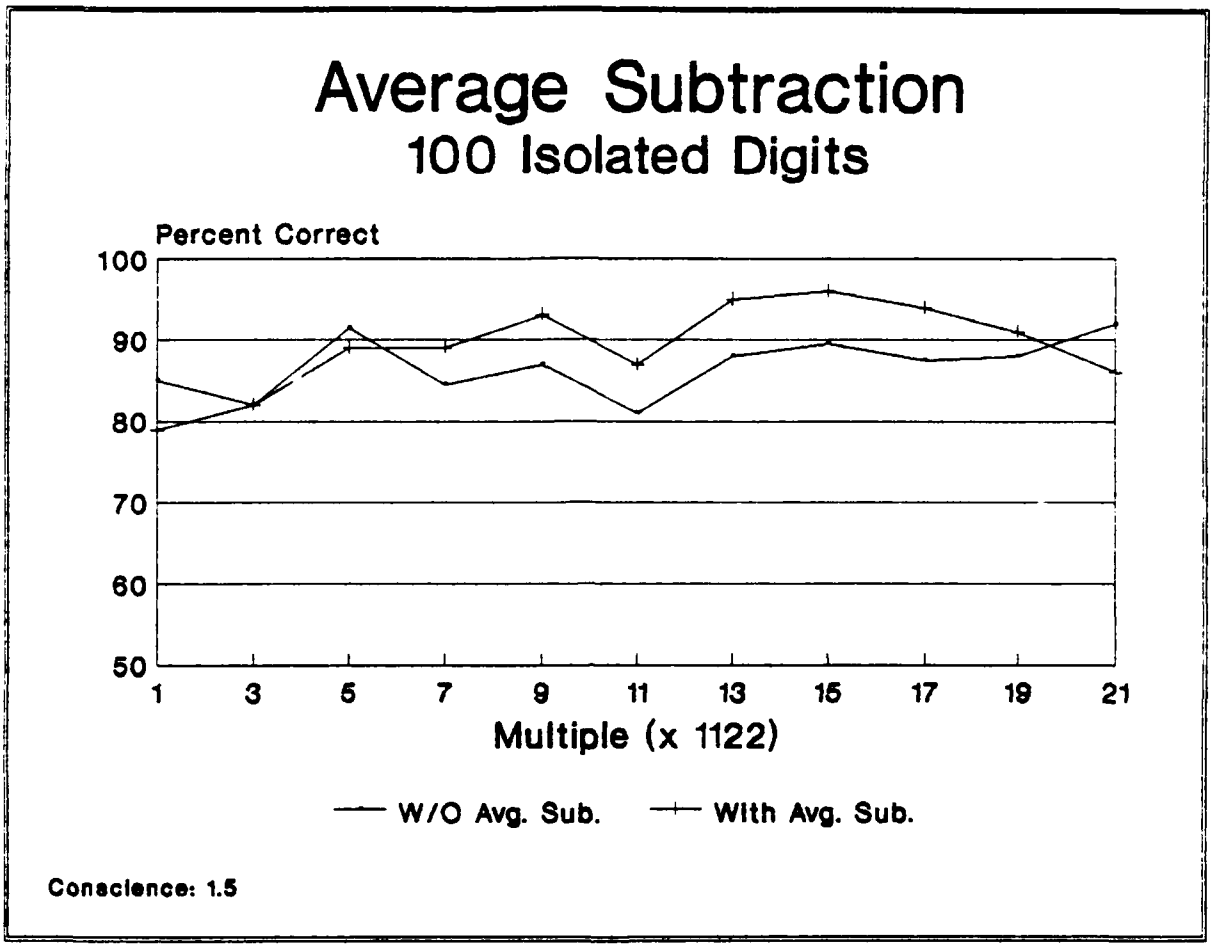

Figure 38. Isolated Word Average Subtraction Test Results (Conscience: 1.5)

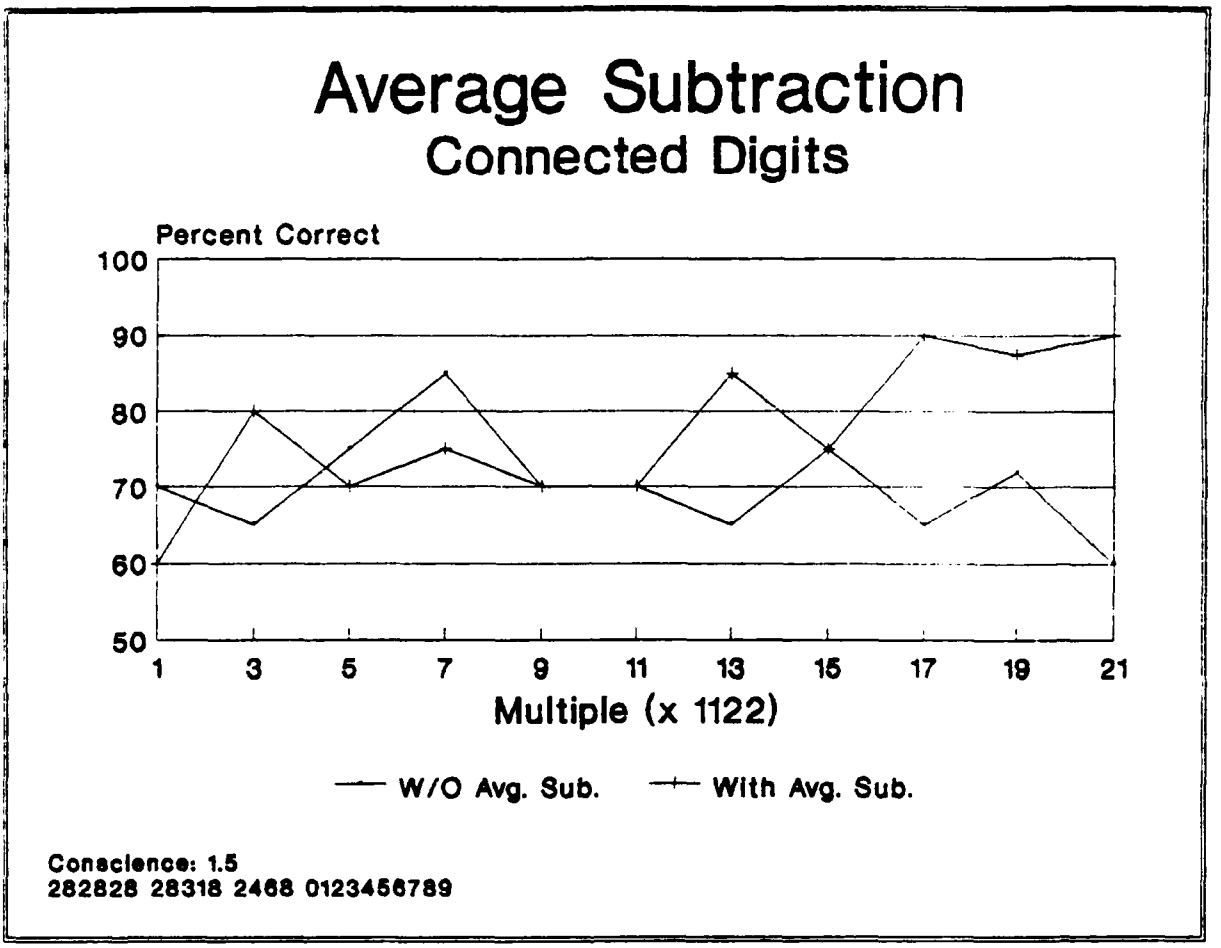

Figure 39. Connected Word Average Subtraction Test Results (Conscience: 1.5)

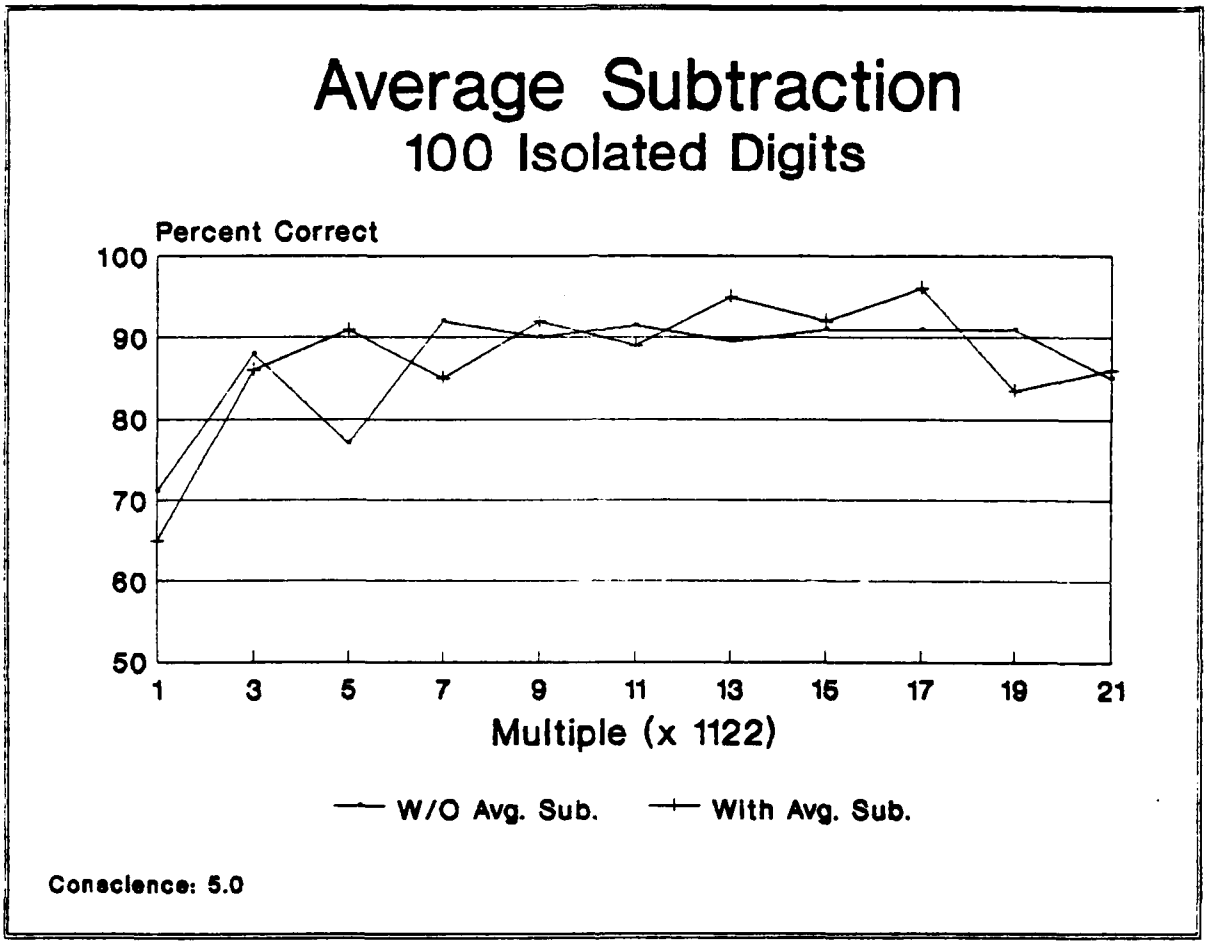

Figure 40. Isolated Word Average Subtraction Test Results (Conscience: 5.0)

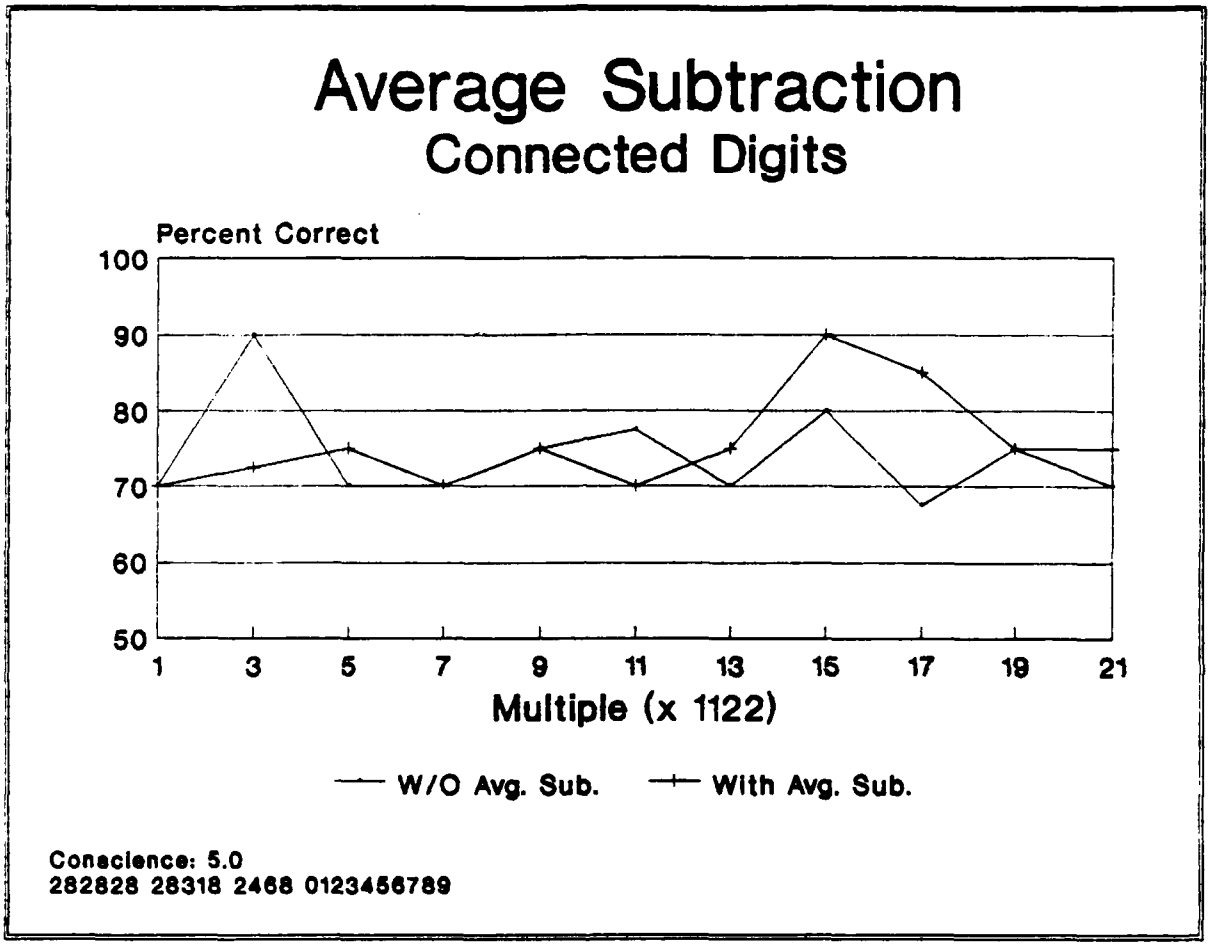

Figure 41. Connected Word Average Subtraction Test Results (Conscience: 5.0)

Table 13. Phase II Test Utterances (Connected Speech)

two eight two eight two eight
two four six eight
two eight three one eight
one seven four three
eight four six nine
five four three one
confirm cancel missile lock-on
report air-to-surface threat
report alpha bravo echo charlie delta
select gun strafe charlie
change frequency two four one

word utterances were used to test the ability of the system to recognize connected speech. The eleven connected word utterances are listed in Table 13.

Each test in Phase II was run on a series of eleven Kohonen networks in which each network was trained on either a multiple or half-multiple of the input data size (9,292). Two sets of tests were completed with each series of tests using a different data file. The number of iterative steps for each network in each set is shown in Table 14. Within each set of tests, LPC data was processed and fused with both $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ processed formant data files. The fusion process was defined in terms of a feature-fusion threshold. Only library words with distance factors less than the feature-fusion threshold value took part in the fusion process. For a feature-fusion threshold of one, all feature-fusion results defaulted to LPC output results.

Set #1 Test Results

Introduction The test results for $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants from the 210 isolated words and eleven sets of connected utterances are presented in both tabular and graphical form. The formants were processed only by dynamic

programming so that there was one average formant value listed for each series of 13 Kohonen networks. The appropriate average formant recognition accuracy is listed in the caption of each table and graph. Each table provides the range and average values for LPC and feature-fusion results within a series.

The results of the different feature-fusion thresholds were slightly different for the various neural networks, but the average value for each series of eleven networks was similar. Each feature-fusion threshold is plotted with its corresponding LPC input data represented in terms of recognition accuracy. For each series, an additional graph is presented which shows individual feature-fusion plots for a comparison between the different fusion threshold values.

Isolated Speech Figure 42 compares a feature-fusion t reshold of 1.5 with LPC data values using $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. With dynamic programming, the average $\mathcal{F}1, \mathcal{F}2$ formant accuracy was 89.3%, and the average $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formant accuracy was 84.0%.

The recognition accuracy for one-half multiple was 86.9% using only LPC data (see Figure 43). After performing feature-fusion, the accuracy for one-half multiple was 91.6% using $\mathcal{F}1, \mathcal{F}2$ formants. Note that the network trained on 0.5 multiple of the input data file was trained on only half of the words in the library. A high of 97.1% was obtained when the Kohonen network had seen the data set seven times. Using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants, the half-multiple recognition accuracy was 92.9% with a high recognition accuracy of 95.2% obtained when the data set was seen six times. The use of feature-fusion increased the isolated word recognition accuracy between 3% and 8%. Table 15 shows the range and average values using $\mathcal{F}1, \mathcal{F}2$ formants.

Figure 43 and Figure 44 compare feature-fusion thresholds of 2.0 and 2.5 with LPC data values using $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. The highest one-half multiple recognition accuracy was 95.2% using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants and a

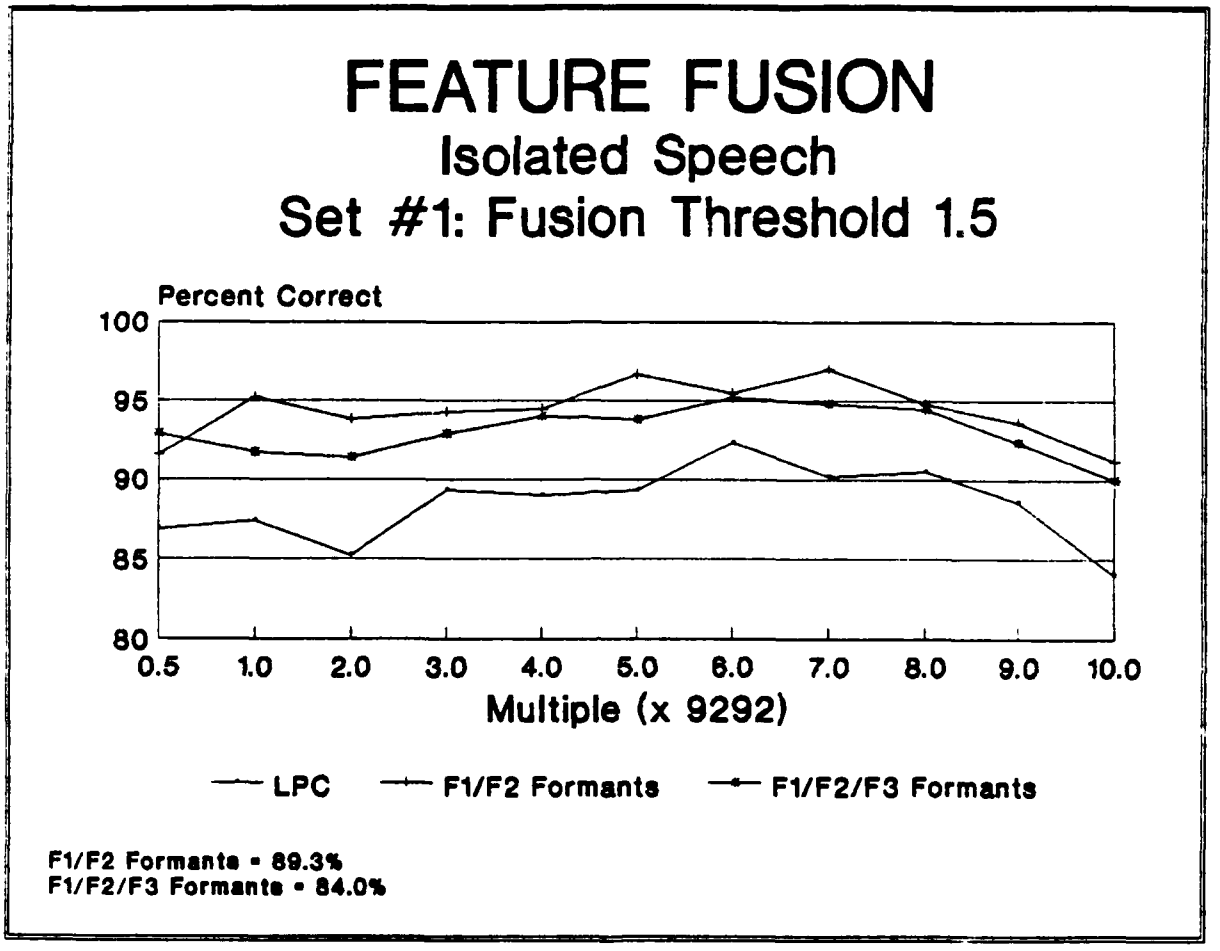

Figure 42. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 1.5, LPC, and Formant Data

fusion threshold of 2.5. It is interesting to note that this value was the highest recognition rate achieved in the series of Kohonen networks evaluated at a fusion threshold of 2.5 using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. Table 16 shows the range and average values using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants.

A comparison of individual feature-fusion threshold results is shown in Figure 45 for $\mathcal{F}1, \mathcal{F}2$ formants and Figure 46 for $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. The feature-fusion routine gave recognition rates above 90% on every test. All three fusion thresholds had similar average recognition accuracies of about 94% using $\mathcal{F}1, \mathcal{F}2$ formants and 93% using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants..

The average feature-fusion recognition accuracy was slightly lower (1%) using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants when compared to $\mathcal{F}1, \mathcal{F}2$ formants. The lower accuracy is probably due to the lower formant accuracy for $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants (84% vs 89%).

Connected Speech The recognition accuracies for connected speech were *lower than for isolated speech*. See Table 17 and Table 18 for the results on both formant representations. The average LPC recognition accuracy was 60.1%. The average formant recognition accuracies were very low. $\mathcal{F}1, \mathcal{F}2$ recognition accuracy was 28.6%, and the $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ recognition accuracy was 20.4%. The feature-fusion routine, which combines the LPC and formant features of speech, had outputs that reflected the average accuracy rates of the individual input features.

Figure 47 shows LPC recognition results compared with a feature-fusion threshold of 1.5 for $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. The low formant accuracy actually caused a *negative* fusion, where the average fusion recognition accuracy was 10% less than the average LPC recognition accuracy. The feature-fusion recognition accuracy was 50.8% for a feature-fusion threshold of 1.5. The fusion results for using the other thresholds gave slightly lower recognition accuracies. Figure 48 compares the individual feature-fusion threshold results for $\mathcal{F}1, \mathcal{F}2$ formants.

The recognition system displays the top choices (user specified) instead of just

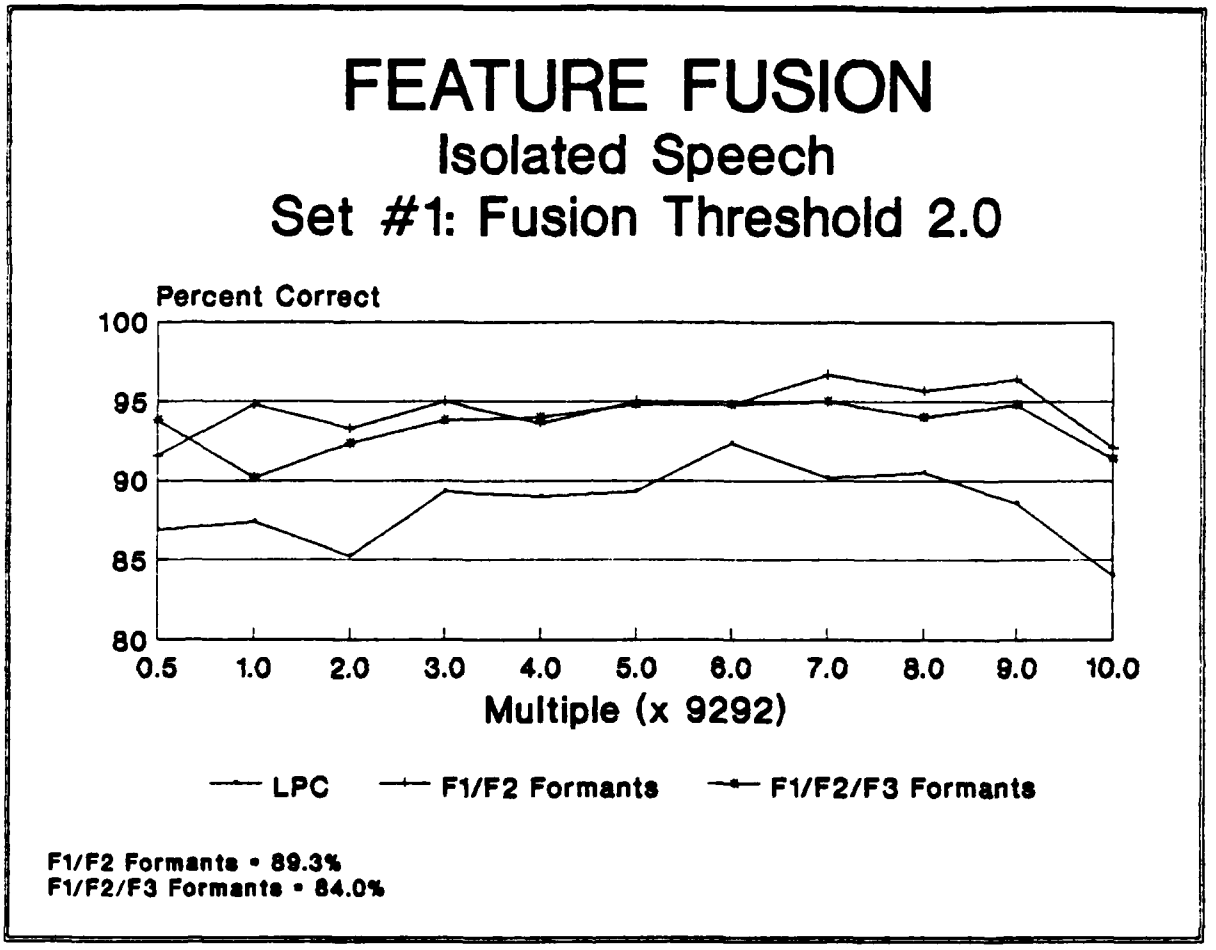

Figure 43. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 2.0, LPC, and Formant Data

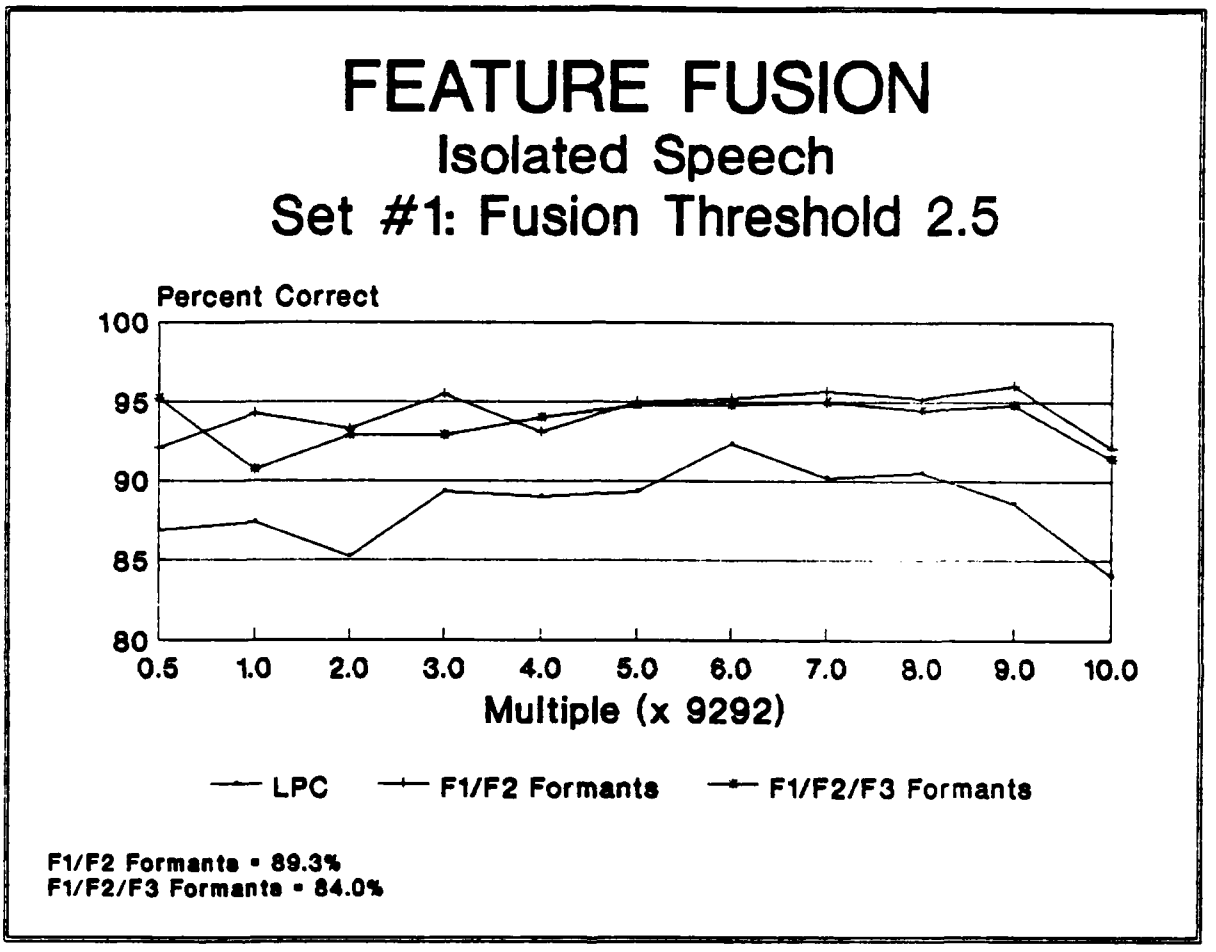

Figure 44. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 2.5, LPC, and Formant Data

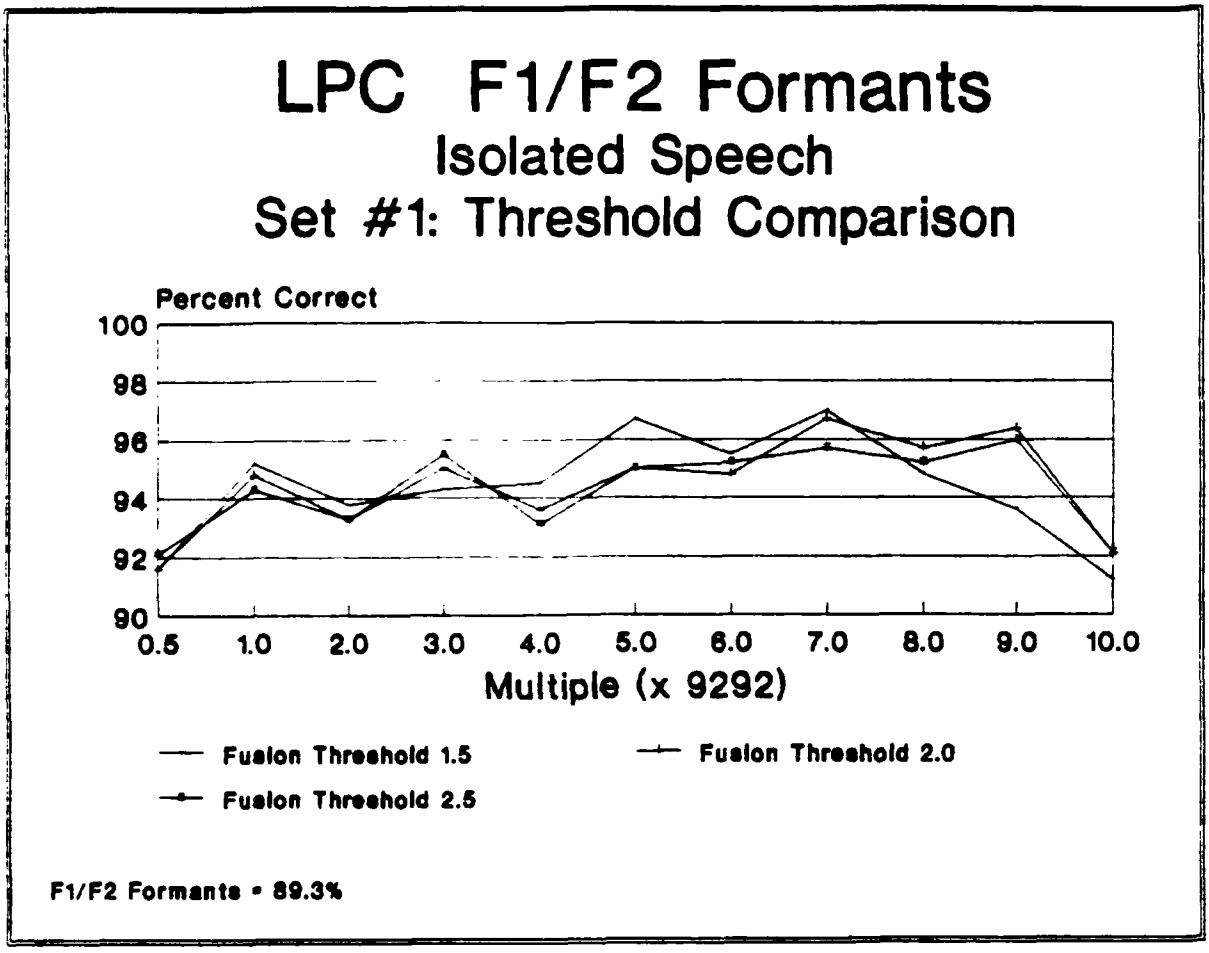

Figure 45. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech for Feature-Fusion Threshold Values of 1.5, 2.0 and 2.5 Using $\mathcal{F}1, \mathcal{F}2$ Formants

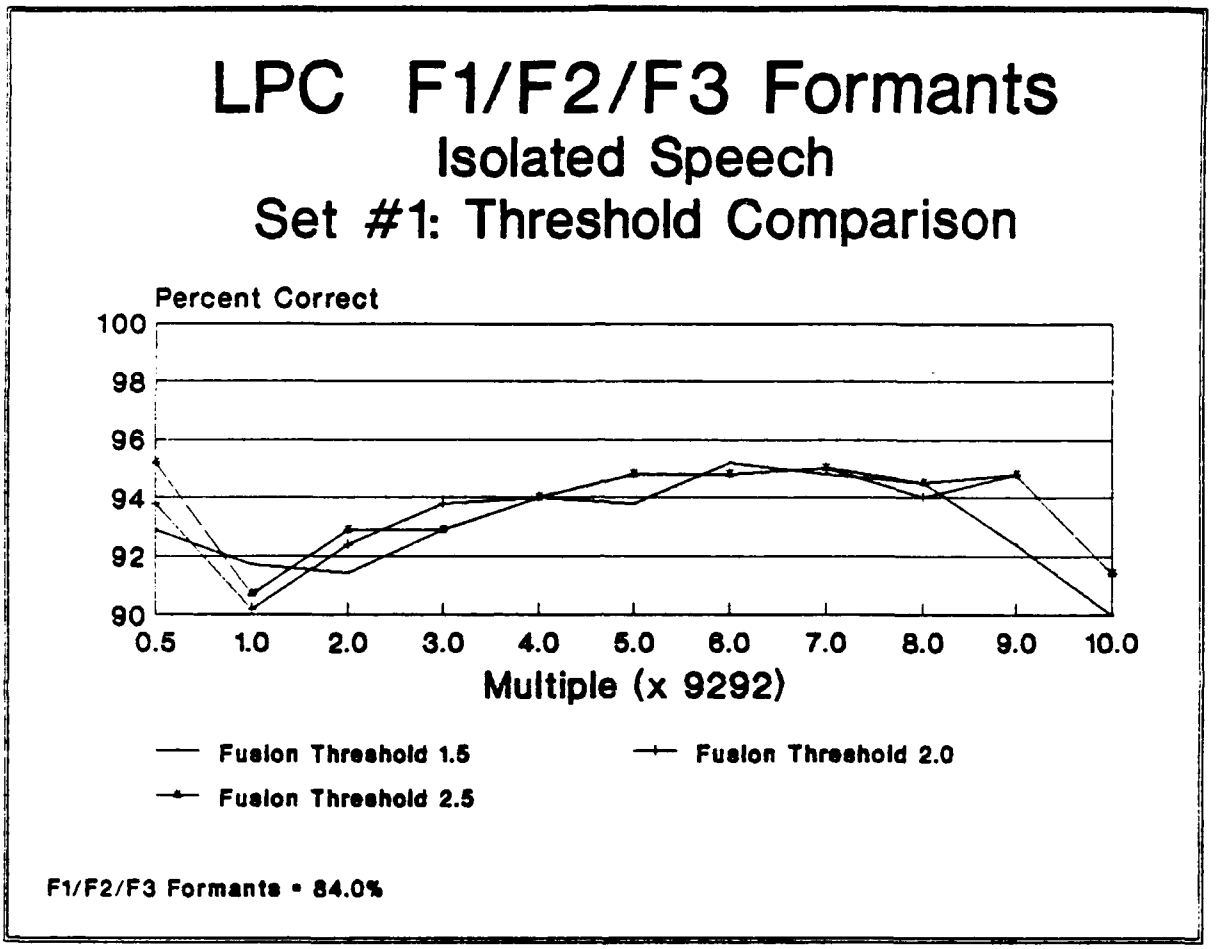

Figure 46. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech for Feature-Fusion Threshold Values of 1.5, 2.0 and 2.5 Using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants

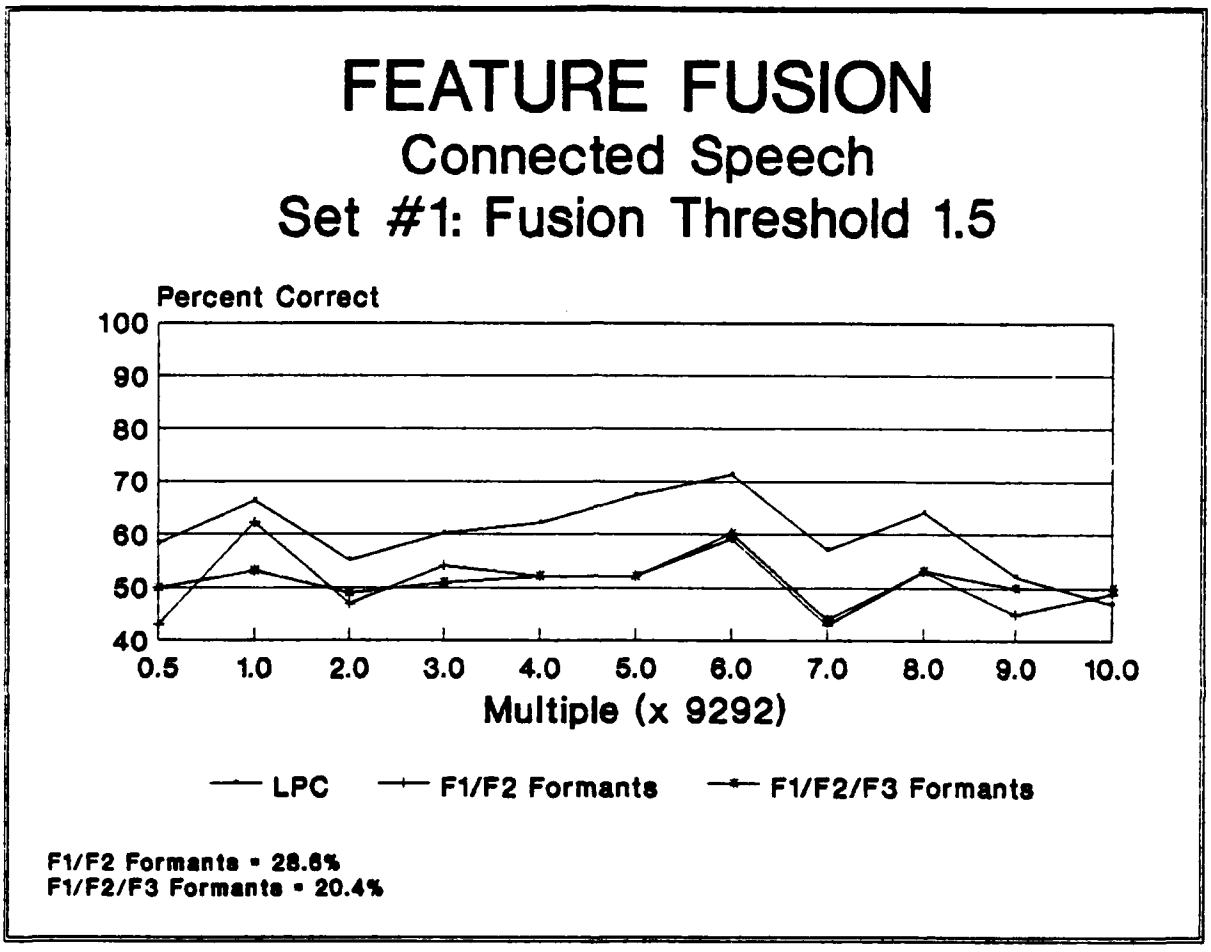

Figure 47. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech Using a Feature-Fusion Threshold of 1.5, LPC, and Formant Data

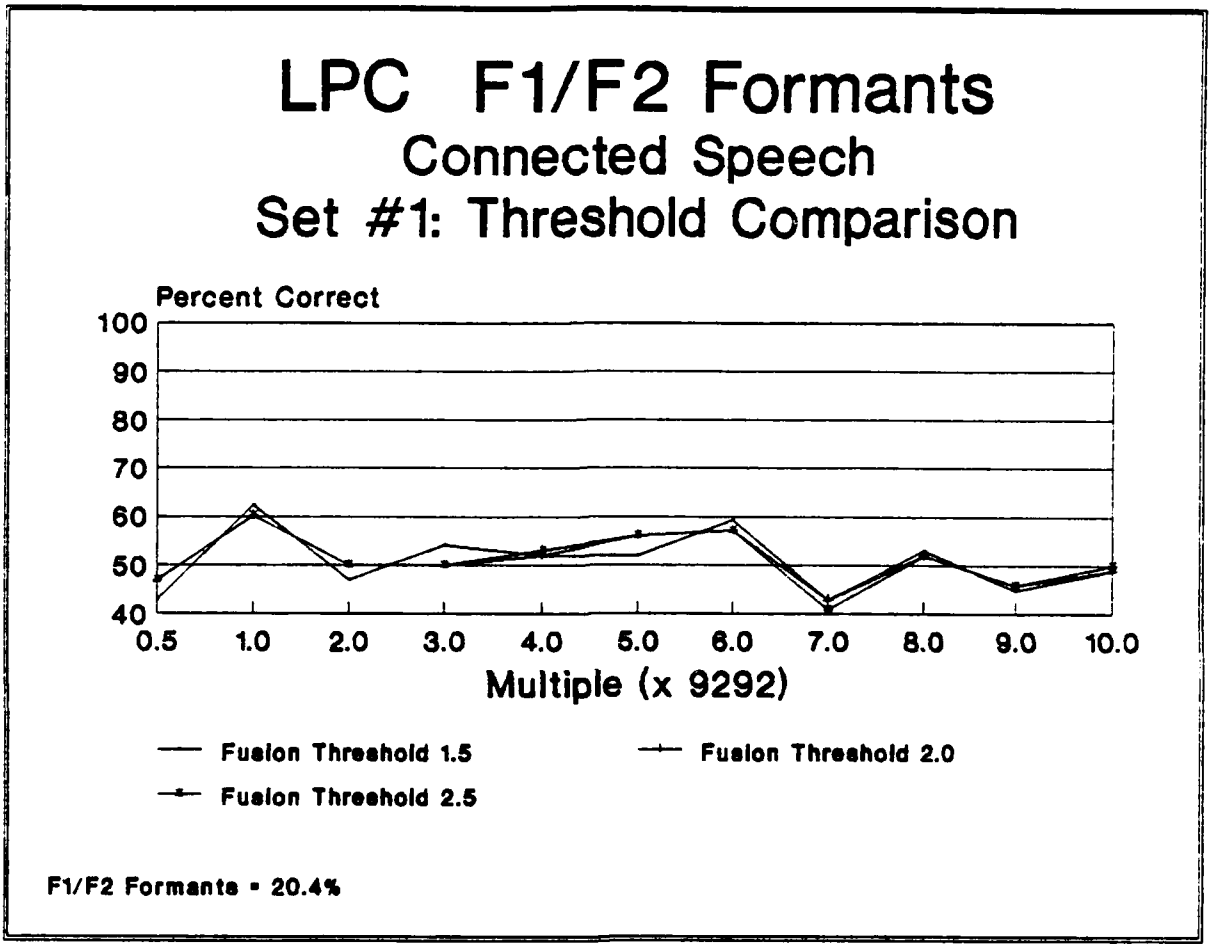

Figure 48. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech for Feature-Fusion Threshold Values of 1.5, 2.0 and 2.5 Using $\mathcal{F}1, \mathcal{F}2$ Formants

the best choice. Analysis of system outputs showed that the separation between dynamic programming distances was much smaller with connected speech than with isolated speech. The smaller separation distance was caused by the blending or overlapping of sound features between words (coarticulation)[25:92]. Another set of tests were conducted using smaller fusion thresholds to counter the effects of coarticulation. The threshold values chosen were 1.05, 1.1, and 1.2. The results are shown in Table 19 for $\mathcal{F}1, \mathcal{F}2$ formants and Table 20 for $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants.

Table 17 thru Table 20 show that as the fusion threshold value decreased toward one, feature-fusion accuracy increased. The increase in accuracy was mostly artificial, since the fusion output defaulted to the LPC output as the fusion threshold approached one. The need for a lower fusion threshold is probably needed with connected speech because of the effects of coarticulation. It is interesting to note that there were a few cases in which feature-fusion increased the LPC accuracy using a fusion threshold of 1.05. Figure 49 shows a feature-fusion threshold of 1.05 plotted against LPC data for both formant sets.

Set #2 Test Results

Introduction In an effort to improve overall recognition accuracy, the range of raw formant data files were evaluated. A few words were found to vary significantly in formant range from other words of the same kind. Nine out of the 350 words in the system were replaced. Three of these words, *map*, *mark*, and *three* were replaced from the set of 70 words used to train the Kohonen networks. A new data file was created to test the Kohonen networks. It had a multiple of 9,454. This data file was used in the Set #2 series of tests. The iteration cycle times for the networks trained in Set #2 are shown in Table 14. The same set of tests performed in Set #1 are performed in Set #2 testing. The presentation of results is also the same as in Set #1.

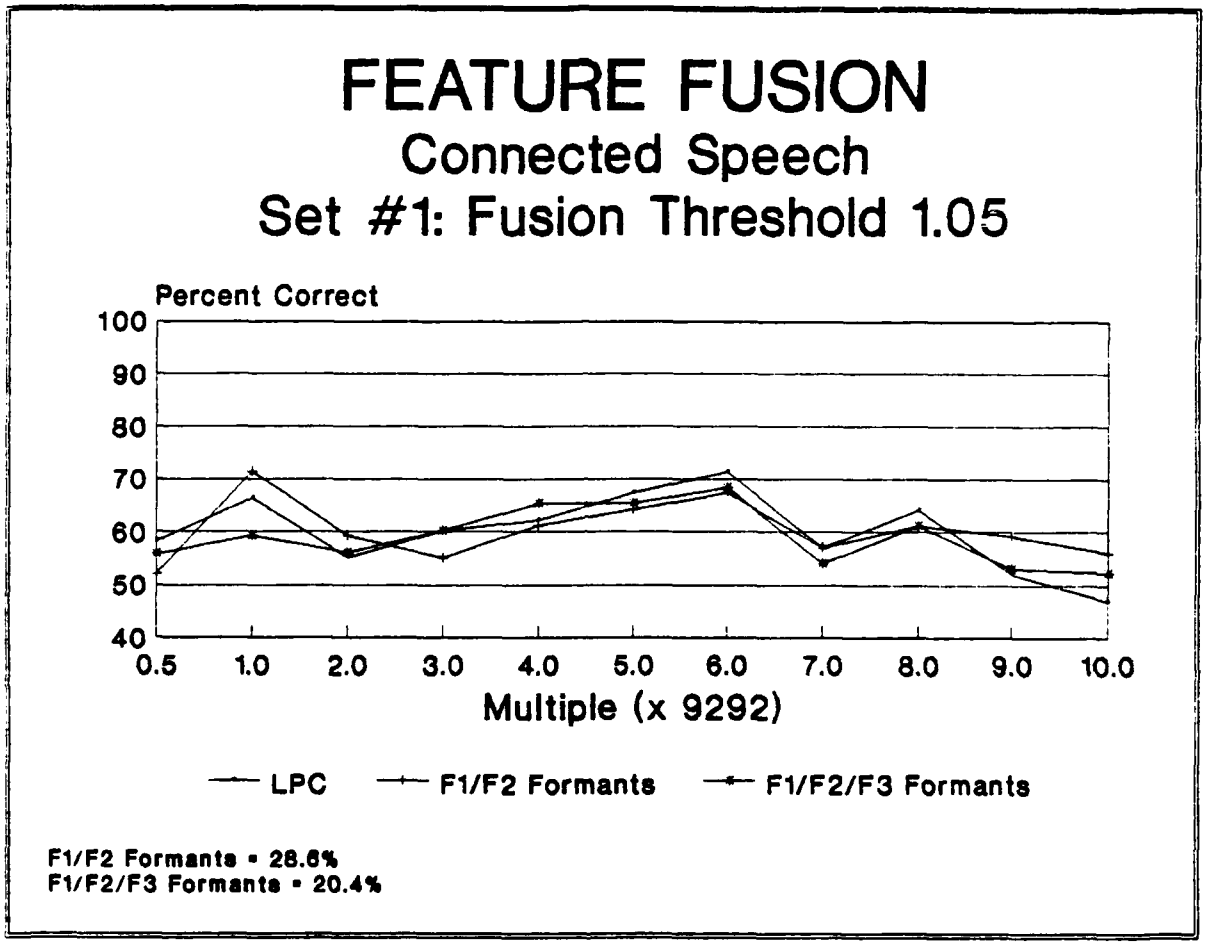

Figure 49. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech Using a Feature-Fusion Threshold Value of 1.05, LPC, and Formant Data

Isolated Speech Table 21 and Table 22 show the range and average recognition accuracies using $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. The average formant recognition accuracies were 88.1% and 87.9%. The formant accuracies are similar to Set #1 values. The average LPC recognition accuracy, however, increased from 88.4% to 94.1%.

The fusion-factor test results for all thresholds values were excellent. Both $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants had an average recognition accuracy around 97%. The three fusion thresholds are plotted against LPC data values and formant data in Figure 50, Figure 51, and Figure 52. Using a fusion threshold of 2.5, the feature-fusion recognition accuracy for one-half multiple was 97.1% with both $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formant data sets. A high of 98.3% was obtained for both eight and nine multiples using $\mathcal{F}1, \mathcal{F}2$ formants. With $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants, a high of 98.6% was obtained using fusion thresholds of both 2.0 and 2.5 for multiples of nine and ten.

Figure 53 and Figure 54 show that every fusion recognition accuracy was above 95%. A feature-fusion threshold of 50 was also evaluated with $\mathcal{F}1, \mathcal{F}2$ formants to see if the recognition rate would improve by making all 70 words in the library eligible for fusion. A fusion threshold of 50 gave a range from 95.7% to 97.4%, and an average accuracy of 96.7%. These results are slightly less than the values obtained for fusion thresholds of 2.0 and 2.5 using $\mathcal{F}1, \mathcal{F}2$ formants.

Connected Speech The average LPC recognition accuracy increased from 60% to 70%. The highest LPC recognition accuracy was 83.7% for four multiples. The average formant recognition accuracies for both $\mathcal{F}1, \mathcal{F}2$ formants and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants, however, were about 20%. These rates are slightly less than the values obtained in Set #1. These low formant recognition rates precluded use of the formant routine with connected speech. Table 23 and Table 24 show the range and average recognition accuracies using $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ for-

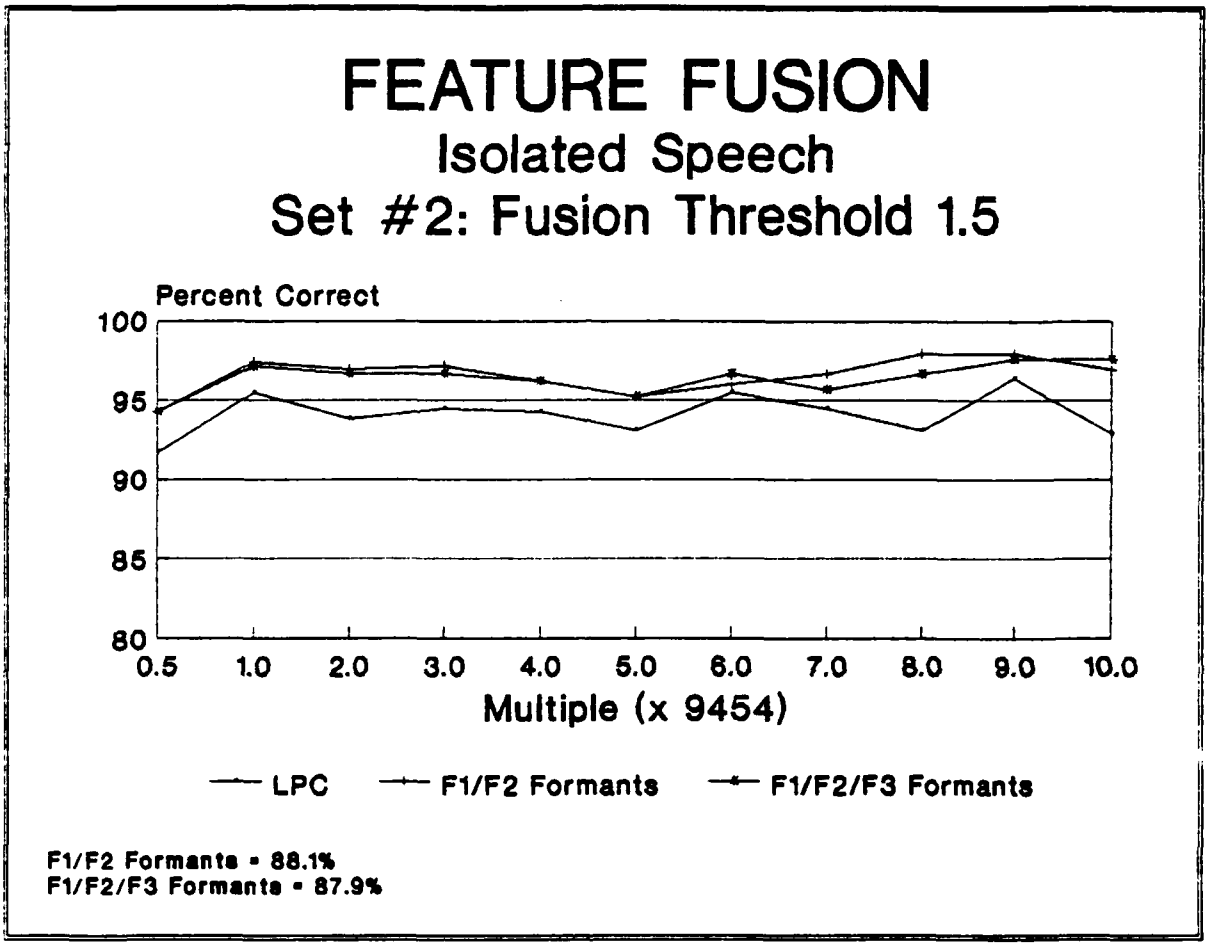

Figure 50. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 1.5, LPC, and Formant Data

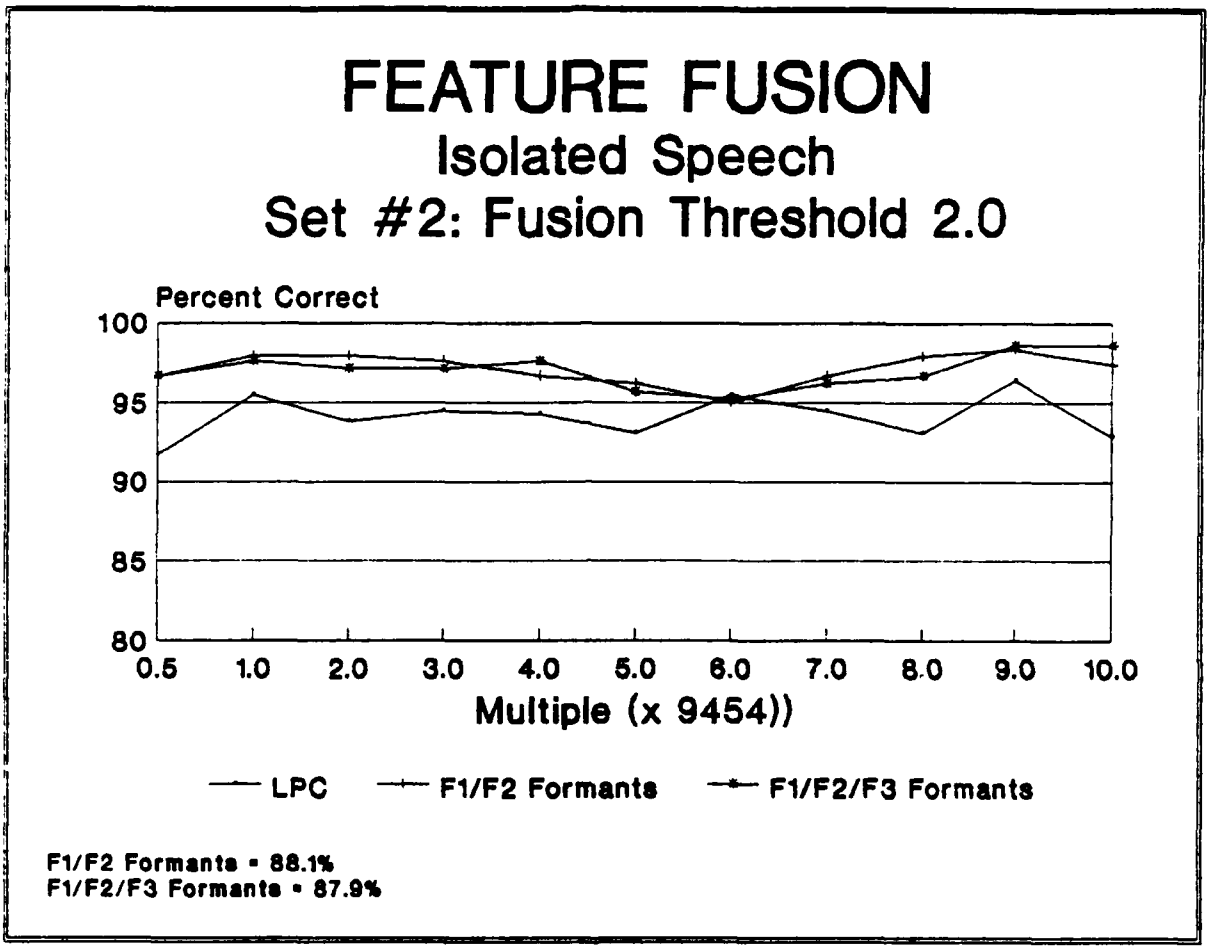

Figure 51. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 2.0, LPC, and Formant Data

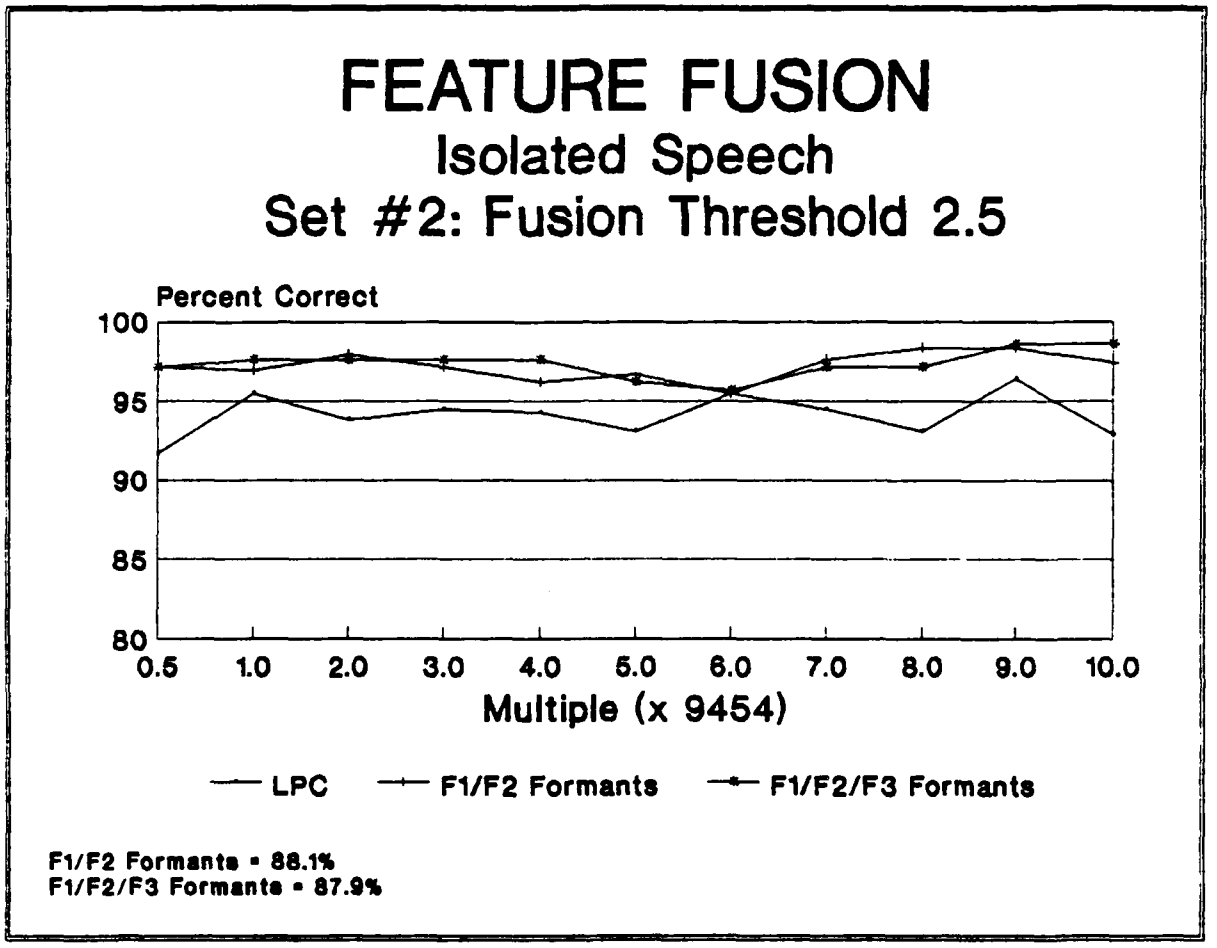

Figure 52. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech Using a Feature-Fusion Threshold of 2.5, LPC, and Formant Data

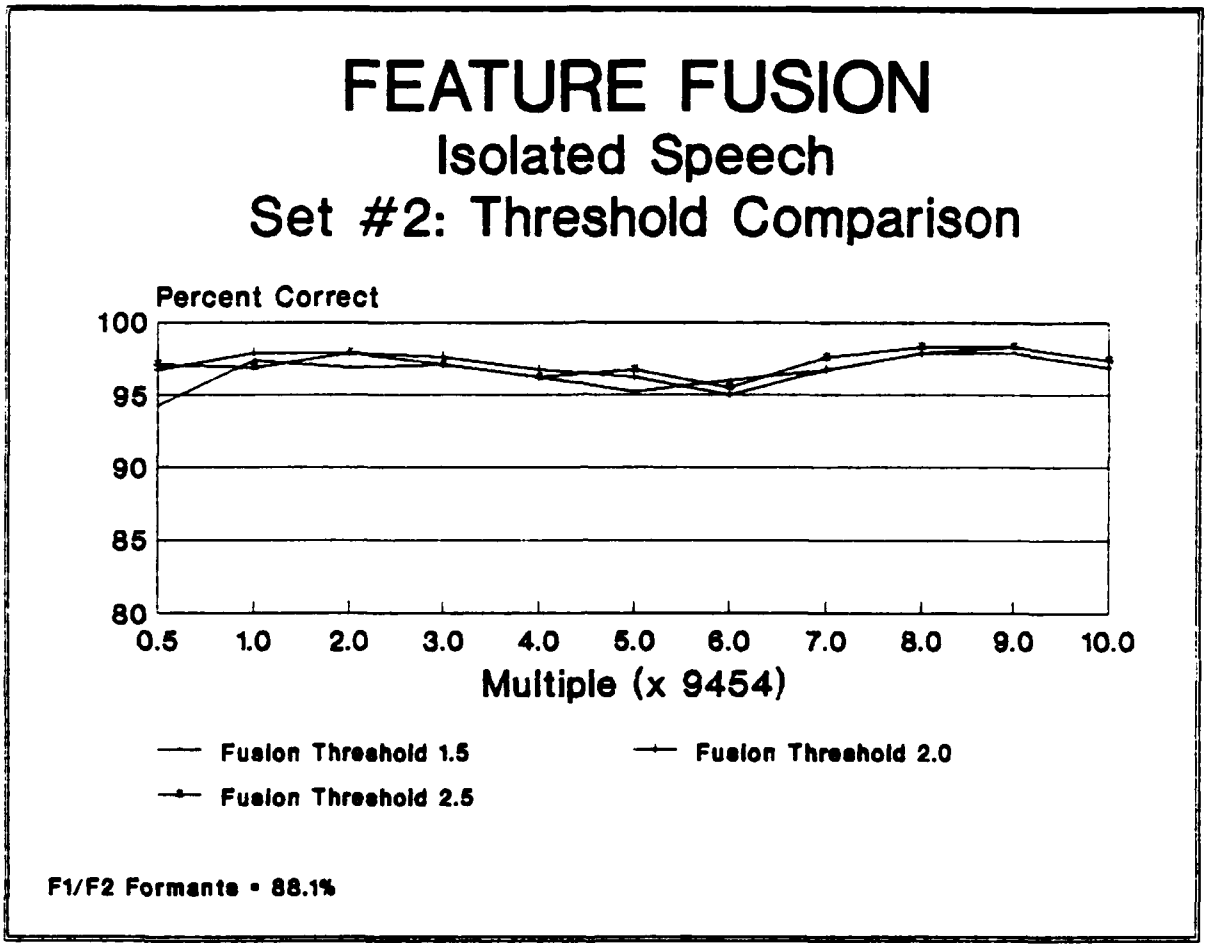

Figure 53. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech for Feature-Fusion Threshold Values of 1.5, 2.0 and 2.5 Using $\mathcal{F}1, \mathcal{F}2$ Formants

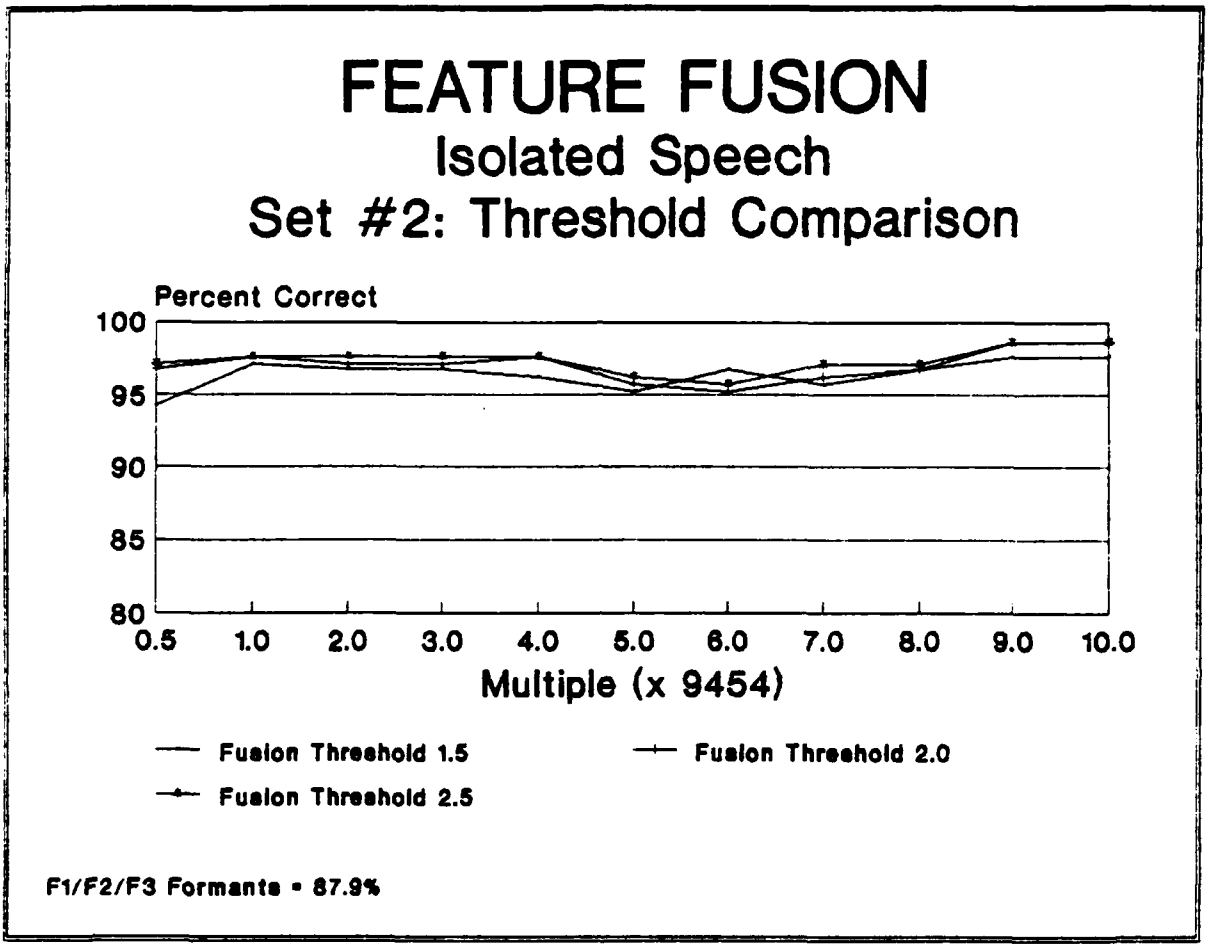

Figure 54. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech for Feature-Fusion Threshold Values of 1.5, 2.0 and 2.5 Using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants

mants. Figure 55 shows LPC recognition results plotted against a fusion threshold value of 1.05 for both $\mathcal{F}1, \mathcal{F}2$ and $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. Even with a low formant recognition rate, Figure 55 shows a few cases in which the feature-fusion results are higher than the LPC recognition rate. Figure 56 shows all three formant thresholds plotted for $\mathcal{F}1, \mathcal{F}2$ formants.

Feature-Fusion Display Results

This section provides seven examples of feature-fusion. Each example shows the display results that are normally sent to the display terminal from the recognition system. The recognition results for each word are displayed over two pages for uncluttered viewing. The first page displays the recognition results for both features (LPC and formants), and feature-fusion using a threshold of 1.5. The second page displays feature-fusion results for thresholds of 2.0 and 2.5. The displayed results are for isolated speech. The seven examples (using seven words) were chosen to show the capabilities of the feature-fusion section and rule-based system. For each example, the correct test word is identified followed by a description of the display results.

- **negative**(page 124): Quite often, when LPC-based dynamic programming gave the incorrect answer, the correct word was the second or third choice. This example shows that case. The LPC feature result was **map**, and the correct word, **negative**, was the next best choice. The formant result was correct. After fusion, the correct choice by the recognition system was made.
- **four**(page 126): In this example, the LPC result was correct, but the formant result was incorrect. Notice the effect of the threshold. **Four** was the only word considered within the feature-fusion section, because the second best choice within the LPC feature, **forward**, had a distance factor of 4.9882.

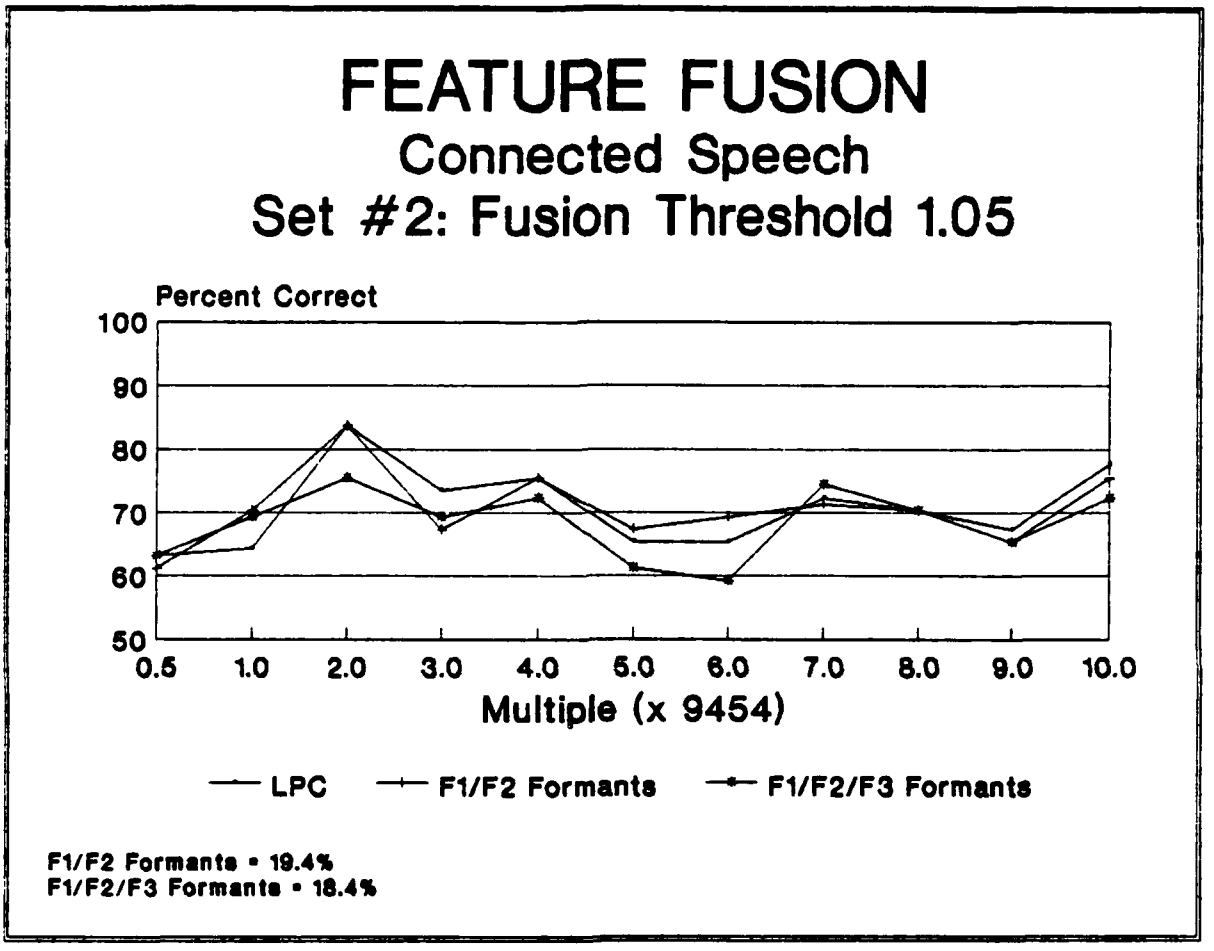

Figure 55. Phase II, Set #2 Feature-Fusion Test Results on Connected Speech Using a Feature-Fusion Threshold of 1.05, LPC, and Formant Data

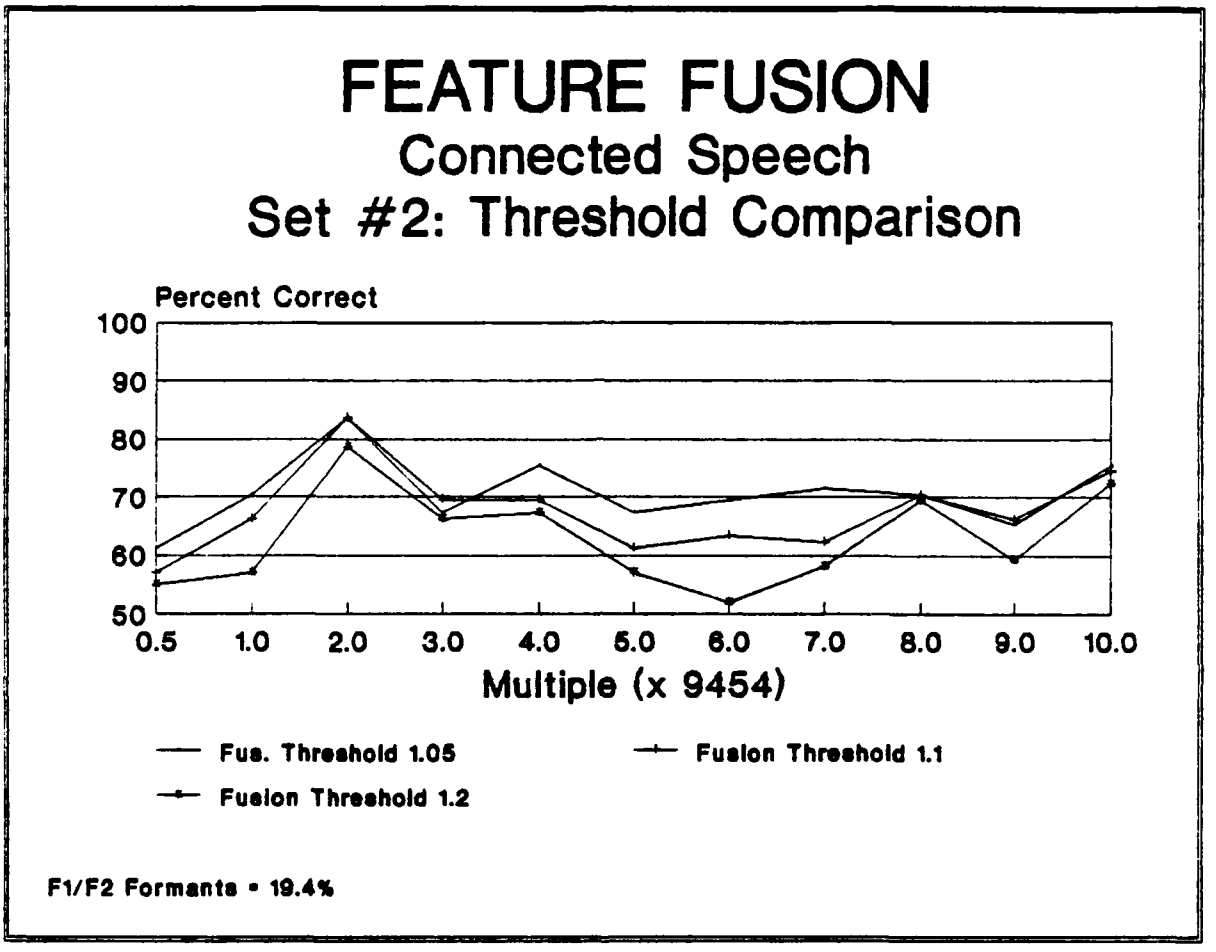

Figure 56. Phase II, Set #2 Feature-Fusion Test Results on Connected Speech for Feature-Fusion Threshold Values of 1.05, 1.1 and 1.2 Using $\mathcal{F}1, \mathcal{F}2$ Formants

- **north**(page 128): The word **north** had a dynamic programming distance of 324.0, which was much larger than the LPC best choice of 221.0 for **nose**. In fact, **north** was the seventh best choice. The formant feature gave the correct result. Feature-fusion of the two features gave the correct result for all three thresholds.

- **cancel**(page 130): In this example, the LPC feature result gave two words when there was only one word spoken. Using only the LPC feature, the recognizer thought the utterance was **cancel south**. The formant result was correct. The rule-based system was used in this example. Since one of the two LPC words matched the formant word, the rule-based system deleted the word **south**. The feature-fusion section only considered the LPC word **cancel** with the formant result.

- **delta**(page 132): This example is another case in which the LPC feature result thought there were two words when only one word was spoken. Neither word matches the correct word which was identified using the formant feature. The rule-based section added the LPC results for every word. The feature-fusion section multiplied this result and the corresponding formant result. The word **delta** was correctly identified in each feature-fusion result.

- **three**(page 134): The real power of feature-fusion is shown in the next two examples. In both cases, each individual feature gave the incorrect result, but feature-fusion gave the correct result. In the first example, the word **three** was output as **charlie** using the LPC feature, and output as **degrees** using formants.

- **threat**(page 136): This example is the second case in which both features incorrectly identified the test word but feature-fusion identified the correct word. The word **threat** was output as **search** using the LPC feature and **west** using formants. Feature-fusion identified the word as **threat** for each threshold.

Table 14. Phase II Training Schedule

Number of Multiples	*Training Iterations*	
	Set #1	*Set #2*
0.5	4,646	4,727
1.0	9,292	9,454
2.0	18,584	18,908
3.0	27,876	28,362
4.0	37,168	37,816
5.0	46,460	47,270
6.0	55,752	56,724
7.0	65,044	66,178
8.0	74,336	75,632
9.0	83,628	85,086
10.0	92,920	94,540

Table 15. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech ($\mathcal{F}1, \mathcal{F}2$ Formants = 89.3%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(84-92)	88.4	(91-97)	94.4	(91-96)	94.4	(92-96)	94.3

Table 16. Phase II, Set #1 Feature-Fusion Test Results on Isolated Speech ($\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants = 84.0%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(84-92)	88.4	(90-95)	93.1	(90-95)	93.5	(91-95)	93.7

Table 17. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech ($\mathcal{F}1, \mathcal{F}2$ Formants = 28.6%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(47-71)	60.1	(43-62)	50.8	(43-60)	51.1	(41-60)	51.1

Table 18. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech ($\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants = 20.4%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(47-71)	60.1	(44-60)	51.3	(44-56)	48.7	(44-56)	48.9

Table 19. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech with Fusion Threshold Values of 1.05, 1.1, and 1.2.($\mathcal{F}1, \mathcal{F}2$ Formants = 28.6%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.05		1.1		1.2	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(47-71)	60.1	(52-71)	60.3	(48-66)	57.5	(44-64)	54.1

Table 20. Phase II, Set #1 Feature-Fusion Test Results on Connected Speech with Fusion Threshold Values of 1.05, 1.1, and 1.2.($\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants = 20.4%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.05		1.1		1.2	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(47-71)	60.1	(52-68)	59.2	(48-64)	56.9	(45-61)	52.7

Table 21. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech ($\mathcal{F}1, \mathcal{F}2$ Formants = 88.1%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(92-96)	94.1	(94-98)	96.6	(95-98)	97.1	(96-98)	97.2

Table 22. Phase II, Set #2 Feature-Fusion Test Results on Isolated Speech ($\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants = 87.9%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(92-96)	94.1	(94-98)	96.4	(95-99)	97.0	(96-99)	97.4

Table 23. Phase II, Set #2 Feature-Fusion Test Results on Connected Speech ($\mathcal{F}1, \mathcal{F}2$ Formants = 18.4%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.05		1.1		1.2	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(63-84)	70.8	(61-83)	70.7	(57-83)	67.6	(52-78)	63.0

Table 24. Phase II, Set #2 Feature-Fusion Test Results on Connected Speech ($\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ Formants = 19.4%)

LPC (%)		*Feature-Fusion Threshold (%)*					
		1.5		2.0		2.5	
Range	*Avg*	*Range*	*Avg*	*Range*	*Avg*	*Range*	*Avg*
(63-84)	70.8	(59-76)	68.5	(51-75)	62.9	(45-72)	57.3

FEATURE-FUSION: NEGATIVE_(#17)

```
 lpc7_wr2_17.trn is:  23
   Should be:  17
    Best choices are:
       MAP had a lpc distance of 438.00 and a distance factor of 1.0000
       NEGATIVE had a lpc distance of 455.00 and a distance factor of 1.0388
       SEVEN had a lpc distance of 479.00 and a distance factor of 1.0936
   correct = 0.000    cum_correct = 0.765
Total wrong lpc digits are 4.00 from a total of 17.00 digits


fmc7_wr2_17.trn is 152 vectors long
 fmc7_wr2_17.trn is:  17
  Should be:  17
   Best choices are:
    NEGATIVE had a formant distance of 17690.60 and a distance factor of 1.0000
    SIX had a formant distance of 21742.70 and a distance factor of 1.2291
    KNOTS had a formant distance of 24919.81 and a distance factor of 1.4086
    correct = 1.000    cum_correct = 0.853
 Total wrong formant digits are 2.50 from a total of 17.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
 The computed utterance string is:  17
     Should be:  17
The # of words less than 1.5 are 36

     Best choices are:
        NEGATIVE had a formant-lpc distance factor of 1.0388
        MAP had a formant-lpc distance factor of 1.5027
        SIX had a formant-lpc distance factor of 1.7159
        CANCEL had a formant-lpc distance factor of 1.7959
        MISSILE had a formant-lpc distance factor of 1.9353
        NORTH had a formant-lpc distance factor of 1.9629
        ENTER had a formant-lpc distance factor of 1.9914
        SEVEN had a formant-lpc distance factor of 2.1517
        HEADING had a formant-lpc distance factor of 2.1741
        AFT had a formant-lpc distance factor of 2.2610
    correct = 1.000    cum_correct = 0.941
 Total wrong fusion digits are 1.00 from a total of 17.00 digits
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  17
     Should be:  17
The # of words less than 2.0 are 69

     Best choices are:
        NEGATIVE had a formant-lpc distance factor of 1.0388
        MAP had a formant-lpc distance factor of 1.5027
        SIX had a formant-lpc distance factor of 1.7159
        CANCEL had a formant-lpc distance factor of 1.7959
        MISSILE had a formant-lpc distance factor of 1.9353
        NORTH had a formant-lpc distance factor of 1.9629
        ENTER had a formant-lpc distance factor of 1.9914
        SEVEN had a formant-lpc distance factor of 2.1517
        HEADING had a formant-lpc distance factor of 2.1741
        AFT had a formant-lpc distance factor of 2.2610
    correct = 1.000     cum_correct = 0.941
 Total wrong fusion digits are 1.00 from a total of 17.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  17
     Should be:  17
The # of words less than 2.5 are 71

     Best choices are:
        NEGATIVE had a formant-lpc distance factor of 1.0388
        MAP had a formant-lpc distance factor of 1.5027
        SIX had a formant-lpc distance factor of 1.7159
        CANCEL had a formant-lpc distance factor of 1.7959
        MISSILE had a formant-lpc distance factor of 1.9353
        NORTH had a formant-lpc distance factor of 1.9629
        ENTER had a formant-lpc distance factor of 1.9914
        SEVEN had a formant-lpc distance factor of 2.1517
        HEADING had a formant-lpc distance factor of 2.1741
        AFT had a formant-lpc distance factor of 2.2610
    correct = 1.000     cum_correct = 0.941
 Total wrong fusion digits are 1.00 from a total of 17.00 digits
```

FEATURE-FUSION: FOUR_(#4)

```
 lpc7_wr2_4.trn is:  4
   Should be:  4
    Best choices are:
       FOUR had a lpc distance of 42.25 and a distance factor of 1.0000
       FORWARD had a lpc distance of 210.75 and a distance factor of 4.9882
       SPACE had a lpc distance of 231.25 and a distance factor of 5.4734
   correct = 1.000    cum_correct = 0.700
Total wrong lpc digits are 1.50 from a total of 5.00 digits


fmc7_wr2_4.trn is 182 vectors long
 fmc7_wr2_4.trn is:  60
  Should be:  4
   Best choices are:
    FORWARD had a formant distance of 16329.25 and a distance factor of 1.0000
    MARK had a formant distance of 16860.30 and a distance factor of 1.0325
    FOUR had a formant distance of 20132.60 and a distance factor of 1.2329
  correct = 0.000    cum_correct = 0.600
Total wrong formant digits are 2.00 from a total of 5.00 digits


The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
The computed utterance string is:  4
    Should be:  4
The # of words less than 1.5 are 1

    Best choices are:
       FOUR had a formant-lpc distance factor of 1.2329
   correct = 1.000    cum_correct = 0.800
Total wrong fusion digits are 1.00 from a total of 5.00 digits
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  4
     Should be:  4
The # of words less than 2.0 are 1

     Best choices are:
        FOUR had a formant-lpc distance factor of 1.2329
    correct = 1.000     cum_correct = 0.800
 Total wrong fusion digits are 1.00 from a total of 5.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  4
     Should be:  4
The # of words less than 2.5 are 1

     Best choices are:
        FOUR had a formant-lpc distance factor of 1.2329
    correct = 1.000     cum_correct = 0.800
 Total wrong fusion digits are 1.00 from a total of 5.00 digits
```

```
                    FEATURE-FUSION: NORTH_(#57)

  lpc7_wr3_57.trn is:  45
    Should be:  57
     Best choices are:
        NOSE had a lpc distance of 221.00 and a distance factor of 1.0000
        KNOTS had a lpc distance of 229.50 and a distance factor of 1.0385
        MILES had a lpc distance of 266.00 and a distance factor of 1.2036
        MARK had a lpc distance of 278.25 and a distance factor of 1.2590
        SPACE had a lpc distance of 302.75 and a distance factor of 1.3699
        THREAT had a lpc distance of 322.75 and a distance factor of 1.4604
        NORTH had a lpc distance of 324.00 and a distance factor of 1.4661
    correct = 0.000     cum_correct = 0.906
 Total wrong lpc digits are 12.00 from a total of 127.00 digits


fmc7_wr3_57.trn is 84 vectors long
  fmc7_wr3_57.trn is:  57
    Should be:  57
     Best choices are:
        NORTH had a formant distance of 5354.80 and a distance factor of 1.0000
        MARK had a formant distance of 13771.05 and a distance factor of 2.5717
        FAULT had a formant distance of 15717.15 and a distance factor of 2.9352
    correct = 1.000     cum_correct = 0.898
 Total wrong formant digits are 13.00 from a total of 127.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
 The computed utterance string is:  57
     Should be:  57
The # of words less than 1.5 are 8

     Best choices are:
        NORTH had a formant-lpc distance factor of 1.4661
        MARK had a formant-lpc distance factor of 3.2379
        KNOTS had a formant-lpc distance factor of 3.9073
        NOSE had a formant-lpc distance factor of 4.1603
        THREAT had a formant-lpc distance factor of 5.0513
        MILES had a formant-lpc distance factor of 8.9762
        MINUS had a formant-lpc distance factor of 15.7931
        SPACE had a formant-lpc distance factor of 46.7062
    correct = 1.000     cum_correct = 0.949
    Total wrong fusion words are 6.50 from a total of 127.00 words
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  57
     Should be:  57
The # of words less than 2.0 are 18

     Best choices are:
        NORTH had a formant-lpc distance factor of 1.4661
        MARK had a formant-lpc distance factor of 3.2379
        KNOTS had a formant-lpc distance factor of 3.9073
        NOSE had a formant-lpc distance factor of 4.1603
        THREAT had a formant-lpc distance factor of 5.0513
        FAULT had a formant-lpc distance factor of 5.1697
        WEST had a formant-lpc distance factor of 6.6020
        POINT had a formant-lpc distance factor of 7.1808
        CANCEL had a formant-lpc distance factor of 8.8345
        MILES had a formant-lpc distance factor of 8.9762
    correct = 1.000     cum_correct = 0.937
    Total wrong fusion words are 8.00 from a total of 127.00 words


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  57
     Should be:  57
The # of words less than 2.5 are 32

     Best choices are:
        NORTH had a formant-lpc distance factor of 1.4661
        MARK had a formant-lpc distance factor of 3.2379
        KNOTS had a formant-lpc distance factor of 3.9073
        NOSE had a formant-lpc distance factor of 4.1603
        THREAT had a formant-lpc distance factor of 5.0513
        FAULT had a formant-lpc distance factor of 5.1697
        WEST had a formant-lpc distance factor of 6.6020
        POINT had a formant-lpc distance factor of 7.1808
        MISSILE had a formant-lpc distance factor of 7.7583
        SOUTH had a formant-lpc distance factor of 7.9360
    correct = 1.000     cum_correct = 0.929
    Total wrong fusion words are 9.00 from a total of 127.00 words
```

FEATURE-FUSION: CANCEL_(#33)

```
 lpc7_wr2_33.trn is:  33 47
   Should be:  33
    Best choices are:
       SOUTH had a lpc distance of 163.75 and a distance factor of 1.0000
       SPACE had a lpc distance of 205.25 and a distance factor of 1.2534
       TWO had a lpc distance of 276.50 and a distance factor of 1.6885
       ECHO had a lpc distance of 292.50 and a distance factor of 1.7863
    Best choices are:
       CANCEL had a lpc distance of 42.25 and a distance factor of 1.0000
       TAIL had a lpc distance of 218.75 and a distance factor of 5.1775
       SEVEN had a lpc distance of 260.50 and a distance factor of 6.1657
       HEADING had a lpc distance of 269.50 and a distance factor of 6.3787
   correct = 0.500     cum_correct = 0.818
Total wrong lpc digits are 6.00 from a total of 33.00 digits


fmc7_wr2_33.trn is 98 vectors long
 fmc7_wr2_33.trn is:  33
  Should be:  33
   Best choices are:
    CANCEL had a formant distance of 15739.49 and a distance factor of 1.0000
    FAULT had a formant distance of 19042.55 and a distance factor of 1.2099
    ECHO had a formant distance of 19775.15 and a distance factor of 1.2564
   correct = 1.000     cum_correct = 0.864
Total wrong formant digits are 4.50 from a total of 33.00 digits


The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
The computed utterance string is:  33
    Should be:  33
The # of words less than 1.5 are 1

    Best choices are:
       CANCEL had a formant-lpc distance factor of 1.0000
   correct = 1.000     cum_correct = 0.924
Total wrong fusion digits are 2.50 from a total of 33.00 digits
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  33
     Should be:  33
The # of words less than 2.0 are 1

     Best choices are:
        CANCEL had a formant-lpc distance factor of 1.0000
    correct = 1.000     cum_correct = 0.924
 Total wrong fusion digits are 2.50 from a total of 33.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  33
     Should be:  33
The # of words less than 2.5 are 1

     Best choices are:
        CANCEL had a formant-lpc distance factor of 1.0000
    correct = 1.000     cum_correct = 0.924
 Total wrong fusion digits are 2.50 from a total of 33.00 digits
```

FEATURE-FUSION: DELTA_(#14)

```
 lpc7_wr2_14.trn is:  22 47
   Should be:  14
    Best choices are:
       SOUTH had a lpc distance of 204.75 and a distance factor of 1.0000
       ALPHA had a lpc distance of 227.50 and a distance factor of 1.1111
       DELTA had a lpc distance of 268.25 and a distance factor of 1.3101
       THOUSAND had a lpc distance of 316.25 and a distance factor of 1.5446
    Best choices are:
       FUEL had a lpc distance of 127.50 and a distance factor of 1.0000
       TAIL had a lpc distance of 150.50 and a distance factor of 1.1804
       TWO had a lpc distance of 189.50 and a distance factor of 1.4863
       CHANNEL had a lpc distance of 223.00 and a distance factor of 1.7490
   correct = -0.500    cum_correct = 0.786
Total wrong lpc digits are 3.00 from a total of 14.00 digits


fmc7_wr2_14.trn is 136 vectors long
 fmc7_wr2_14.trn is:  14
   Should be:  14
    Best choices are:
       DELTA had a formant distance of 17128.95 and a distance factor of 1.0000
       FAULT had a formant distance of 24352.22 and a distance factor of 1.4217
       SOUTH had a formant distance of 28102.55 and a distance factor of 1.6406
   correct = 1.000    cum_correct = 0.821
Total wrong formant digits are 2.50 from a total of 14.00 digits


The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
The computed utterance string is:  14
    Should be:  14
The # of words less than 1.5 are 3

    Best choices are:
       DELTA had a formant-lpc distance factor of 4.7317
       SOUTH had a formant-lpc distance factor of 5.1986
       ALPHA had a formant-lpc distance factor of 7.4578
   correct = 1.000    cum_correct = 0.929
Total wrong fusion digits are 1.00 from a total of 14.00 digits
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  14
     Should be:  14
The # of words less than 2.0 are 6

     Best choices are:
        DELTA had a formant-lpc distance factor of 4.7317
        SOUTH had a formant-lpc distance factor of 5.1986
        ALPHA had a formant-lpc distance factor of 7.4578
        THOUSAND had a formant-lpc distance factor of 14.2788
        FOUR had a formant-lpc distance factor of 14.8429
        FIVE had a formant-lpc distance factor of 16.0068
    correct = 1.000     cum_correct = 0.929
 Total wrong fusion digits are 1.00 from a total of 14.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  14
     Should be:  14
The # of words less than 2.5 are 8

     Best choices are:
        DELTA had a formant-lpc distance factor of 4.7317
        SOUTH had a formant-lpc distance factor of 5.1986
        ALPHA had a formant-lpc distance factor of 7.4578
        MARK had a formant-lpc distance factor of 7.5601
        THOUSAND had a formant-lpc distance factor of 14.2788
        FOUR had a formant-lpc distance factor of 14.8429
        FIVE had a formant-lpc distance factor of 16.0068
        SPACE had a formant-lpc distance factor of 51.7200
    correct = 1.000     cum_correct = 0.929
 Total wrong fusion digits are 1.00 from a total of 14.00 digits
```

FEATURE-FUSION: THREE_(#3)

```
 lpc7_wr3_3.trn is:  52
   Should be:  3
    Best choices are:
       CHARLIE had a lpc distance of 303.50 and a distance factor of 1.0000
       STRAFE had a lpc distance of 326.75 and a distance factor of 1.0766
       BEARING had a lpc distance of 382.25 and a distance factor of 1.2595
       NORTH had a lpc distance of 395.00 and a distance factor of 1.3015
       THREE had a lpc distance of 425.25 and a distance factor of 1.4012
   correct = 0.000    cum_correct = 0.912
Total wrong lpc digits are 6.50 from a total of 74.00 digits


fmc7_wr3_3.trn is 180 vectors long
 fmc7_wr3_3.trn is:  34
  Should be:  3
   Best choices are:
    DEGREES had a formant distance of 31316.80 and a distance factor of 1.0000
    EAST had a formant distance of 31753.25 and a distance factor of 1.0139
    THREE had a formant distance of 35179.80 and a distance factor of 1.1234
   correct = 0.000    cum_correct = 0.912
 Total wrong formant digits are 6.50 from a total of 74.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
 The computed utterance string is:  3
     Should be:  3
The # of words less than 1.5 are 6

     Best choices are:
        THREE had a formant-lpc distance factor of 1.5740
        STRAFE had a formant-lpc distance factor of 1.9274
        NORTH had a formant-lpc distance factor of 2.0503
        BEARING had a formant-lpc distance factor of 2.2064
        CHARLIE had a formant-lpc distance factor of 2.5236
        ARM had a formant-lpc distance factor of 3.7863
    correct = 1.000    cum_correct = 0.946
    Total wrong fusion words are 4.00 from a total of 74.00 words
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  3
     Should be:  3
The # of words less than 2.0 are 25

     Best choices are:
        THREE had a formant-lpc distance factor of 1.5740
        EIGHT had a formant-lpc distance factor of 1.7521
        STRAFE had a formant-lpc distance factor of 1.9274
        NORTH had a formant-lpc distance factor of 2.0503
        BEARING had a formant-lpc distance factor of 2.2064
        CHARLIE had a formant-lpc distance factor of 2.5236
        MARK had a formant-lpc distance factor of 2.7673
        HEADING had a formant-lpc distance factor of 2.8473
        FREQUENCY had a formant-lpc distance factor of 2.8985
        NINE had a formant-lpc distance factor of 3.0892
    correct = 1.000     cum_correct = 0.946
    Total wrong fusion words are 4.00 from a total of 74.00 words


 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  3
     Should be:  3
The # of words less than 2.5 are 39

     Best choices are:
        THREE had a formant-lpc distance factor of 1.5740
        EIGHT had a formant-lpc distance factor of 1.7521
        STRAFE had a formant-lpc distance factor of 1.9274
        NORTH had a formant-lpc distance factor of 2.0503
        DEGREES had a formant-lpc distance factor of 2.0840
        BEARING had a formant-lpc distance factor of 2.2064
        CHARLIE had a formant-lpc distance factor of 2.5236
        MARK had a formant-lpc distance factor of 2.7673
        HEADING had a formant-lpc distance factor of 2.8473
        FREQUENCY had a formant-lpc distance factor of 2.8985
    correct = 1.000     cum_correct = 0.932
    Total wrong fusion words are 5.00 from a total of 74.00 words
```

```
                    FEATURE-FUSION: THREAT_(#40)

  lpc7_wr4_40.trn is:  46
    Should be:  40
     Best choices are:
        SEARCH had a lpc distance of 236.25 and a distance factor of 1.0000
        THREAT had a lpc distance of 264.00 and a distance factor of 1.1175
        KNOTS had a lpc distance of 304.50 and a distance factor of 1.2889
    correct = 0.000     cum_correct = 0.900
 Total wrong lpc digits are 18.00 from a total of 180.00 digits


fmc7_wr4_40.trn is 72 vectors long
 fmc7_wr4_40.trn is:  50
  Should be:  40
    Best choices are:
   WEST had a formant distance of 9493.40 and a distance factor of 1.0000
   THREAT had a formant distance of 10616.20 and a distance factor of 1.1183
   WEAPON had a formant distance of 11575.40 and a distance factor of 1.2193
    correct = 0.000     cum_correct = 0.875
 Total wrong formant digits are 22.50 from a total of 180.00 digits


 The LPC-F1/F2 fusion results for LPC distances less than 1.50 are:
 The computed utterance string is:  40
     Should be:  40
The # of words less than 1.5 are 8

     Best choices are:
        THREAT had a formant-lpc distance factor of 1.2496
        KNOTS had a formant-lpc distance factor of 1.9531
        FAULT had a formant-lpc distance factor of 2.8167
        SOUTH had a formant-lpc distance factor of 2.8356
        CANCEL had a formant-lpc distance factor of 3.7427
        TARGET had a formant-lpc distance factor of 4.4386
        STRAFE had a formant-lpc distance factor of 4.9690
        SEARCH had a formant-lpc distance factor of 7.1753
    correct = 1.000     cum_correct = 0.947
    Total wrong fusion words are 9.50 from a total of 180.00 words
```

```
 The LPC-F1/F2 fusion results for LPC distances less than 2.00 are:
 The computed utterance string is:  40
     Should be:  40
The # of words less than 2.0 are 22

     Best choices are:
        THREAT had a formant-lpc distance factor of 1.2496
        WEST had a formant-lpc distance factor of 1.6698
        KNOTS had a formant-lpc distance factor of 1.9531
        SELECT had a formant-lpc distance factor of 2.2832
        SIX had a formant-lpc distance factor of 2.5909
        FAULT had a formant-lpc distance factor of 2.8167
        SOUTH had a forma t-lpc distance factor of 2.8356
        MARK had a formant-lpc distance factor of 3.4952
        CANCEL had a formant-lpc distance factor of 3.7427
        TARGET had a formant-lpc distance factor of 4.4386
    correct = 1.000    cum_correct = 0.939
    Total wrong fusion words are 11.00 from a total of 180.00 words

 The LPC-F1/F2 fusion results for LPC distances less than 2.50 are:
 The computed utterance string is:  40
     Should be:  40
The # of words less than 2.5 are 46

     Best choices are:
        THREAT had a formant-lpc distance factor of 1.2496
        WEST had a formant-lpc distance factor of 1.6698
        KNOTS had a formant-lpc distance factor of 1.9531
        SELECT had a formant-lpc distance factor of 2.2832
        SIX had a formant-lpc distance factor of 2.5909
        FAULT had a formant-lpc distance factor of 2.8167
        SOUTH had a formant-lpc distance factor of 2.8356
        WEAPON had a formant-lpc distance factor of 2.8760
        MARK had a formant-lpc distance factor of 3.4952
        CANCEL had a formant-lpc distance factor of 3.7427
    correct = 1.000    cum_correct = 0.928
    Total wrong fusion words are 13.00 from a total of 180.00 words
```

Conclusion

This chapter presented all the tests used to ascertain the performance of the of the basic speech recognition system in Phase I and the improved system developed and tested in Phase II. The feature-fusion section provided excellent results for isolated speech. Formant recognition accuracies of 20% precluded the use of feature-fusion with connected speech.

The last chapter will summarize the results of both Phase I and Phase II testing, will discuss the weak areas in the system, and will suggest ways to improve overall accuracy.

VI. Conclusions and Recomendations

Introduction

This chapter concludes this thesis effort by summarizing the design and performance characteristics of the speech recognition system. Both strengths and weaknesses of the system are discussed. Suggested areas for further research are delineated.

Conclusion

This thesis effort was computationally intensive. An extensive body of knowledge was obtained on the response of a Kohonen neural network to speech input. This knowledge was used in the design of the speech recognition system which included a feature-fusion and rule-based section. The system vocabulary was expanded from 10 to 70 words; the recognition system now supports the F-16 vocabulary.

The feature-fusion section was able to combine the essential attributes present within each feature. The average isolated-word recognition accuracy was 97.2% on eleven Kohonen networks trained with LPC and $\mathcal{F}1, \mathcal{F}2$ formants when feature-fusion was utilized. The average recognition accuracy using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants was 97.4% on these same eleven Kohonen neural networks. The best individual recognition accuracy was 98.3% using $\mathcal{F}1, \mathcal{F}2$ formants, and 98.6% using $\mathcal{F}1, \mathcal{F}2, \mathcal{F}3$ formants. Poor formant recognition accuracy precluded use of the feature-fusion routine with connected speech. The average LPC connected-word recognition accuracy was 70.8% on eleven networks. The best LPC recognition accuracy obtained was 83.7%. A discussion of the recognition system follows.

Preprocessor The method used to process the formants allowed for excellent feature-fusion recognition accuracy on isolated speech. The effects of coarticula-

tion in connected speech corrupted the formant data representations. The result produced formant recognition accuracies around 20%. The basic method used to extract formant information from the SPIRE formant files needs improvement. The energy gate usually was able to reliably signal the regions in the signal where the formant routine had difficulty keeping track of vocal track harmonics. Sections of the signal that contained valid formant information were sometimes not included because the energy level was below -75 dB. All valid formant regions need to be included.

Nine of the 350 words in the system were replaced in the last part of Phase II testing using the range of values in the raw formant data file. Any improvements to a system should be considered valid only if they are applicable operationally. Evaluating raw formant data files could be easily automated.

Kohonen Neural Network The extensive amount of research on the Kohonen network in this thesis confirmed the network's special ability to classify human sounds. The network achieved a 97% recognition rate when trained on only half of the 70 word F-16 vocabulary. This finding supports the idea that only a subset of a recognition vocabulary will be required to train an operational system.

A Kohonen-Dynamic Programming three-dimensional surface was generated for two of the five gain reduction methods tested in this thesis effort. Analysis of the surface shows that a Kohonen neural network segments its surface into discrete sound regions quickly. Test results from Phase II did not show any significant improvement in recognition accuracy on Kohonen networks trained from 4,000 iterations to 90,000 iterations. The flat recognition response is especially true after fusion of speech features. In general, recognition accuracy without fusion tended to decrease as iteration cycle times approach 100,000. Local maximum recognition rates are possible as shown by the rough Kohonen-Dynamic Programming surface.

The Exponential gain reduction method proved superior to the other gain

reduction methods tested in this thesis. The Kohonen-Dynamic Programming three-dimensional surface produced with Exponential gain reduction is much higher overall (ie. higher recognition accuracy) than the Piecewise-Linear gain reduction method. Exponential gain reduction also segmented the sound groups more quickly on the Kohonen surface giving higher recognition accuracies with shorter training times.

The use of conscience allowed a Kohonen network to train faster as observed on the Kohonen-Dynamic Programming surface graphs. Tests showed that conscience does affect the recognition accuracy at least up to a conscience factor value of 20. The optimum conscience factor range observed was from five to seven for speech input when Exponential gain reduction was employed.

Dynamic Programming Sound types are grouped into distinct areas on the Kohonen surface providing two types of information to a word classifier. The network provides both relative and absolute coordinate information on word sounds. Dynamic programming operates by comparing the relative relationships between words in the Kohonen surface. Dynamic programming does not use the absolute coordinate information. Even without this information, dynamic programming provided excellent recognition accuracies with the system developed in this thesis.

Feature-Fusion Feature-fusion performed better than expected. Isolated-word recognition accuracies were above 90% on all tests. Phase II, Set #2 had average recognition accuracies above 95%. As noted earlier, feature-fusion was not usable on connected speech because of low average formant input accuracies.

All fusion thresholds gave similar results for isolated speech. Overall, a fusion threshold of 2.5 is recommended for isolated speech. The optimum fusion threshold for connected speech cannot be determined until the formant preprocessing method is improved. A lower value than 2.5 used for isolated speech will probably be required due to the effects of coarticulation.

Feature-fusion of three formants instead of just the first two formants did not significantly improve the recognition accuracy. Processing only the first two formants appears adequate for speech recognition and is recommended in future research.

Rule-Based System The rule-based section, which supports the feature-fusion routine, was not fully tested because of the poor formant accuracy on connected speech. The part of the rule-based section that supports isolated speech performed well as noted by the test results.

Recomendations

The following items are suggested areas for further research.

- Evaluate the method used to process raw formants. Even a modest improvement could potentially increase recognition performance significantly. In valid formant regions, formant tracks do not make abrupt changes in frequency. A different formant processing scheme could consider this formant characteristic and process the data files directly without using other features.
- Take advantage of the absolute coordinate information available from a Kohonen network. The word sounds or phonemes may need to be mapped on the Kohonen surface. Caution is advised since there is not total agreement on the number of actual phonemes in English. Also, most phonemes cannot be recorded easily in pure form. The vowels are an exception. An attempt should be made to map the vowel sounds on the Kohonen surface.
- Add a third feature to the recognition system. Data processed by a Fast Hartley Transform or Cepstral analysis may provide information to the fusion process that is not present in other features. Additional features, however, require additional processing.

- Improve the rule-based system. The rule-based system was designed to match unequal word segments. The system represents a shell for an expanded rule-based system based on phonetic, vocabulary, and syntax rules within a language.

Final Remarks

An attempt was made to expand the horizon in speech research. Usually speech systems are one-dimensional in which one feature or representation of speech is processed using a specific design methodology. The feature and design methodology have undergone significant revision through the years in an effort to improve performance.

Clearly, the human perception system is multi-dimensional; humans process more information than just the sound of a word during recognition. Context, inflection, facial expression, and rules of the language are a few of the sensory inputs used by the human perception system. Any speech recognition system that mimics the human perception system will need to be multi-dimensional. This thesis effort represents a step in that direction.

Appendix A. *Computer Source Code*

This section includes source code listings for a few of the programs developed in this thesis effort.

```
/*****************************************************************************
*
*       DATE: 14 Sept 1989
*       VERSION: 3.2
*
*       NAME: Kohonen_net.c
*       DESCRIPTION: Using a Kohonen neural network, this program creates
*       a two-dimensional representation of a multi-dimensional problem space.
*       ALGORITHM: Implementation of Kohonen neural network algorithm as
*       illustrated and described in IEEE magazine, Apr 87, by Dr. Lippman.
*
*       FILES READ: None
*       FILES WRITTEN: Net_file
*       MODULES CALLED: Kohonen_data.c
*
*       AUTHORS: Wayne F. Recla
*                Gary Barmore
*       HISTORY: The basic program was written by Barmore and has undergone
*          extensive modification and revision.
*
*****************************************************************************/

# include math
# include stdio
# include time

        int     conscience[20][20] ;            /* records # times closest */
        int     nodes ;                         /* number of nodes */
        long    its ;
        long    nodes_elim ;

        float   map[20][20][16] ;               /*  output nodes        */
        double  input[16] ;                     /*  input nodes         */
        double  gain ;
        double  mcount ;
        double  percent ;

        int     gain_type, conscience_type;
        int     filter_prompt = 0;
        double  low_iteration_value = 1;
        double  high_iteration_value = 1;
        double  surface_constant, time_dep_range, consc;

        int     closest[2] ;                    /*  closest node        */
        int     neigh[2] ;                      /*  neighbor            */
        int     nrangex, nrangey ;              /*  neighbor range      */
        int     nfactorx, nfactory ;            /*  neighbor factor     */
        long    count ;                         /*  # of iterations     */
        int     status_iteration ;              /*  # between plots     */
        int     seed ;
        int     maxneighx, maxneighy ;          /*  Starting area       */
        int     minneighx, minneighy ;          /*  Final area          */
        int     xsize, ysize ;                  /*  Size of array       */
        int     number_inputs ;

        char    training_file[30], net_file[30] ;
        char    net_name[15] ;

        struct curve {
                int             type ;
                double          maxgain ;
                double          mingain ;
                double          midgain ;
                int             midtime ;
                } gcurve ;
```



```
        struct flg {
                int             rnd_in ;
                } flag ;

/*****************************************************************************
        ROUTINE USED TO SET THE NETWORK WEIGHTS TO INITIAL RANDOM VALUES    **/
init (map)
        float   map[20][20][16] ;
{
        int     r, c, i ;
        float   max_rand = pow(2.0, 31.0) - 1.0 ;

        nodes_elim = 0 ;
        nodes = ysize * xsize ;
        for (r = 0 ; r < ysize ; r++)
          {
           for (c = 0 ; c < xsize ; c++)
             {
                conscience[r][c] = 0 ;
                for (i = 0 ; i < number_inputs ; i++)
                  {
                    map[r][c][i] = rand () / max_rand / 10.0 - .05 ;
                  }
             }
          }
}
/*****************************************************************************
        ROUTINE USED TO DETERMINE THE WINNING NODE DURING TRAINING          **/

mindist1 (map, inp, close)
        double          inp[16] ;
        int             close[2] ;
        float           map[20][20][16] ;
{
        int             r, c, i ;
        double          dot_product ;
        double          maximum = 0.0 ;

        for (r = 0 ; r < ysize ; r++)
          {
           for (c = 0 ; c < xsize ; c++)
             {
              if (conscience[r][c] < consc * its / nodes)
               {
                dot_product = 0.0;
                for (i = 0 ; i < number_inputs ; i++)
                  dot_product += inp[i] * map[r][c][i];
                if (dot_product > maximum)
                 {
                  maximum = dot_product;
                  close[0] = c;
                  close[1] = r;
                 } /** end if **/
               } /** end if **/
              else
                  nodes_elim++ ;
             } /** end for x **/
          } /** end for y **/
    conscience[close[1]][close[0]] += 1;
}

/*****************************************************************************
       ROUTINE USED TO OBTAIN INITIAL INFORMATION FROM THE USER.            **/
userinp ()
{
```

```
        int                     line ;
        int                     c ;
        struct tm               *localtime(), *time ;
        int                     *bintim;
        int             correct_response = 0;
 do
   {
    printf ("\n\n  KONONEN_NET (Net training with conscience prompt!) ... \n\n")
;

       /** Get network size **/
    printf("\n Enter size 'm n' (for an m x n) of array = ?  [int int]  ") ;
    scanf("%d %d", &ysize, &xsize) ;
    if (ysize < 2)
        ysize = 2 ;
    else if (ysize > 20)
        ysize = 20 ;
    if (xsize < 2)
        xsize = 2 ;
    else if (xsize > 20)
        xsize = 20 ;

       /** Get name of training file **/
    printf("\n Enter name of training file [.trn assumed]:  ") ;
    scanf ("%s", net_name) ;
    sprintf (training_file, "%s.trn", net_name) ;
    printf("     Training file is:  %s\n", training_file) ;

       /** Enter number of input nodes **/
    do
    {
     printf ("\n Enter the number of data inputs: [int] ");
     scanf ("%d", &number_inputs);
     if ((number_inputs > 0) && (number_inputs <= 16))
      correct_response = 1;
    }
    while (! correct_response);

       /** Enter name of net file **/
    printf("\n Enter name of net file to create [.net appended]:  ") ;
    scanf ("%s", net_name) ;
    sprintf (net_file, "%s.net", net_name) ;
    printf("     Net file to be created:  %s\n", net_file) ;

       /** Enter the number of training iterations **/
    printf ("\n Number of iterations = ?  [int]  ") ;
    scanf ("%ld", &count) ;
    if (count < 0)
        count = 0;
    if (count > 180000)
        count = 1000;
    mcount = (double) count ;

       /** Enter the iteration time between status messages **/
    printf ("\n Number of iterations between status messages = ?  [int]  ") ;
    scanf ("%d", &status_iteration) ;
    if (status_iteration < 1)
        status_iteration = 0;
    if (status_iteration > count)
        status_iteration = count;

    ingain () ;

    correct_response = 0;

       /** Conscience information **/
```



```
    conscience_type = 1;
    printf ("\n Enter a conscience value. ");
    scanf ("%f", &consc);

    if ((gcurve.type == 0) || (gcurve.type == 1))
    {
     printf ("\n\n #### The following two items are used only for Linear Type #1
 ");
     printf ("and \n   Piecewise Linear training. ####\n");
     printf ("\n Starting size of neighborhoods 'yn xn' = ?  [int int]  ") ;
     scanf ("%d %d", &maxneighy, &maxneighx) ;
     if (maxneighx < 2 || maxneighx > xsize - 1)
         maxneighx = 2 ;
     if (maxneighy < 2 || maxneighy > ysize - 1)
         maxneighy = 2 ;

     printf ("\n Final size of neighborhoods 'yn xn' = ?  [int int]  ") ;
     scanf ("%d %d", &minneighy, &minneighx) ;
     if (minneighx < 1 || minneighx > maxneighx)
         minneighx = 1 ;
     if (minneighy < 1 || minneighy > maxneighy)
         minneighy = 1 ;
    } /** end if **/

    printf
      ("\n\n Initial seed for random # generator (0 SELECTS TIME)  = ? [int]  ")
;
    scanf ("%d", &seed);
/**  Structure written by Barmore. Not needed and doesn't work.
    if (seed == 0) {
         time = localtime (bintim) ;
         time.tm_sec %= 60 ;
         time.tm_min %= 60 ;
         seed = time.tm_sec * time.tm_min ;
         }
**/
    srand (seed) ;

    printf("\n\n   Ready to begin?  (y/n)  ") ;
    while ((c = getc (stdin)) == ' ' || c == '\n' || c == '\t')
        ;
  } while (c != 'y') ;
} /** end function **/

/*****************************************************************************
        HAVE USER INPUT THE DESIRED GAIN REDUCTION METHOD                    **/
ingain ()
{
        int       line ;
        int       correct_response = 0;
do
{
   printf ("\n Choose the desired gain reduction method: \n");
   printf ("\n   0) Linear Type #1");
   printf ("\n   1) Piecewise Linear");
   printf ("\n   2) Linear Type #2");
   printf ("\n   3) Exponental");
   printf ("\n   4) Central-Adaption");
   printf ("\n   5) Exponential-Central-Adaption \n");

   scanf ("%d", &gcurve.type);
   gain_type = gcurve.type;
   if ((gcurve.type >= 0) && (gcurve.type <= 5))
     correct_response = 1;
 }
 while (! correct_response);
```

```
/*** Get more information on Linear, Piecewise Linear and Central-Adaption
reduction methods. ***/

switch (gcurve.type)
{
  case 0:  /*** Linear Type #1 ***/

      printf ("Maximum gain = ?  [float]  ") ;
      scanf ("%E", &gcurve.maxgain) ;
      if (gcurve.maxgain >= 1.0 || gcurve.maxgain <= 0.0)
          gcurve.maxgain = .99 ;

      printf ("Minimum gain = ?  [float]  ") ;
      scanf ("%E", &gcurve.mingain) ;
      if (gcurve.mingain <= 0.0 || gcurve.mingain >= 1.0)
          gcurve.mingain = 0.0 ;

      break;

  case 1:  /*** Piecewise Linear ***/

      printf ("First segment starting gain = ?  [float]  ") ;
      scanf ("%E", &gcurve.maxgain) ;
      if (gcurve.maxgain >= 1.0 || gcurve.maxgain <= 0.0)
          gcurve.maxgain = .99 ;

      printf ("Second segment starting gain = ?  [float]  ") ;
      scanf ("%E", &gcurve.midgain) ;
      if (gcurve.midgain <= 0.0 || gcurve.midgain >= 1.0)
          gcurve.midgain = 0.0 ;

      printf ("Second segment starting iteration = ?  [float]  ") ;
      scanf ("%d", &gcurve.midtime) ;
      if (gcurve.midtime <= 0 || gcurve.midtime > count)
          gcurve.midtime = count / 2 ;

          gcurve.mingain = 0.0 ;
      break;

 case 4:  /** Central-Adaption **/

      printf (" Enter a low iteration factor: [float] ");
      scanf ("%f", &low_iteration_value);
      printf (" Enter a high iteration factor: [float] ");
      scanf ("%f", &high_iteration_value);
      break;

 case 5:  /** Exponential-Central-Adaption **/

      printf (" Enter a low iteration factor: [float] ");
      scanf ("%f", &low_iteration_value);
      printf (" Enter a high iteration factor: [float] ");
      scanf ("%f", &high_iteration_value);
      break;

} /*** end switch ***/
} /** end function **/

/*************************************************************************
              DETERMINE THE CURRENT ITERATION GAIN VALUE                 **/
get_gain(i)

long    i;
{
```



```
switch (gcurve.type)
{
   case 0:  /*** Linear Type #1 ***/
      gain = (percent * (gcurve.maxgain - gcurve.mingain)) +
             gcurve.mingain;
      break;

   case 1:  /*** Piecewise Linear ***/
      if (i < gcurve.midtime)
        gain = gcurve.maxgain * (1.0 - (double) i / gcurve.midtime);
      else
        gain = gcurve.midgain * (1.0 - (double) i / mcount);
      break;

   case 2:  /*** Linear Type #2 ***/
      gain = 0.1 * (1.0 - (double) i / mcount);
      break;

   case 3:  /*** Exponential ***/
      gain = 0.1 * (1.0 - (double) i / mcount);
      break;

   case 4:  /*** Central-Adaption ***/
      gain = 0.2 * (1.0 - (double) i / mcount);
      break;

   case 5:  /*** Exponential-Central-Adaption ***/
      gain = 0.2 * (1.0 - (double) i /mcount);
      break;

}  /** end switch **/
}  /*** end function **/

/*************************************************************************
        SAVE NETWORK DATA AFTER TRAINING FOR PORTING THE DTW ROUTINE     **/
save_net ()
{
        int                     r, c, i ;
        FILE                    *fnet ;

        fnet = fopen(net_file,"w");
        fprintf (fnet,"%d %d %d", ysize, xsize, number_inputs);
        for (r = 0 ; r < ysize ; r++)
          {
           for (c = 0 ; c < xsize ; c++)
             {
              for (i = 0 ; i < number_inputs ; i++)
                {
                 fprintf (fnet," %f", map[r][c][i]);
                }
             }
          }
        fclose (fnet);
} /** end function **/

/*************************************************************************
        FUNCTION USED TO DETERMINE NODE UTILIZATION DURING CONSCIENCE TESTING.
        GOOD INDICATOR WHEN TRAINING NETS FOR LESS THAN 10K ITERATIONS.  **/
node_utilization()
{
   int     r, c;
   int     zero_count = 0;
   float   node_utilization_value;

   for (r = 0; r < ysize; r++)
```

```
    {
     for (c = 0; c < xsize; c++)
      {
        if (conscience[r][c] == 0)
          zero_count += 1;
      } /** end for **/
    } /** end for **/

    node_utilization_value = (float) ((xsize * ysize) - zero_count) /
    (xsize * ysize);
   printf ("\n Net node utilization is: %f",node_utilization_value);
}

/*****************************************************************************
                              MAIN PROGRAM                                  **/
main()
{
        long    i ;
        char    s1[10] ;
        int     k ;
        int     ws_id = 1 ;
        int     clear_flag = 1;
        FILE    *tf ;
        extern unsigned _stklen ;

        _stklen = 8192 ;

        userinp () ;                            /*  Get input values          */

        surface_constant = sqrt ((double) (ysize * ysize) + (xsize * xsize));

        nfactorx = maxneighx - minneighx + 1 ;
        nfactory = maxneighy - minneighy + 1 ;
        init (map) ;                            /*  Initialize weights        */

        read_trn_file () ;

        /*** START ITERATIVE PROCESS (NET TRAINING) ***/

        for (i = 1 ; i <= count ; i++)
        {
          its = i ;  /** Change iteration number i to a long integer. **/
          if (i % status_iteration == 0)
           {
        /** PRINT OUT STATUS MESSAAGES **/
            if (gcurve.type == 0 || gcurve.type == 1)
                /*** Linear Type #1 and Piecewise Linear ***/
             {
                if (gcurve.type == 0)  /** Linear Type #1 **/
           printf (" KOHONEN NET TRAINING using LINEAR TYPE #1 reduction. \n");
                else  /** Piecewise linear **/
           printf (" KOHONEN NET TRAINING using PIECEWISE LINEAR reduction. \n");

                printf (" gain = %f, yrange = %d, ", gain, nrangey) ;
                printf ("xrange = %d, iteration # %d", nrangex,i) ;
                printf (" (of %ld)\n", count);
             }  /** end linear and piecewise linear section **/

            else if (gcurve.type == 4)  /*** Central-Adaption ***/
             {
  printf (" KOHONEN NET TRAINING using CENTRAL-ADAPTION reduction. \n");
                printf (" gain = %f (partial factor only)", gain );
                printf (" iteration # %d (of %ld)  \n", i, count);
                printf (" range factor = %f \n", time_dep_range);
             } /** end central-adaption section **/
```



```
            else if (gcurve.type == 5) /*** EXPONENTIAL-CENTRAL-ADAPTION ***/
             {
 printf(" KOHONEN NET TRAINING using EXPONENTIAL-CENTRAL-ADAPTION reduction.\n");

                printf (" gain = %f (partial factor only)", gain );
                printf (" iteration # %d (of %ld)  \n", i, count);
                printf (" range factor = %f \n", time_dep_range);
             } /** end exponential-central-adaption section **/

            else  /*** Linear and Exponential ***/
             {
               if (gcurve.type == 2)
          printf (" KOHONEN NET TRAINING using LINEAR TYPE #2 reduction. \n");
               else
         printf (" KOHONEN NET TRAINING using EXPONENTIAL reduction. \n");
               printf (" gain = %f ",gain);
               printf (" iteration # %d (of %ld) \n  ", i, count);
               printf (" range factor = %f \n", time_dep_range );
             }
            k = nodes_elim / (double) status_iteration;
            printf ("%d ave nodes eliminated!\n",k);
            nodes_elim = 0;
            }  /*** end if ***/

            percent = (mcount - i) / mcount ;

        /** DETERMINE NEIGHBORHOOD SIZE **/
            switch (gcurve.type)
             {
              case 0:  /*** Linear Type #1 ***/
                nrangex = minneighx + percent * nfactorx;
                nrangey = minneighy + percent * nfactory;

                neigh[0] = nrangex;
                neigh[1] = nrangey;
                break;

              case 1:  /*** Piecewise Linear ***/
                if (i < gcurve.midtime)
                 {
                  nrangex = minneighx + nfactorx *
                            ((double) (gcurve.midtime - i)) / gcurve.midtime;
                  nrangey = minneighy + nfactory *
                            ((double) (gcurve.midtime - i)) / gcurve.midtime;
                 }
                else
                 {
                  nrangex = minneighx;
                  nrangey = minneighy;
                 }
                neigh[0] = nrangex;
                neigh[1] = nrangex; /** Keep neighborhoods square **/
                break;

              case 2:  /*** Linear Type #2 ***/
                time_dep_range = surface_constant + (1.0 - surface_constant) *
                                 (double) i / mcount;
                break;

              case 3:  /*** Exponential ***/
                time_dep_range = pow (surface_constant, (double)(1.0 -
                                 (double) i / mcount));
                break;
              case 4:  /*** Central-Adaption Type #1 ***/
                time_dep_range = surface_constant;
                break;
```

```
            case 5:  /*** Exponential-Central-Adaption ***/
              time_dep_range = pow (surface_constant, (double)(1.0 -
                               (double) i / mcount));
              break;

            }  /*** end switch ***/

          get_gain(i);

          getin1 (); /** Zero series not filtered out **/

          mindist1 (map, input, closest);
          /** weightem (map) ; **/
          calculate_new_weights (map, i);
        } /*** end for loop ***/

      node_utilization ();
      save_net () ;
      printf ("\nNet file:  %s saved!\n", net_file) ;
} /*** end main ***/
```

```
/*****************************************************************************
*
*       DATE: 5 Sept 1989
*       VERSION: 2.1
*
*       NAME: Kohonen_data.c
*       DESCRIPTION: Provides data for Kohonen net processing.
*       ALGORITHM: None
*       FILES READ: Training_file
*       FILES WRITTEN: None
*       MODULES CALLED: None
*       CALLING MODULES: Kohonen_net.c
*
*       AUTHOR: Wayne F. Recla
*               Gary Barmore
*       HISTORY: The basic program was written by Barmore and has undergone
*                extensive modification and revision.
*
*****************************************************************************/

# include math
# include stdio
# include stat

        extern double   input[16] ;             /*  input nodes         */
        extern double   gain ;

        extern int      closest[2] ;            /*  closest node        */
        extern int      neigh[2] ;              /*  neighbor            */
        extern int      xsize, ysize ;          /*  Size of array       */
        extern int      number_inputs ;
        extern char     training_file[30] ;

        extern int      gain_type, status_iteration;
        extern double   time_dep_range, surface_constant, mcount;
        extern double   low_iteration_value, high_iteration_value;

        int             tr_length ;
        float           tr_data[350000] ;  /** CHANGE ARRAY SIZE **/
        int             tr_counter = 0 ;
        int             tr_vectors ;
        int             node_sound[225] ;
        int             num_words ;
        int             word_limits[25][2] ;

/*****************************************************************************
                ROUTINE TO READ IN DATA FOR KOHONEN PROCESSING              **/
read_trn_file ()
{
        FILE            *tf ;
        float           value = 1.0 ;
        unsigned        memory ;

        tf = fopen (training_file, "r") ;
        tr_length = 0 ;
        while (feof(tf) == 0) {
                fscanf (tf, "%f", &value) ;
                *(tr_data + tr_length) = value ;
                tr_length++ ;
                }
        fclose (tf) ;
        tr_length-- ;
 /** CHANGED HARDCODED 15 TO NUMBER_INPUTS  and  MADE THE ARGUEMENT
     TO FLOOR A EXPLICIT DOUBLE **/
        tr_vectors = floor ((double) tr_length / (float) number_inputs) ;
```



```
        tr_length = number_inputs * tr_vectors ;
) /** end function **/

/*****************************************************************************
        ROUTINE GETS THE NEXT SEQUENCE OF VECTORS FOR PROCESSING            **/
getin1 ()
{
        int     i;

        if (tr_counter == tr_length)
         tr_counter = 0;
        for (i = 0; i < number_inputs; i++)
         {
         input[i] = *(tr_data + tr_counter);
         tr_counter++;
         }
) /** end function **/

/*****************************************************************************
                ROUTINE TO CALCULATE NEW WEIGHT FUNCTION                    **/

calculate_new_weights (map, i)
   float    map[20][20][16];
   long     i;
{
   int      nright, nleft, nup, ndown, r , c, in ;
   double   map_distance, super_script, total_gain;

if (gain_type == 0 || gain_type == 1)
     /***  Linear Type #1 and Piecewise Linear  ***/
 {
    if (neigh[0] > 0 && neigh[1] > 0)
      {
       nright = closest[0] + neigh[0] - 1 ;
       if (nright >= xsize)
            nright = xsize - 1 ;
       nleft = closest[0] - neigh[0] + 1 ;
       if (nleft < 0)
            nleft = 0 ;
       nup = closest[1] - neigh[1] + 1 ;
       if (nup < 0)
            nup = 0 ;
       ndown = closest[1] + neigh[1] - 1 ;
       if (ndown >= ysize)
            ndown = ysize - 1 ;

      }
    else
      {
        nright = closest[0] ;
        nleft = closest[0] ;
        nup = closest[1] ;
        ndown = closest[1] ;
      } /** end if **/

    for (r = nup; r <= ndown ; r++)
     {
      for (c = nleft ; c <= nright ; c++)
       {
        for (in = 0 ; in < number_inputs ; in++)
             map[r][c][in] += gain * (input[in] - map[r][c][in]) ;
      }      /** end for **/
     }      /** end for **/
 }     /** end if (gain_type = 0 or 1 **/

else  /*** Linear Type #2, Exponential, and Central-Adaptive ***/
```

```
{
 for (r = 0; r < ysize; r++)
   {
   for (c = 0; c < xsize; c++)
     {
         map_distance = pow((double) (c - closest[0]), (double) 2.0) +
                        pow((double) (r - closest[1]), (double) 2.0);
         map_distance = sqrt(map_distance);
         if (map_distance <= time_dep_range)
          {
            if (gain_type == 4 || gain_type == 5)
             /** Central-Adaption  and Exponential-Central-Adaption **/
             {
              super_script = -1.0 * map_distance *
               ((double) 1.4 / (low_iteration_value * surface_constant) +
               (double) i / (high_iteration_value * mcount)
               * (5.6 - (double) 1.4 / surface_constant));

               total_gain = gain * exp(super_script);

               for (in = 0; in < number_inputs; in++)
                 map[r][c][in] += total_gain * (input[in] - map[r][c][in]);
             } /** end if (central-adation) **/

            else

             (  /** Linear Type #2 and Exponential **/

              for (in = 0; in < number_inputs; in++)
                  map[r][c][in] += gain * (input[in] - map[r][c][in]);
             )  /** end if (gcurve.type = 2, 3, 4 or 5 **/
           ) /** end if (map_distance < time_dep_range) **/

     )  /** end for **/
   )  /** end for **/
 )  /** end if **/
) /** end                on **/
```

```
/*************************************************************************
*
*       DATE: 27 Sept 1989
*       Version: 3.4
*
*       NAME: Dtw_f12.c
*       DESCRIPTION: Performs dynamic time warping on two features
*       of speech.  The features are LPC and F1/F2 formants.
*       ALGORITHM: Ney's Dynamic Time Warping Algorithm
*       PASSED VARIABLES: None
*       FILES READ: 1) Template Library File. File contains the 71
*                   LPC and formant files for the DTW library.
*                   2) Test Batch File. File contains a list of file
*                   names.  Each filename represents a file containing
*                   words to test the recognition system.
*                   3) LPC Kohonen Network file.
*
*       FILES WRITTEN: Log File.  File contains system recognition results.
*
*       AUTHOR: Wayne F. Reela
*               Gary Barmore
*       HISTORY: A few functions from the Barmore thesis are used in this
*                program including the DTW source code.  Most of these functions
*                underwent extensive modification.
*
*************************************************************************/

# include stdio
# include math

/*************************************************************************
       THE LIBRARY SIZE CAN BE EXPANDED BY CHANGING THE DEFINE
       STATEMENT BELOW AND UPDATING THE LIST OF LIBRARY WORDS.
       THE LIBRARY WORDS ARE POSITION SENSITIVE.                    **/

# define library_size 71

/*************************************************************************
                        VARIABLE LISTING                            **/
       float   lhorz, lvert, fhorz, fvert;
       float   map[20][20][16];
       int     xsize, ysize;
       int     number_inputs;;
       char    training_file[30];
       int     num_templates;

       float   total_lpc_digits;
       float   wrong_lpc_digits;
       float   total_for_digits;
       float   wrong_for_digits;
       float   total_lpc_for_digits0;
       float   wrong_lpc_for_digits0;
       float   total_lpc_for_digits1;
       float   wrong_lpc_for_digits1;
       float   total_lpc_for_digits2;
       float   wrong_lpc_for_digits2;
       float   total_lpc_for_digits3;
       float   wrong_lpc_for_digits3;

       int     tem_array[library_size], lutt_array[50], futt_array[50];
       int     lptr_array[50], fptr_array[50];
       int     t_words;
       int     num_lpc_utt, num_for_utt;
       float   percent_corr;
```



```
       char    f_training_file[30];
       char    batch_file[30];

       float   fdistance_array[50][library_size];
       float   ldistance_array[50][library_size];
       float   lpc_distance[3000][library_size];
       float   formant_distance[3000][library_size];
       float   formant_lpc_distance[50][library_size];
       int     word_lpc[3000][library_size];
       int     word_formant[3000][library_size];
       int     lword_array[50][library_size];
       int     fword_array[50][library_size];

       int     data_value;
       int     best_choices, word_number;
       int     correct_response, formant_map_included, f_flag;
       int     process_type, f_number_inputs;

/*************************************************************************
                        RECOGNITION SYSTEM LIBRARY WORDS            **/

       const char *library_words[library_size] = { "ZERO", "ONE", "TWO", "THREE"
,
                "FOUR", "FIVE", "SIX", "SEVEN", "EIGHT", "NINE", "SPACE",
       "AFT", "ARM", "CHAFF", "DELTA", "FLARES", "HUNDRED", "NEGATIVE",
       "AIR-TO-AIR", "BACKSPACE", "THOUSAND", "EAST", "FUEL", "MAP",
       "TAIL", "CLEAR", "PROFILE", "REPORT", "AFFIRMATIVE", "FREQUENCY",
       "WAYPOINT", "ADVISE", "BEARING", "CANCEL", "DEGREES", "FOXTROT",
       "LOCK-ON", "MISSILE", "RADAR", "STATION", "THREAT", "BRAVO", "ECHO",
       "ENTER", "MARK", "NOSE", "SEARCH", "SOUTH", "TARGET", "WEAPON",
       "WEST", "AIR-TO-SURFACE", "CHARLIE", "FAULT", "GUN", "KNOTS",
       "MILES", "NORTH", "RANGE", "SELECT", "FORWARD", "ALPHA", "CHANGE",
       "CHANNEL", "CONFIRM", "HEADING", "MINUS", "RHAW", "SMS", "STRAFE",
       "POINT");
/*************************************************************************/
main ()
{
/*************************************************************************
                        VARIABLE LISTING                            **/

       char    temp[30], temp_file[30], utt_file[30], ltemp[30];
       char    file_name[30], file_descr[80] ;
       char    sub[library_size][30] ;
       FILE    *flog, *fnet, *f_bat, *ft_map, *fu_map, *f_lib, *f_in;
       char    template[library_size][30] ;
       int     t_array[library_size][1000][2], u_array[1000][2];

       int     t_length[library_size];
       int     u_length, dist, r, c, i ;
       int     entries_std, entries_cat ;
       int     i_std, i_cat, k, kk;

       char    f_template[library_size][30], f_temp[30], for_utt_file[30];
       char    log_file[30];
       int     uf_array[3000], ufor_array[3000];
       int     tf_array[library_size][1000];
       int     f_array[library_size][1000];
       int     a, p, j, f1_factor, num_reduction_passes;
       int     eof_flag, f_length, max_length, min_length;
       int     f_threshold, formant_threshold;
       int     tf_length[library_size];
       int     uf_length, ufor_length, reduction_requested;

       float   f1, f2, f3, f4, threshold_factor;

/*************************************************************************
```

```
                        PRINT HEADER                                    **/

      printf ("\n\nDTW_F1F2:       ******** DYNAMIC TIME WARPING ********\n\n")
 ;

/*************************************************************************
          ENTER NAME OF TEMPLATE (LIBRARY) FILE                         **/
       printf ("  Enter name of template (library) file [less .hdr]:  ") ;
       scanf ("%s", temp) ;
       sprintf (temp_file, "%s.hdr", temp) ;
       f_lib = fopen (temp_file, "r") ;
       fscanf (f_lib, "%d", &num_templates);
       printf ("\n %d templates in library.\n\n", num_templates);

/*************************************************************************
               ENTER NAME OF TEST BAT FILE                              **/
       printf ("  \nEnter name of the test batch file [less .hdr]: \n ");
       scanf ("%s", temp) ;
       sprintf (batch_file, "%s.hdr", temp) ;

/*************************************************************************
                    ENTER TEMPLATE NAMES INTO PROGRAM                   **/
      for (i = 0 ; i < num_templates ; i++)
       {
        fscanf (f_lib, "%s",   template[i]);
        fscanf (f_lib, "%s", f_template[i]);
       }
      fclose (f_lib);

/*************************************************************************
                ENTER VERTICAL AND HORIZONTAL FACTORS                   **/
      printf ("  \n\nEnter LPC horizontal weight:  ") ;
      scanf ("%f", &lhorz) ;
      printf ("  \nEnter LPC vertical weight:    ") ;
      scanf ("%f", &lvert);

      printf ("  \n\nEnter FORMANT horizontal weight:  ") ;
      scanf ("%f", &fhorz) ;
      printf ("  \nEnter FORMANT vertical weight:    ") ;
      scanf ("%f", &fvert);

/*************************************************************************
          ENTER MINIMUM NUMBER OF CHOICES TO PRINT OUT                  **/
      printf ("  \n\nEnter minimum number of choices to print out:   ");
      scanf ("%d", &best_choices);

/*************************************************************************
                        FORMANT REDUCTION REQUEST                       **/
      printf ("  \n\nDo you want to reduce the formant data? 0]no or 1]yes   ")
;
      scanf ("%d", &reduction_requested);

/*************************************************************************
            ASK FOR ADDITIONAL INFORMATION IF FORMANT REDUCTION
            IS REQUESTED, ELSE SET NUM_REDUCTIONS TO ZERO.              **/

   if (reduction_requested)
    {
/*************************************************************************
                ENTER A FORMANT THRESHOLD FACTOR                        **/
      printf ("  \nEnter a formant threshold factor: [float]   ");
      scanf ("%f", &threshold_factor);

/*************************************************************************
                ENTER A F1 RANGE FACTOR                                 **/
      printf ("  \nEnter a F1 range factor: [int]   ");
```



```
      scanf ("%d", &f1_factor);

/*************************************************************************
                ENTER NUMBER OF FORMANT REDUCTION PASSES                **/
      printf ("  \nEnter number of formant reduction passes: [int]   ");
      scanf ("%d", &num_reduction_passes);

/*************************************************************************/
  }
  else /** REDUCTION NOT REQUESTED **/
  {
  num_reduction_passes = 0;
  }

/*************************************************************************
                    ENTER NAME OF LPC NET                               **/
        printf("  \n\nEnter name of primary net to use [less .net]:  ");
        scanf ("%s", temp);
        sprintf (training_file, "%s.net", temp);

/*************************************************************************
                    ENTER NAME OF LOG FILE                              **/
        printf("  \n\nEnter name of log output file. [less .dlog]:  ");
        scanf ("%s", ltemp);
        sprintf (log_file, "%s.dlog", ltemp);

/*************************************************************************
              OPEN OUTPUT LOG FILE AND PRINT START DATA                 **/
      flog = fopen (log_file, "w");

      printf ("\n Log file: %s created and opened.\n", log_file);
      printf ("\n LPC horizontal and vertical weights are %1.2f and %1.2f",
              lhorz, lvert);
     printf ("\n FORMANT horizontal and vertical weights are %1.2f and %1.2f\n"
,
              fhorz, fvert);
     printf ("\n The formant threshold is %1.2f", threshold_factor);
     printf ("\n The F1 range factor is %3d", f1_factor);
     printf ("\n The number of formant reduction passes is %2d\n",
             num_reduction_passes);


      fprintf (flog, "\nDynamic Time Warping using %s\n",training_file);
      fprintf (flog, "\n The output log file is %s \n", log_file);
      fprintf (flog," LPC Horizontal and vertical weights are %g and %g.\n",
              lhorz, lvert);
      fprintf (flog, " FORMANT Horizontal and vertical weights are %g and %g.\n
",
              fhorz, fvert);
      fprintf (flog, "  The formant threshold is %1.2f", threshold_factor);
      fprintf (flog, "\n  The F1 range factor is %3d", f1_factor);
      fprintf (flog, "\n  The number of formant reduction passes is %2d\n\n",
              num_reduction_passes);

/*************************************************************************
                        OPEN NET FILE AND IMPORT DATA                   **/

        fnet = fopen (training_file, "r") ;
      fscanf (fnet,"%d %d %d", &ysize, &xsize, &number_inputs);
        for (r = 0 ; r < ysize ; r++)
          {
           for (c = 0 ; c < xsize ; c++)
            {
             for (i = 0 ; i < number_inputs ; i++)
              {
               fscanf (fnet," %f", &map[r][c][i]) ;
```

```
            )
          )
        )
      fclose (fnet);

/*****************************************************************************
      REDUCE LIBRARY LPC VECTORS AND PRINT VECTOR LENGTH                  **/
      for (i = 0 ; i < num_templates ; i++)
       {
        get_vectors(template[i], &t_array[i][0][0], &t_length[i]);
        printf ("  %s is %d vectors long\n",template[i], t_length[i]);
      fprintf(flog,"  %s is %d vectors long\n", template[i], t_length[i]);
       }
      printf ("\n");
      fprintf(flog,"\n");

/*****************************************************************************
                        ENTER FORMANT LIBRARY FILE DATA                   **/
       for (i = 0; i < num_templates; i++)
       {
        printf ("\n Processing %s \n", f_template[i]);
        ft_map = fopen (f_template[i], "r");
        f_length = 0;
        max_length = 0;
        min_length = 1000;
        eof_flag = 0;

        while (eof_flag != 1)
        {
         fscanf (ft_map, "%f", &f1);

         if (feof(ft_map) != 0)
           /** if end of file reached, save length of file and
               range of file. **/
           {
            eof_flag = 1;
            tf_length[i] = f_length;
            if (f_length < min_length)
              min_length = f_length;
            if (f_length > max_length)
              max_length = f_length;
           }
         else
           {
            fscanf (ft_map, "%f", &f2);
            fscanf (ft_map, "%f", &f3);
            fscanf (ft_map, "%f", &f4);

            if (f1 != 0)
             /** only save nonzero values. **/
             {
              f_array[i][f_length] = f1;
              f_length++;
              f_array[i][f_length] = f2;
              f_length++;
             } /** end if **/
           } /** end if **/
        } /** end while **/
       fclose (ft_map);
      } /** end for i **/

      max_length = 0;
      min_length = 1000;
      for (i = 0; i < num_templates; i++)
        {
         if (tf_length[i] < min_length)
```



```
           min_length = tf_length[i];
         if (tf_length[i] > max_length)
           max_length = tf_length[i];
        }
      f_threshold = threshold_factor * (max_length - min_length);
      printf ("\n min is %d, max is %d, and threshold is %d\n",
              min_length, max_length, f_threshold);
      printf ("\n  with a F1 factor of %d\n", f1_factor);

      fprintf (flog, "\n min is %d, max is %d, and threshold is %d\n",
            min_length, max_length, f_threshold);
      fprintf (flog, "  with a F1 factor of %d\n", f1_factor);

/*****************************************************************************
                        REDUCE FORMANT LIBRARY VECTORS                    **/
      for (j = 0; j < num_reduction_passes; j++)
       {
        for (i = 0; i < num_templates; i++)
        {
         formant_threshold = (j + 1) * f_threshold;
         if (tf_length[i] > formant_threshold)
          {
          f_length = 0;
          for (a = 0; a < tf_length[i] / 4; a++)
           {
          p = a * 4;
          if ((abs(f_array[i][p] - f_array[i][p + 2]) <= f1_factor) &&
          (abs(f_array[i][p + 1] - f_array[i][p + 3]) <= 3.0 * f1_factor))
                {

                 tf_array[i][f_length] = f_array[i][p];
                 f_length++;
                 tf_array[i][f_length] = f_array[i][p + 1];
                 f_length++;
                }
               else
                {
                 tf_array[i][f_length] = f_array[i][p];
                 f_length++;
                 tf_array[i][f_length] = f_array[i][p + 1];
                 f_length++;

                 tf_array[i][f_length] = f_array[i][p + 2];
                 f_length++;
                 tf_array[i][f_length] = f_array[i][p + 3];
                 f_length++;
                } /** end if **/
           } /** end for a **/
          printf (" \n%s is %d vectors long with a start length of %d\n",
                  f_template[i], f_length, tf_length[i]);
      fprintf (flog, " %s is %d vector long with a start length of %d\n",
                  f_template[i], f_length, tf_length[i]);
          tf_length[i] = f_length;
          } /** end if **/
         else
          {
            for (k = 0; k < num_templates; k++)
             for (kk = 0; kk < tf_length[i]; kk++)
              tf_array[k][kk] = f_array[k][kk];

            printf (" \n%s is %d vectors long\n", f_template[i],
                  tf_length[i]);
            fprintf (flog, " %s is %d vectors long\n", f_template[i],
                  tf_length[i]);
          }
       } /** end for i **/
```

```
        ) /** end for j **/

        if (num_reduction_passes == 0)
         {
         for (i = 0; i < num_templates; i++)
          {
           for (j = 0; j < tf_length[i]; j++)
            tf_array[i][j] = f_array[i][j];
           printf (" \n%s is %d vectors long\n", f_template[i],
                   tf_length[i]);
           fprintf (flog, " %s is %d vectors long\n", f_template[i],
                   tf_length[i]);
          ) /** end for **/
         ) /** end if **/

/*************************************************************************
                OPEN BATCH.HDR FILE AND PROCESS                          **/
       f_bat = fopen (batch_file, "r");
       fscanf (f_bat, "%d", &entries_std);
       for (i_std = 0 ; i_std < entries_std ; i_std++)
         {
              fscanf (f_bat, "%s %s", file_name, file_descr);
              printf ("\x7");
              printf ("\n%s\n", file_descr);
              total_lpc_digits = 0.0;
              wrong_lpc_digits = 0.0;
              total_for_digits = 0.0;
              wrong_for_digits = 0.0;
              total_lpc_for_digits0 = 0.0;
              wrong_lpc_for_digits0 = 0.0;
              total_lpc_for_digits1 = 0.0;
              wrong_lpc_for_digits1 = 0.0;
              total_lpc_for_digits2 = 0.0;
              wrong_lpc_for_digits2 = 0.0;
              total_lpc_for_digits3 = 0.0;
              wrong_lpc_for_digits3 = 0.0;


              fprintf (flog, "\n%s\n", file_descr) ;
              f_in = fopen (file_name, "r") ;
              fscanf (f_in, "%d", &entries_cat) ;

/*************************************************************************
                    PROCESS TEST WORDS                                   **/
              for (i_cat = 0 ; i_cat < entries_cat ; i_cat++)
               {
                  fscanf (f_in, "%s %d", utt_file, &t_words) ;
                  fscanf (f_in, "%s ", for_utt_file);
                  for (i = 0 ; i < t_words ; i++)
                    fscanf (f_in, "%d", &tem_array[i]);
                  get_vectors(utt_file, u_array, &u_length);

                  printf ("  %-12s is:  ", utt_file) ;
                  fprintf (flog,"  %-12s is:  ", utt_file);
/*************************************************************************
               PERFORM DTW ON LPC PROCESSED WORD                         **/
                  cdtw (t_array,u_array,t_length,u_length,flog);

/*************************************************************************
                      ENTER TEST FILE FORMANT DATA                       **/
                  fu_map = fopen (for_utt_file, "r");
                  uf_length = 0;
                  max_length = 0;
                  min_length = 1000;
                  eof_flag = 0;
```



```
                  while (eof_flag != 1)
                   {
                    fscanf (fu_map, "%f", &f1);

                    if (feof(fu_map) != 0)
                      /** if end of file reached, save length of file and
                          range of file. **/
                     {
                      eof_flag = 1;
                      if (uf_length < min_length)
                        min_length = uf_length;
                      if (uf_length > max_length)
                        max_length = uf_length;
                     }
                    else
                     {
                      fscanf (fu_map, "%f", &f2);
                      fscanf (fu_map, "%f", &f3);
                      fscanf (fu_map, "%f", &f4);

                      if (f1 != 0)
                        /** only save nonzero values. **/
                       {
                        uf_array[uf_length] = f1;
                        uf_length++;
                        uf_array[uf_length] = f2;
                        uf_length++;
                       ) /** end if **/
                      ) /** end if **/
                     ) /** end while **/
                    fclose (fu_map);

/*************************************************************************
                          REDUCE FORMANT TEST VECTOR                     **/
                for (j = 0; j < num_reduction_passes; j++)
                {
                 formant_threshold = (j + 1) * f_threshold;
                 if (uf_length > formant_threshold)
                  {
                  f_length = 0;
                  for (a = 0; a < uf_length / 4; a++)
                   {
                   p = a * 4;
                   if ((abs(uf_array[p] - uf_array[p + 2]) <= f1_factor) &&
                    (abs(uf_array[p + 1] - uf_array[p + 3]) <= 3.0 * f1_factor))
                        {
                          ufor_array[f_length] = uf_array[p];
                          f_length++;
                          ufor_array[f_length] = uf_array[p + 1];
                          f_length++;
                        }
                       else
                        {
                          ufor_array[f_length] = uf_array[p];
                          f_length++;
                          ufor_array[f_length] = uf_array[p + 1];
                          f_length++;

                          ufor_array[f_length] = uf_array[p + 2];
                          f_length++;
                          ufor_array[f_length] = uf_array[p + 3];
                          f_length++;
                      ) /** end if **/
                    ) /** end for a **/
                    printf (" \n%s is %d vectors long from a length of %d",
                           for_utt_file, f_length, uf_length);
```

```
        fprintf (flog, " \n%s is %d vectors long from a length of %d",
                        for_utt_file, f_length, uf_length);
               ufor_length = f_length;
               uf_length = f_length;
              ) /** end if **/
              else
               {
                 ufor_length = uf_length;
                 for (kk = 0; kk < ufor_length; kk++)
                   ufor_array[kk] = uf_array[kk];

                 printf (" \n%s is %d vectors long\n", for_utt_file,
                         ufor_length);
        fprintf (flog, " \n%s is %d vectors long\n", for_utt_file,
                        ufor_length);

               }
             } /** end num reduction passes **/

            if (num_reduction_passes == 0)
             {
              for (j = 0; j < uf_length; j++)
               ufor_array[j] = uf_array[j];
              ufor_length = uf_length;
              printf (" \n%s is %d vectors long\n", for_utt_file,
                      ufor_length);
              fprintf (flog, " \n%s is %d vectors long\n", for_utt_file,
                      ufor_length);
             ) /** end if **/


            printf ("  %-12s is:  ", for_utt_file);
            fprintf (flog,"  %-12s is:  ", for_utt_file);
/*************************************************************************
                  PERFORM DTW ON FORMANT PROCESSED WORD                **/
              fdtw (tf_array,ufor_array,tf_length,ufor_length,flog);

/*************************************************************************
                  PERFORM FEATURE FUSION                               **/
              feature_fusion(flog);

           } /** end for i_cat **/
          } /** end for i_std **/
        fclose (f_bat);
        fclose (flog);


  } /** end main **/

/*************************************************************************
        THIS ROUTINE PERFORMS DYNAMIC TIME WARPING ON FORMANT DATA      **/

fdtw (t_array, u_array, t_length, u_length, flog)
        int     t_array[][1000], u_array[3000];
        int     t_length[library_size], u_length;
        FILE    *flog ;
{
        float   accum_dist[2][library_size][1000] ;
        int     utterance[20];
        int     back_ptr[2][library_size][1000] ;
        int     from_template[2000] ;
        int     from_frame[2000] ;
        int     kk;
        float   d1, d2, d3, dist ;
        float   min_dist ;
        register int j, k, ptr, b_ptr, i;
```



```
        int     best_words[library_size];
        float   best_distances[library_size], fzero_flag[20];
        float   cum_per_corr, distance_measure;
        int     found_match, finished, utt_count;
        int     new_best_choice_found, k1;

        ptr = 0 ;
        for (k = 0 ; k < num_templates ; k++)
          {
                for (j = 0 ; j < t_length[k] ; j++)
                  {
                        if (j == 0)
                        {
                          accum_dist[ptr][k][j] =
                           (dist = abs(u_array[0] - t_array[k][j]));
                        }
                        else
                        {
                          accum_dist[ptr][k][j] = (dist += fvert *
                           (abs(u_array[0] - t_array[k][j])));
                        }
                        back_ptr[ptr][k][j] = 0 ;
                  }
         }
        for (i = 1 ; i < u_length ; i++) {
                if (ptr == 0) {
                        ptr = 1 ;
                        b_ptr = 0 ;
                        }
                else {
                        ptr = 0 ;
                        b_ptr = 1 ;
                        }
                for (k = 0 ; k < num_templates ; k++) {
                    for (j = 0 ; j < t_length[k] ; j++) {
                        dist = abs(u_array[i] - t_array[k][j]);
                        if (j == 0) {
                                min_dist = 99999.0 ;
                                for (kk = 0 ; kk < num_templates ; kk++) {
                                    if (min_dist >
                                    accum_dist[b_ptr][kk][t_length[kk]-1])
                                        min_dist =
                                        accum_dist[b_ptr][kk][t_length[kk]-1];
                                        }
                                if (accum_dist[b_ptr][k][0] < min_dist) {
                                        accum_dist[ptr][k][0] = fhorz * dist +
                                          accum_dist[b_ptr][k][0] ;
                                        back_ptr[ptr][k][0] =
                                          back_ptr[b_ptr][k][0] ;
                                        }
                                else {
                                        accum_dist[ptr][k][0] =
                                          min_dist + dist ;
                                        back_ptr[ptr][k][0] = i-1 ;
                                        }
                                }
                        else {
                                d1 = accum_dist[b_ptr][k][j-1] + dist ;
                                d2 = accum_dist[ptr][k][j-1] + (fvert * dist) ;
                                d3 = accum_dist[b_ptr][k][j] + (fhorz * dist) ;
                                if (d2 <= d3 && d2 < d1) {
                                        accum_dist[ptr][k][j] = d2 ;
                                        back_ptr[ptr][k][j] =
                                                back_ptr[ptr][k][j-1] ;
                                        }
```

```
                        else if (d3 <= d2 && d3 < d1) {
                                accum_dist[ptr][k][j] = d3 ;
                                back_ptr[ptr][k][j] =
                                        back_ptr[b_ptr][k][j] ;
                                }
                        else {
                                accum_dist[ptr][k][j] = d1 ;
                                back_ptr[ptr][k][j] =
                                        back_ptr[b_ptr][k][j-1] ;
                                }
                        }
                    } /** next j **/
                } /** next k **/

            min_dist = 999999.0;
            for (k = 0 ; k < num_templates ; k++)
             {
             word_formant[i][k] = k;
             formant_distance[i][k] = accum_dist[ptr][k][t_length[k]-1];
             if (min_dist > accum_dist[ptr][k][t_length[k]-1])
              {
               min_dist = accum_dist[ptr][k][t_length[k]-1] ;
               kk = k ;
              }
             }
            from_template[i] = kk ;
            from_frame[i] = back_ptr[ptr][kk][t_length[kk]-1] ;
  } /** next i **/
/*************************************************************************/

      ptr = u_length - 1 ;
      i = -1;
      while ((ptr > 0) && (i < 99))
       {
        utterance[++i] = from_template[ptr];
        i--;
        fptr_array[++i] = ptr;
        fzero_flag[i] = 0;

        sort_best (word_formant, formant_distance, num_templates, ptr);

        for (k = 0; k < num_templates; k++)
         {
          fword_array[i][k] = word_formant[ptr][k];
          if (formant_distance[ptr][0] != 0)
           {
            fdistance_array[i][k] = formant_distance[ptr][k] /
                                    formant_distance[ptr][0];
           }
          else
           {
            fdistance_array[i][k] = formant_distance[ptr][k];
            fzero_flag[i] = 1;
           }
         } /** end for **/

        ptr = from_frame[ptr] ;
       } /** end while **/
       num_for_utt = i + 1;
       k = 0 ;
       for (j = i; j >= 0; j--)
        {
         printf ("%d ", utterance[j]) ;
         fprintf (flog, "%d ", utterance[j]) ;
        futt_array[k++] = utterance[j] ;
        }
```



```
       printf ("\n    Should be:  ") ;
       fprintf (flog, "\n    Should be:  ") ;
       for (j = 0 ; j < t_words ; j++)
        {
         printf ("%d ", tem_array[j]);
         fprintf (flog, "%d ", tem_array[j]) ;
        }

/*************************************************************************
                    PRINT OUT THE BEST CHOICES                          ****/

       for (j = 0; j < num_for_utt; j++)
         {
          printf ("\n    Best choices are: ");
          fprintf (flog, "\n     Best choices are: ");

          found_match = 0;
          finished = 0;
          k1 = -1;
          do
          {
           k1++;
           printf ("\n     %s had a formant distance of %5.2f and a ",
                  library_words[fword_array[j][k1]],
                   formant_distance[fptr_array[j]][k1]);
           printf ("distance factor of %3.4f", fdistance_array[j][k1]);

  fprintf (flog, "\n        %s had a formant distance of %5.2f and a ",
                   library_words[fword_array[j][k1]],
                   formant_distance[fptr_array[j]][k1]);
  fprintf (flog, "distance factor of %3.4f", fdistance_array[j][k1]);

           if (num_for_utt == t_words)
            {
             if ((fword_array[j][k1] == tem_array[j]) || k1 > 10)
               found_match = 1;
            }
           else
            {
             if (k1 >= best_choices)
               found_match = 1;
            }
             if ((k1 >= best_choices - 1) && (found_match))
              finished = 1;
          } /** end do-while **/
          while (! finished);
         } /** end for j **/


       grader (tem_array, futt_array, t_words, k, &distance_measure);

       total_for_digits += t_words;
       wrong_for_digits += distance_measure;
       cum_per_corr = (total_for_digits - wrong_for_digits) /
                       total_for_digits;

       printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
               percent_corr, cum_per_corr) ;
       fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
               percent_corr, cum_per_corr);
 printf ("    Total wrong formant words are %3.2f from a total of %3.2f words\n\n
",
        wrong_for_digits, total_for_digits);
fprintf (flog,"    Total wrong formant words are %3.2f from a total of %3.2f word
s\n\n",
        wrong_for_digits, total_for_digits);
```

```
) /** end fdtw routine **/


/*****************************************************************************
                    THIS ROUTINE SORTS DTW ARRAYS.                          **/

sort_best (best_words, best_distances, total_templates, i_row)

        int             best_words[][library_size], total_templates, i_row;
        float           best_distances[][library_size];
{
        int             in, out, temp_word;
        float           temp_distance;


        for (out = 0; out < total_templates ; out++)
           for (in = 0; in < total_templates ; in++)
              if (best_distances[i_row][out] < best_distances[i_row][in])
               {
                temp_word = best_words[i_row][in];
                temp_distance = best_distances[i_row][in];

                best_words[i_row][in] = best_words[i_row][out];
                best_distances[i_row][in] = best_distances[i_row][out];

                best_words[i_row][out] = temp_word;
                best_distances[i_row][out] = temp_distance;
               }

 }

/*****************************************************************************
        THIS ROUTINE PERFORMS DYNAMIC TIME WARPING ON LPC DATA.         **/

cdtw (t_array, u_array, t_length, u_length, flog)
        int     t_array[][1000][2], u_array[][2] ;
        int     t_length[library_size], u_length;
        FILE    *flog ;
{
        float   accum_dist[2][library_size][1000] ;
        int     utterance[20];
        int     back_ptr[2][library_size][1000] ;
        int     from_template[2000] ;
        int     from_frame[2000] ;

        float   d1, d2, d3, dist ;
        float   min_dist ;
        register int j, k, ptr, b_ptr, i, kk;

        int     best_words[library_size], lzero_flag[20];
        float   cum_per_corr, distance_measure;
        float   best_distances[library_size];
        int     found_match, finished, utt_count;
        int     new_best_choice_found, k1;

        ptr = 0 ;
        for (k = 0 ; k < num_templates ; k++) {
                for (j = 0 ; j < t_length[k] ; j++) {
                        if (j == 0) {
                                accum_dist[ptr][k][j] = (dist =
                                        abs(u_array[0][0]- t_array[k][j][0]) +
                                        abs(u_array[0][1]- t_array[k][j][1]));
                                }
                        else {
                                accum_dist[ptr][k][j] = (dist += lvert * (
```



```
                                        abs(u_array[0][0] - t_array[k][j][0]) +
                                        abs(u_array[0][1] - t_array[k][j][1])));
                                }
                        back_ptr[ptr][k][j] = 0 ;
                        }
                }
        for (i = 1 ; i < u_length ; i++) {
                if (ptr == 0) {
                        ptr = 1 ;
                        b_ptr = 0 ;
                        }
                else {
                        ptr = 0 ;
                        b_ptr = 1 ;
                        }
                for (k = 0 ; k < num_templates ; k++) {
                    for (j = 0 ; j < t_length[k] ; j++) {
                        dist =  abs(u_array[i][0] - t_array[k][j][0]) +
                                abs(u_array[i][1] - t_array[k][j][1]) ;
                        if (j == 0) {
                                min_dist = 99999.0 ;
                                for (kk = 0 ; kk < num_templates ; kk++) {
                                    if (min_dist >
                                    accum_dist[b_ptr][kk][t_length[kk]-1])
                                        min_dist =
                                        accum_dist[b_ptr][kk][t_length[kk]-1];
                                        }
                                if (accum_dist[b_ptr][k][0] < min_dist) {
                                        accum_dist[ptr][k][0] = lhorz * dist +
                                          accum_dist[b_ptr][k][0] ;
                                        back_ptr[ptr][k][0] =
                                          back_ptr[b_ptr][k][0] ;
                                        }
                                else {
                                        accum_dist[ptr][k][0] =
                                          min_dist + dist ;
                                        back_ptr[ptr][k][0] = i-1 ;
                                        }
                                }
                        else {
                                d1 = accum_dist[b_ptr][k][j-1] + dist ;
                                d2 = accum_dist[ptr][k][j-1] + (lvert * dist) ;
                                d3 = accum_dist[b_ptr][k][j] + (lhorz * dist) ;
                                if (d2 <= d3 && d2 < d1) {
                                        accum_dist[ptr][k][j] = d2 ;
                                        back_ptr[ptr][k][j] =
                                                back_ptr[ptr][k][j-1] ;
                                        }
                                else if (d3 <= d2 && d3 < d1) {
                                        accum_dist[ptr][k][j] = d3 ;
                                        back_ptr[ptr][k][j] =
                                                back_ptr[b_ptr][k][j] ;
                                        }
                                else {
                                        accum_dist[ptr][k][j] = d1 ;
                                        back_ptr[ptr][k][j] =
                                                back_ptr[b_ptr][k][j-1] ;
                                        }
                                }
                        } /** next j **/
                    } /** next k **/

                min_dist = 99999.0;
                for (k = 0 ; k < num_templates ; k++)
                 {
                 word_lpc[i][k] = k;
```

```
                lpc_distance[i][k] = accum_dist[ptr][k][t_length[k]-1];
                if (min_dist > accum_dist[ptr][k][t_length[k]-1])
                 {
                  min_dist = accum_dist[ptr][k][t_length[k]-1] ;
                  kk = k ;
                 }
                }
               from_template[i] = kk ;
               from_frame[i] = back_ptr[ptr][kk][t_length[kk]-1];
    } /** next i **/
/*****************************************************************************/

        ptr = u_length - 1 ;
        i = -1 ;
        while ((ptr > 0) && (i < 99))
         {
          utterance[++i] = from_template[ptr];
          i--;
          lptr_array[++i] = ptr;
          lzero_flag[i] = 0;

          sort_best (word_lpc, lpc_distance, num_templates, ptr);

          for (k = 0 ; k < num_templates ; k++)
           {
             lword_array[i][k] = word_lpc[ptr][k];
             if (lpc_distance[ptr][0] != 0)
              {
               ldistance_array[i][k] = lpc_distance[ptr][k] /
                                       lpc_distance[ptr][0];
              }
             else
              {
               ldistance_array[i][k] = lpc_distance[ptr][k];
               lzero_flag[i] = 1;
              }
           } /** end for k **/

          ptr = from_frame[ptr];
         } /** end while **/
        num_lpc_utt = i + 1;
        k = 0;
        for (j = i ; j >= 0 ; j--)
         {
          if (utterance[j] == 10)
           {
            printf (".. ") ;
            fprintf (flog, ".. ") ;
           }
          else
           {
            printf ("%d ", utterance[j]) ;
            fprintf (flog, "%d ", utterance[j]) ;
            lutt_array[k++] = utterance[j] ;
           }
         }
        printf ("\n    Should be:  ") ;
        fprintf (flog, "\n    Should be:  ") ;
        for (j = 0 ; j < t_words ; j++)
         {
          printf ("%d ", tem_array[j]);
          fprintf (flog, "%d ", tem_array[j]) ;
         }
/*****************************************************************************
                    PRINT OUT THE BEST CHOICES                          ****/
```



```
        for (j = 0; j < num_lpc_utt; j++)
          {
           printf ("\n    Best choices are: ");
           fprintf (flog, "\n     Best choices are: ");

           found_match = 0;
           finished = 0;
           k1 = -1;
           do
           {
            k1++;

            printf ("\n     %s had a lpc distance of %5.2f and a ",
                   library_words[lword_array[j][k1]],
                    lpc_distance[lptr_array[j]][k1]);
            printf ("distance factor of %3.4f", ldistance_array[j][k1]);

   fprintf (flog, "\n        %s had a lpc distance of %5.2f and a ",
                    library_words[lword_array[j][k1]],
                    lpc_distance[lptr_array[j]][k1]);
   fprintf (flog, "distance factor of %3.4f", ldistance_array[j][k1]);

            if (num_lpc_utt == t_words)
             {
              if ((lword_array[j][k1] == tem_array[j]) || k1 > 10)
                found_match = 1;
             }
            else
             {
              if (k1 >= best_choices)
                found_match = 1;
             }
              if ((k1 >= best_choices - 1) && (found_match))
               finished = 1;
           } /** end do-while **/
           while (! finished);
          } /** end for j **/

        grader (tem_array, lutt_array, t_words, k, &distance_measure);

        total_lpc_digits += t_words;
        wrong_lpc_digits += distance_measure;
        cum_per_corr = (total_lpc_digits - wrong_lpc_digits) /
                        total_lpc_digits;
        printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
                percent_corr, cum_per_corr) ;
        fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
                percent_corr, cum_per_corr);
 printf ("    Total wrong lpc words are %3.2f from a total of %3.2f words\n\n",
         wrong_lpc_digits, total_lpc_digits);
fprintf (flog,"    Total wrong lpc words are %3.2f from a total of %3.2f words\n\
n",
         wrong_lpc_digits, total_lpc_digits);
) /** end cdtw routine **/

/*****************************************************************************
        THIS ROUTINE PERFORMS FUSION ON THE TWO FEATURES OF SPEECH
        AND WITH THE HELP OF THE RULE-BASED SYSTEM, OUTPUTS ONE
        RESULT.                                                         **/

feature_fusion (flog)

FILE    *flog;
{
     register int     k, ki, kk, i, j;
     int     maxcount[50], s, match_found, z, found_best, bad_match;
```

```
    int     utt_array[20], num_words;
    int     temp_lword_array[50][71], temp_fword_array[50][71];
    int     found_space = 0;
    float   temp_ldistance_array[50][71], temp_fdistance_array[50][71];
    float   s_inc, distance_measure, cum_per_corr, s1, s2, s3;
/*************************************************************************
    ONLY PERFORM FUSION IF THE NUMBER OF LPC AND FORMANT WORDS DETERMINED
    BY DTW IS 7 OR LESS WHEN THERE IS A MISMATCH.  IF NO MISMATCH, PERFORM
    FUSION FOR ANY NUMBER OF WORDS.
**/

 if ((num_lpc_utt <= 7 && num_for_utt <= 7) || (num_lpc_utt == num_for_utt))
 {
  for (s = 0; s <= 2 ; s++)
   {
/*************************************************************************
        THE THRESHOLD VALUES ARE DEFINED IN THIS AREA.                  **/
    if (s == 0)
        s_inc = 1.5;
    if (s == 1)
        s_inc = 2.0;
    if (s == 2)
        s_inc = 2.5;

/*************************************************************************/
  for (k = 0; k < num_templates; k++)
    num_words = num_lpc_utt;

/*************************************************************************
                        PRINT OUT HEADER                                **/
  printf ("\n The LPC-F1/F2 fusion results for LPC distances less");
  printf (" than %2.2f are:\n", s_inc);
  fprintf (flog, "\n The LPC-F1/F2 fusion results for LPC distances less");
  fprintf (flog, " than %2.2f are:\n",s_inc);

/*************************************************************************
                    TEST LPC AND FORMANT LENGTHS
                PERFORM FUSION BASED ON LENGTH COMPARISON

*************************************************************************
          FIRST CHECK IF THE LPC RESULT GAVE AN ANSWER OF 2
          AND FORMANT RESULT IS 1. IF SO, IF ONE OF THE LPC
          RESULTS MATCHES THE FORMANT RESULT, ASSUME THAT IS
          THE CORRECT ANSWER. SET NUM WORDS TO 1 AND USE ONLY
          THE LPC/FORMANT MATCH TO COMPUTE THE FUSION RESULT.          **/

    if (num_lpc_utt == 2 && num_for_utt == 1)
     {
     if (lword_array[0][0] == fword_array[0][0])
      {
       /** 2nd lpc word matches the formant word **/
       z = 1;
       num_words = 1;
       num_lpc_utt = 1;
      }
     if (lword_array[1][0] == fword_array[0][0])
      {
       /** 1st lpc word matches the formant word **/
       z = 0;
       num_words = 1;
       num_lpc_utt = 1;
       for (k = 0; k < num_templates; k++)
        {
         lword_array[0][k] = lword_array[1][k];
         ldistance_array[0][k] = ldistance_array[1][k];
        }
```



```
      ) /** end if **/
     ) /** end if num_lpc_utt == 2 and num_for_utt == 1 **/

/*************************************************************************
            IF A MATCH WAS NOT PRESENT, DELETE A SPACE IF
            IT APPEARS IF IT PROBABLY IS REALLY A SPACE.
            NO GUARANTEES, JUST A BEST GUESS.                           **/

    if (num_lpc_utt == 2)
     {
     if (lword_array[0][0] == 10 || lword_array[1][0] == 10)
      {
       if (lword_array[0][0] == 10) /** is space the 2nd word **/
       {
        z = 1;
        found_space = 1;
       }
       if (lword_array[1][0] == 10) /** is space the 1st word **/
       {
        z = 0;
        found_space = 1;
       }
       if (found_space)
        {
         s2 = lpc_distance[lptr_array[0]][0]; /** 2nd word **/
         s1 = lpc_distance[lptr_array[1]][0]; /** 1st word **/

            /** if space is the second word **/
        if (z == 1 && s1 < 120.0 && s2 > 200.0)
         num_words = 1;
           /** if space is the first word **/
        if (z == 0 && s2 < 120.0 && s1 > 200.0)
         {
         for (k = 0; k < num_templates; k++)
          {
          lword_array[0][k] = lword_array[1][k];
          ldistance_array[0][k] = ldistance_array[1][k];
          }
         num_words = 1;
         } /** end if **/
        } /** if found space **/
      } /** end if a space found **/
     } /** end if num_lpc_utt == 2 **/

/*************************************************************************
                        EQUAL LPC AND FORMANT LENGTHS                   **/

    if (num_lpc_utt == num_for_utt)
      {
         num_words = num_lpc_utt;
         for (i = 0; i < num_lpc_utt; i++)
          {
           maxcount[i] = 0;
           for (k = 0; k < num_templates; k++)
            {
              if (ldistance_array[i][k] < s_inc)
              {
               maxcount[i]++;
               match_found = 0;
               ki = -1;
               while (! match_found || ki < 70)
                {
                  ki++;
                  if (lword_array[i][k] == fword_array[i][ki])
                    {
                     temp_lword_array[i][k] = lword_array[i][k];
```

```
                    formant_lpc_distance[i][k] = ldistance_array[i][k] *
                                                 fdistance_array[i][ki];
                    match_found = 1;
                   ) /** end if **/
                 ) /** end while **/

               ) /** end if **/
              ) /** end for k **/
             sort_best (temp_lword_array, formant_lpc_distance, maxcount[i], i);
            ) /** end for i **/

           printf (" The computed utterance string is:  ");
           fprintf (flog," The computed utterance string is:  ");

          k = 0;
          for (j = num_words - 1 ; j >= 0 ; j--)
           {
            if (temp_lword_array[j][0] == 10)
             {
              printf (".. ");
              fprintf (flog, ".. ");
             }
            else
             {
              printf ("%d ", temp_lword_array[j][0]);
              fprintf (flog, "%d ", temp_lword_array[j][0]);
              utt_array[k++] = temp_lword_array[j][0];
             }
           }
         ) /** end if **/

/*************************************************************************
                    UNEQUAL LPC/FORMANT RECOGNITION MISMATCH          **/

        if (num_lpc_utt != num_for_utt)
         {
          found_best = 0;
          match_found = 0;
/*************************************************************************
                         ISOLATED WORD MISMATCH                       **/

           if (num_lpc_utt == 1 && num_for_utt == 2)
            {
             maxcount[0] = 0;
             num_words = 1;
               s1 = formant_distance[fptr_array[0]][0];
               s2 = formant_distance[fptr_array[1]][0];

               if (s1 * 2.5 <= s2)
                {
                 z = 0;
                 found_best = 1;
                }
               if (s2 * 2.5 < s1)
                {
                 z = 1;
                 found_best = 1;
                }

               if (found_best)
                {
                 for (k = 0; k < num_templates; k++)
                  {
                   temp_fdistance_array[0][k] = fdistance_array[z][k];
                   temp_fword_array[0][k] = fword_array[z][k];
                  )
```



```
               ) /** end if **/
              else
               {
                /**** ADD FORMANT FACTORS TOGETHER ****/
                for (k = 0; k < num_templates; k++)
                 {
                  match_found = 0;
                  kk = -1;
                  while (! match_found || kk <= 70)
                   {
                    kk++;
                    if (fword_array[0][k] == fword_array[1][kk])
                     {
                      temp_fword_array[0][k] = fword_array[0][k];
                      temp_fdistance_array[0][k] =
                      fdistance_array[0][k] + fdistance_array[1][kk];
                      match_found = 1;
                     }
                   ) /** end while **/
                 ) /** end for k **/
               ) /** end if **/
             ) /** end if **/

/*************************************************************************/
        if (num_lpc_utt == 1 && num_for_utt == 3)
        {
              maxcount[0] = 0;
              num_words = 1;
              s1 = formant_distance[fptr_array[0]][0];
              s2 = formant_distance[fptr_array[1]][0];
              s3 = formant_distance[fptr_array[2]][0];
              if (s1 * 2.5 <= s2 && s1 * 2.5 <= s3)
               {
                z = 0;
                found_best = 1;
               }
              if (s2 * 2.5 < s1 && s2 * 2.5 < s3)
               {
                z = 1;
                found_best = 1;
               }
              if (s3 * 2.5 < s1 && s3 * 2.5 < s2)
               {
                z = 2;
                found_best = 1;
               }

              if (found_best)
               {
                for (k = 0; k < num_templates; k++)
                 {
                  for (k = 0; k < num_templates; k++)
                   {
                    temp_fdistance_array[0][k] = fdistance_array[z][k];
                    temp_fword_array[0][k] = fword_array[z][k];
                   }
                 ) /** end for **/
               ) /** end if **/
              else
               {
              /**** ADD FORMANT FACTORS TOGETHER:  PASS #1****/
              for (k = 0; k < num_templates; k++)
               {
                match_found = 0;
                kk = -1;
                while (! match_found || kk <= 70)
```

```
                {
                 kk++;
                 if (fword_array[0][k] == fword_array[1][kk])
                  {
                   temp_fword_array[0][k] = fword_array[0][k];
                   temp_fdistance_array[0][k] =
                   fdistance_array[0][k] + fdistance_array[1][kk];
                   match_found = 1;
                  }
                } /** end while **/
              } /** end for k **/

             /**** ADD FORMANT FACTORS TOGETHER: PASS #2 ****/
             for (k = 0; k < num_templates; k++)
              {
               match_found = 0;
               kk = -1;
                while (! match_found || kk <= 70)
                {
                 kk++;
                 if (temp_fword_array[0][k] == fword_array[2][kk])
                  {
                   temp_fdistance_array[0][k] += fdistance_array[2][kk];
                   match_found = 1;
                  }
                } /** end while **/
              } /** end for k **/
            } /** end if found best **/
          } /** end if num_for_utt == 3 **/

/*****************************************************************************/
        if ((num_lpc_utt == 1 && num_for_utt == 2) ||
           (num_lpc_utt == 1 && num_for_utt == 3))
           {
           /**** COMBINE LPC AND FORMANT DISTANCES ****/
           for (k = 0; k < num_templates; k++)
            {
             if (ldistance_array[0][k] < s_inc)
              {
               maxcount[0]++;
               match_found = 0;
               kk = -1;
               while (! match_found || kk <= 70)
               {
                kk++;
                  if (lword_array[0][k] == temp_fword_array[0][kk])
                   {
                    temp_lword_array[0][k] = lword_array[0][k];
                    formant_lpc_distance[0][k] = (temp_fdistance_array[0][kk] *
                                                  ldistance_array[0][k]);
                    match_found = 1;
                   }
               } /** end while **/
              } /** end if **/
            } /** end for k **/
           sort_best (temp_lword_array, formant_lpc_distance, maxcount[0], 0);

         printf (" The computed utterance string is:  ");
         fprintf (flog," The computed utterance string is:  ");

        k = 0;
        for (j = num_words - 1 ; j >= 0 ; j--)
         {
          if (temp_lword_array[j][0] == 10)
           {
            printf (".. ");
```



```
            fprintf (flog, ".. ");
           }
          else
           {
            printf ("%d ", temp_lword_array[j][0]);
            fprintf (flog, "%d ", temp_lword_array[j][0]);
            utt_array[k++] = temp_lword_array[j][0];
           }
         }
          } /** end if **/

/*****************************************************************************/

        if (num_for_utt == 1 && num_lpc_utt == 2)
         {
             num_words = 1;
             maxcount[0] = 0;
             found_best = 0;
             match_found = 0;
             /**** ADD  LPC DISTANCES TOGETHER IF A GOOD DISTANCE
                  MATCH WAS NOT FOUND                                    ****/
             s1 = lpc_distance[lptr_array[0]][0];
             s2 = lpc_distance[lptr_array[1]][0];

             if (s1 < 120.0 && s2 > 120.0)
                 {
                   z = 0;
                   found_best = 1;
                 }
              if (s2 < 120.0 && s1 > 120.0)
               {
                  z = 1;
                  found_best = 1;
               }

             if (found_best)
             {
               /*** COME HERE IF A SMALL DISTANCE WAS FOUND ***/
               for (k = 0; k < num_templates; k++)
                {
                  temp_ldistance_array[0][k] = ldistance_array[z][k];
                  temp_lword_array[0][k] = lword_array[z][k];
                }
              } /** end if **/
             else
              {
                 /*** COME HERE IF A SMALL DISTANCE WAS NOT FOUND SO
                      DISTANCES CAN BE ADDED.
                  ***/
                  if (s1 <= s2)
                    z = 0;
                  if (s1 > s2)
                    z = 1;
                 for (k = 0; k < num_templates; k++)
                  {
                   match_found = 0;
                   kk = -1;
                   while (! match_found || kk <= 70)
                    {
                     kk++;
                     if (lword_array[0][k] == lword_array[1][kk])
                      {
                       temp_lword_array[0][k] = lword_array[0][k];
                       temp_ldistance_array[0][k] = ldistance_array[0][k] +
                                                    ldistance_array[1][kk];
                       match_found = 1;
```

```
                    }
                  } /** end while **/
                } /** end for k **/
              } /** end if found best **/
            } /** end if num_for_utt == 1 && num_lpc_utt == 2 **/

/*****************************************************************************/
        if (num_for_utt == 1 && num_lpc_utt == 3)
         {
            num_words = 1;
            maxcount[0] = 0;
            found_best = 0;
            /****ADD  LPC DISTANCES TOGETHER IF A GOOD DISTANCE
                  MATCH WAS NOT FOUND                                  ****/
            s1 = lpc_distance[lptr_array[0]][0];
            s2 = lpc_distance[lptr_array[1]][0];
            s3 = lpc_distance[lptr_array[2]][0];


            if (s1 < 120.0 && s2 > 120.0 && s3 > 120.0)
               {
                 z = 0;
                 found_best = 1;
               }
             if (s2 < 120.0 && s1 > 120.0 && s3 > 120.0)
              {
                 z = 1;
                 found_best = 1;
              }
             if (s3 < 120.0 && s1 > 120.0 && s2 > 120.0)
              {
                 z = 2;
                 found_best = 1;
              }

            if (found_best)
             {
               /*** COME HERE IF A SMALL DISTANCE WAS FOUND ***/
               for (k = 0; k < num_templates; k++)
                {
                  temp_ldistance_array[0][k] = ldistance_array[z][k];
                  temp_lword_array[0][k] = lword_array[z][k];
                }
             } /** end if **/
            else
             {
                 /*** COME HERE IF A SMALL DISTANCE WAS NOT FOUND SO
                      DISTANCES CAN BE ADDED.
                  ***/
                if (s1 < s2 && s1 < s3)
                   z = 0;
                if (s2 < s1 && s2 < s3)
                   z = 1;
                if (s3 <= s1 && s3 <= s2)
                   z = 2;

                 for (k = 0; k < num_templates; k++)
                  {
                   for (i = 1; i <= 2; i++)
                    {
                     match_found = 0;
                     kk = -1;
                     while (! match_found || kk <= 70)
                      {
                       kk++;
                       if (lword_array[0][k] == lword_array[i][kk])
```



```
                        {
                         temp_lword_array[0][k] = lword_array[0][k];
                         temp_ldistance_array[0][k] += ldistance_array[0][k] +
                                                       ldistance_array[i][kk];
                         match_found = 1;
                        }
                       } /** end while **/
                     } /** end for i **/
                   } /** end for k **/
                 } /** end if found match **/
               } /** end if num_for_utt == 1 && num_lpc_utt == 3 **/

/*****************************************************************************/
        if ((num_for_utt == 1 && num_lpc_utt == 2) ||
            (num_for_utt == 1 && num_lpc_utt == 3))
            {
                  /*** NOW PERFORM FUSION ***/
                  for (k = 0; k < num_templates; k++)
                   {
                    if (ldistance_array[z][k] < s_inc)
                     {
                      maxcount[0]++;
                      match_found = 0;
                      kk = -1;
                      while (! match_found || kk <= 70)
                       {
                        kk++;
                        if (temp_lword_array[0][k] == fword_array[0][kk])
                         {
                          formant_lpc_distance[0][k] =
                          fdistance_array[0][kk] * temp_ldistance_array[0][k];
                          match_found = 1;
                         }
                       } /** end while **/
                     } /** end if **/
                   } /** end for k **/
              sort_best (temp_lword_array, formant_lpc_distance, maxcount[0], 0);

             printf (" The computed utterance string is:  ");
             fprintf (flog," The computed utterance string is:  ");

            k = 0;
            for (j = num_words - 1 ; j >= 0 ; j--)
             {
              if (temp_lword_array[j][0] == 10)
               {
                printf (".. ");
                fprintf (flog, ".. ");
               }
              else
               {
                printf ("%d ", temp_lword_array[j][0]);
                fprintf (flog, "%d ", temp_lword_array[j][0]);
                utt_array[k++] = temp_lword_array[j][0];
               }
             }
          } /** end if  for == 1 and lpc == 2 or 3 **/

/*****************************************************************************
                      THE FOLLOWING CASES PRESUME
                    CONNECTED SPEECH IS SPOKEN BUT
                    THE NUMBER OF LPC AND FORMANT
                       WORDS DO NOT MATCH.                          (XYZ)
*****************************************************************************/

      if ((num_lpc_utt >= 2 && num_for_utt >= 2) &&
```

```
    (num_lpc_utt != num_for_utt))
  {
  /*************************************
      DELETE SPACES IF NECESSARY IF
       NUM_LPC_UTT > NUM_FOR_UTT      **/
  if (num_lpc_utt > num_for_utt)
   {
    for (k = 0; k < num_lpc_utt; k++)
     {
      if ((lword_array[k][0] == 10) && (num_lpc_utt > num_for_utt) &&
          (k != num_lpc_utt - 1))
       {
        for (kk = k; kk < num_lpc_utt - 1; kk++)
         {
          for (j = 0; j < num_templates; j++)
           {
            lword_array[kk][j] = lword_array[kk+1][j];
            ldistance_array[kk][j] = ldistance_array[kk+1][j];
           } /** end for **/
         } /** end for **/
        num_lpc_utt--;
       } /** end if **/
      if ((lword_array[k][0] == 10) && (num_lpc_utt > num_for_utt) &&
          (k == num_lpc_utt - 1))
            num_lpc_utt--;
      } /** end for **/
    } /** end if **/


  /*************************************/
  if (num_lpc_utt == 3 && num_for_utt == 2)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
     } /** end for k **/
  } /** end if lpc = 3 and for = 2 **/

  /*************************************/
  else if (num_lpc_utt == 4 && num_for_utt == 2)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 4 and for = 2 **/

  /*************************************/
  else if (num_lpc_utt == 5 && num_for_utt == 2)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
```



```
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
      fword_array[4][k] = fword_array[1][k];
      fdistance_array[4][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 5 and for = 2 **/

  /*************************************/
  else if (num_lpc_utt == 6 && num_for_utt == 2)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
      fword_array[4][k] = fword_array[1][k];
      fdistance_array[4][k] = fdistance_array[1][k];
      fword_array[5][k] = fword_array[1][k];
      fdistance_array[5][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
      fword_array[2][k] = fword_array[0][k];
      fdistance_array[2][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 6 and for = 2 **/

  /*************************************************/
  else if (num_lpc_utt == 7 && num_for_utt == 2)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
      fword_array[4][k] = fword_array[1][k];
      fdistance_array[4][k] = fdistance_array[1][k];
      fword_array[5][k] = fword_array[1][k];
      fdistance_array[5][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
      fword_array[2][k] = fword_array[0][k];
      fdistance_array[2][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 7 and for = 2 **/

  /*************************************/
  else if (num_lpc_utt == 4 && num_for_utt == 3)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[2][k];
      fdistance_array[3][k] = fdistance_array[2][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
```

```
    {
     fword_array[2][k] = fword_array[1][k];
     fdistance_array[2][k] = fdistance_array[1][k];
    } /** end for k **/
  } /** end if lpc = 4 and for = 3 **/

  /*************************************/
  else if (num_lpc_utt == 5 && num_for_utt == 3)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[2][k];
      fdistance_array[3][k] = fdistance_array[2][k];
      fword_array[4][k] = fword_array[2][k];
      fdistance_array[4][k] = fdistance_array[2][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
     } /** end for k **/
  } /** end if lpc = 5 and for = 3 **/

  /*************************************/
  else if (num_lpc_utt == 5 && num_for_utt == 4)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[4][k] = fword_array[3][k];
      fdistance_array[4][k] = fdistance_array[3][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[2][k];
      fdistance_array[3][k] = fdistance_array[2][k];
     } /** end for k **/

  } /** end if lpc = 5 and for = 4 **/

  /*************************************/
  else if (num_lpc_utt == 6 && num_for_utt == 3)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[4][k] = fword_array[2][k];
      fdistance_array[4][k] = fdistance_array[2][k];
      fword_array[5][k] = fword_array[2][k];
      fdistance_array[5][k] = fdistance_array[2][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 6 and for = 3 **/
```



```
  /*************************************/
  else if (num_lpc_utt == 6 && num_for_utt == 4)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[5][k] = fword_array[3][k];
      fdistance_array[5][k] = fdistance_array[3][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[3][k] = fword_array[2][k];
      fdistance_array[3][k] = fdistance_array[2][k];
      fword_array[4][k] = fword_array[2][k];
      fdistance_array[4][k] = fdistance_array[2][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
     } /** end for k **/
  } /** end if lpc = 6 and for = 4 **/

  /*************************************/
  else if (num_lpc_utt == 6 && num_for_utt == 5)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[5][k] = fword_array[4][k];
      fdistance_array[5][k] = fdistance_array[4][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[4][k] = fword_array[3][k];
      fdistance_array[4][k] = fdistance_array[3][k];
     } /** end for k **/

  } /** end if lpc = 6 and for = 5 **/

  /*************************************/
  else if (num_lpc_utt == 7 && num_for_utt == 3)
  {
   num_words = num_lpc_utt;
    for (k = 0; k < num_templates; k++)
     {
      fword_array[5][k] = fword_array[2][k];
      fdistance_array[5][k] = fdistance_array[2][k];
      fword_array[6][k] = fword_array[2][k];
      fdistance_array[6][k] = fdistance_array[2][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[2][k] = fword_array[1][k];
      fdistance_array[2][k] = fdistance_array[1][k];
      fword_array[3][k] = fword_array[1][k];
      fdistance_array[3][k] = fdistance_array[1][k];
      fword_array[4][k] = fword_array[1][k];
      fdistance_array[4][k] = fdistance_array[1][k];
     } /** end for k **/
    for (k = 0; k < num_templates; k++)
     {
      fword_array[1][k] = fword_array[0][k];
      fdistance_array[1][k] = fdistance_array[0][k];
     } /** end for k **/
  } /** end if lpc = 7 and for = 3 **/
```

```
/*****************************************/
else if (num_lpc_utt == 7 && num_for_utt == 4)
{
 num_words = num_lpc_utt;
  for (k = 0; k < num_templates; k++)
   {
    fword_array[5][k] = fword_array[3][k];
    fdistance_array[5][k] = fdistance_array[3][k];
    fword_array[6][k] = fword_array[3][k];
    fdistance_array[6][k] = fdistance_array[3][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[3][k] = fword_array[2][k];
    fdistance_array[3][k] = fdistance_array[2][k];
    fword_array[4][k] = fword_array[2][k];
    fdistance_array[4][k] = fdistance_array[2][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[2][k] = fword_array[1][k];
    fdistance_array[2][k] = fdistance_array[1][k];
   } /** end for k **/
} /** end if lpc = 7 and for = 4 **/

/*****************************************/
else if (num_lpc_utt == 7 && num_for_utt == 5)
{
 num_words = num_lpc_utt;
  for (k = 0; k < num_templates; k++)
   {
    fword_array[6][k] = fword_array[4][k];
    fdistance_array[6][k] = fdistance_array[4][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[4][k] = fword_array[3][k];
    fdistance_array[4][k] = fdistance_array[3][k];
    fword_array[5][k] = fword_array[3][k];
    fdistance_array[5][k] = fdistance_array[3][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[3][k] = fword_array[2][k];
    fdistance_array[3][k] = fdistance_array[2][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[2][k] = fword_array[1][k];
    fdistance_array[2][k] = fdistance_array[1][k];
   } /** end for k **/
} /** end if lpc = 7 and for = 5 **/

/*****************************************/
else if (num_lpc_utt == 7 && num_for_utt == 6)
{
 num_words = num_lpc_utt;
  for (k = 0; k < num_templates; k++)
   {
    fword_array[6][k] = fword_array[5][k];
    fdistance_array[6][k] = fdistance_array[5][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    fword_array[5][k] = fword_array[4][k];
```



```
    fdistance_array[5][k] = fdistance_array[4][k];
   } /** end for k **/
} /** end if lpc = 7 and for = 6 **/

/*****************************************/
else if (num_lpc_utt == 2 && num_for_utt == 3)
{
 num_words = 3;
  for (k = 0; k < num_templates; k++)
   {
    lword_array[2][k] = lword_array[1][k];
    ldistance_array[2][k] = ldistance_array[1][k];
   } /** end for k **/
} /** end if lpc = 2 and for = 3 **/

/*****************************************/
else if (num_lpc_utt == 2 && num_for_utt == 4)
{
 num_words = 3 ;
  for (k = 0; k < num_templates; k++)
   {
    lword_array[2][k] = lword_array[1][k];
    ldistance_array[2][k] = ldistance_array[1][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
     match_found = 0;
     ki = -1;
     while (! match_found || ki < 70)
      {
        ki++;
        if (fword_array[2][k] == fword_array[3][ki])
          {
           fdistance_array[2][k] = (0.5 * fdistance_array[2][k]) +
                                   (0.5 * fdistance_array[3][ki]);
           match_found = 1;
          } /** end if **/
      } /** end while **/
   } /** end for k **/
} /** end if lpc = 2 and for = 4 **/

/*****************************************/
else if (num_lpc_utt == 2 && num_for_utt == 5)
{
 num_words = 4;
  for (k = 0; k < num_templates; k++)
   {
    lword_array[2][k] = lword_array[1][k];
    ldistance_array[2][k] = ldistance_array[1][k];
    lword_array[3][k] = lword_array[1][k];
    ldistance_array[3][k] = ldistance_array[1][k];
   } /** end for k **/
  for (k = 0; k < num_templates; k++)
   {
    lword_array[1][k] = lword_array[0][k];
    ldistance_array[1][k] = ldistance_array[0][k];
   } /** end for k **/

  for (k = 0; k < num_templates; k++)
   {
     match_found = 0;
     ki = -1;
     while (! match_found || ki < 70)
      {
        ki++;
        if (fword_array[1][k] == fword_array[2][ki])
```

```
            {
             fdistance_array[1][k] = (0.5 * fdistance_array[1][k]) +
                                     (0.5 * fdistance_array[2][ki]);
             match_found = 1;
            } /** end if **/
          } /** end while **/
        } /** end for k **/
       for (k = 0; k < num_templates; k++)
        {
         fword_array[2][k] = fword_array[3][k];
         fdistance_array[2][k] = fdistance_array[3][k];
         fword_array[3][k] = fword_array[4][k];
         fdistance_array[3][k] = fdistance_array[4][k];
        } /** end for k **/
      } /** end if lpc = 2 and for = 5 **/

      /*************************************/
      else if (num_lpc_utt == 3 && num_for_utt == 4)
      {
          words = 3;
           (k = 0; k < num_templates; k++)
        {
          match_found = 0;
          ki = -1;
          while (! match_found || ki < 70)
           {
             ki++;
             if (fword_array[1][k] == fword_array[2][ki])
               {
                fdistance_array[1][k] = (0.5 * fdistance_array[1][k]) +
                                        (0.5 * fdistance_array[2][ki]);
                match_found = 1;
               } /** end if **/
           } /** end while **/
        } /** end for k **/
       for (k = 0; k < num_templates; k++)
        {
         fword_array[2][k] = fword_array[3][k];
         fdistance_array[2][k] = fdistance_array[3][k];
        } /** end for k **/
      } /** end if lpc = 3 and for = 4 **/

      /*************************************/
      else if (num_lpc_utt == 3 && num_for_utt == 5)
      {
       num_words = 4;
        for (k = 0; k < num_templates; k++)
         {
          lword_array[3][k] = lword_array[2][k];
          ldistance_array[3][k] = ldistance_array[2][k];
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
          lword_array[2][k] = lword_array[1][k];
          ldistance_array[2][k] = ldistance_array[1][k];
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
           match_found = 0;
           ki = -1;
           while (! match_found || ki < 70)
            {
              ki++;
              if (fword_array[2][k] == fword_array[3][ki])
                {
                 fdistance_array[2][k] = (0.5 * fdistance_array[2][k]) +
```



```
                                         (0.5 * fdistance_array[3][ki]);
                 match_found = 1;
                } /** end if **/
            } /** end while **/
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
          fword_array[3][k] = fword_array[4][k];
          fdistance_array[3][k] = fdistance_array[4][k];
         } /** end for k **/
      } /** end if lpc = 3 and for = 5 **/

      /*************************************/
      else if (num_lpc_utt == 4 && num_for_utt == 5)
      {
       num_words = 4;
        for (k = 0; k < num_templates; k++)
         {
           match_found = 0;
           ki = -1;
           while (! match_found || ki < 70)
            {
              ki++;
              if (fword_array[2][k] == fword_array[3][ki])
                {
                 fdistance_array[2][k] = (0.5 * fdistance_array[2][k]) +
                                         (0.5 * fdistance_array[3][ki]);
                 match_found = 1;
                } /** end if **/
            } /** end while **/
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
          fword_array[3][k] = fword_array[4][k];
          fdistance_array[3][k] = fdistance_array[4][k];
         } /** end for k **/
      } /** end if lpc = 4 and for = 5 **/

      /*************************************/
      else if (num_lpc_utt == 4 && num_for_utt == 6)
      {
       num_words = 5;
        for (k = 0; k < num_templates; k++)
         {
          lword_array[4][k] = lword_array[3][k];
          ldistance_array[4][k] = ldistance_array[3][k];
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
          lword_array[3][k] = lword_array[2][k];
          ldistance_array[3][k] = ldistance_array[2][k];
         } /** end for k **/
        for (k = 0; k < num_templates; k++)
         {
           match_found = 0;
           ki = -1;
           while (! match_found || ki < 70)
            {
              ki++;
              if (fword_array[3][k] == fword_array[4][ki])
                {
                 fdistance_array[3][k] = (0.5 * fdistance_array[3][k]) +
                                         (0.5 * fdistance_array[4][ki]);
                 match_found = 1;
                } /** end if **/
            } /** end while **/
```

```
          ) /** end for k **/
         for (k = 0; k < num_templates; k++)
          {
           fword_array[4][k] = fword_array[5][k];
           fdistance_array[4][k] = fdistance_array[5][k];
          ) /** end for k **/
        ) /** end if lpc = 4 and for = 6 **/
       else
        {
         if (num_lpc_utt < num_for_utt)
           num_words = num_lpc_utt;
         else
          {
           num_words - num_for_utt;
           num_lpc_utt = num_for_utt;
          }
        ) /** end if else statements **/
/*****************************************************************************
                    NOW PERFORM FUSION ON UNEQUAL SPEECH                  **/

       for (i = 0; i < num_words; i++)
        {
         maxcount[i] = 0;
         for (k = 0; k < num_templates; k++)
          {
            if (ldistance_array[i][k] < s_inc)
            {
             maxcount[i]++;
             match_found = 0;
             ki = -1;
             while (! match_found || ki < 70)
              {
                ki++;
                if (lword_array[i][k] == fword_array[i][ki])
                  {
                    temp_lword_array[i][k] = lword_array[i][k];
                    formant_lpc_distance[i][k] = ldistance_array[i][k] *
                                                 fdistance_array[i][ki];
                    match_found = 1;
                  ) /** end if **/
              ) /** end while **/

            ) /** end if **/
          ) /** end for k **/
         sort_best (temp_lword_array, formant_lpc_distance, maxcount[i], i);
        ) /** end for i **/

       printf (" The computed utterance string is:  ");
       fprintf (flog," The computed utterance string is:  ");

      k = 0;
      for (j = num_words - 1 ; j >= 0 ; j--)
       {
        if (temp_lword_array[j][0] == 10)
         {
          printf (".. ");
          fprintf (flog, ".. ");
         }
        else
         {
          printf ("%d ", temp_lword_array[j][0]);
          fprintf (flog, "%d ", temp_lword_array[j][0]);
          utt_array[k++] = temp_lword_array[j][0];
         }
       )
      ) /** end if lpc >= 2 and for >= 2 and lpc != for **/
```



```
    ) /** end if **/

/*****************************************************************************
                    PRINT OUT THE BEST CHOICES                            ****/

        printf ("\n     Should be:  ");
        fprintf (flog, "\n     Should be:  ");
        for (j = 0; j < t_words; j++)
          {
            printf ("%d ", tem_array[j]);
            fprintf (flog, "%d ", tem_array[j]);
          ) /** end for j **/

       for (i = 0; i < num_words; i++)
        {
      printf (" \nThe # of words less than %2.1f are %d\n",s_inc,maxcount[i]);
        fprintf (flog, " \nThe # of words less than %2.1f are %d\n",
                                  s_inc, maxcount[i]);
          printf ("\n    Best choices are: ");
          fprintf (flog, "\n     Best choices are: ");

          if (maxcount[i] > 10)
            maxcount[i] = 10;
          for (k = 0; k < maxcount[i]; k++)
           {
           printf ("\n     %s had a formant-lpc distance factor of %3.4f",
                  library_words[temp_lword_array[i][k]],
                  formant_lpc_distance[i][k]);
           fprintf (flog, "\n       %s had a formant-lpc distance factor of %3.
4f",
                  library_words[temp_lword_array[i][k]],
                  formant_lpc_distance[i][k]);
           ) /** end for k **/

          ) /** end for i **/

       grader (tem_array, utt_array, t_words, num_words, &distance_measure);

       if (s == 0)
        {
         total_lpc_for_digits0 += t_words;
         wrong_lpc_for_digits0 += distance_measure;
         cum_per_corr = (total_lpc_for_digits0 - wrong_lpc_for_digits0) /
                        total_lpc_for_digits0;
         printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr) ;
         fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr);
 printf ("    Total wrong fusion words are %3.2f from a total of %3.2f words\n\n"
,
         wrong_lpc_for_digits0, total_lpc_for_digits0);
 fprintf (flog,"    Total wrong fusion words are %3.2f from a total of %3.2f word
s\n\n",
         wrong_lpc_for_digits0, total_lpc_for_digits0);
       }
       else if (s == 1)
       {
         total_lpc_for_digits1 += t_words;
         wrong_lpc_for_digits1 += distance_measure;
         cum_per_corr = (total_lpc_for_digits1 - wrong_lpc_for_digits1) /
                        total_lpc_for_digits1;
         printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr) ;
         fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr);
 printf ("    Total wrong fusion words are %3.2f from a total of %3.2f words\n\n"
```

```
,
        wrong_lpc_for_digits1, total_lpc_for_digits1);
 fprintf (flog,"    Total wrong fusion words are %3.2f from a total of %3.2f word
s\n\n",
        wrong_lpc_for_digits1, total_lpc_for_digits1);
       )
       else if (s == 2)
       {
         total_lpc_for_digits2 += t_words;
         wrong_lpc_for_digits2 += distance_measure;
         cum_per_corr = (total_lpc_for_digits2 - wrong_lpc_for_digits2) /
                        total_lpc_for_digits2;
         printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr) ;
         fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr);
 printf ("    Total wrong fusion words are %3.2f from a total of %3.2f words\n\n"
,
        wrong_lpc_for_digits2, total_lpc_for_digits2);
 fprintf (flog,"    Total wrong fusion words are %3.2f from a total of %3.2f word
s\n\n",
        wrong_lpc_for_digits2, total_lpc_for_digits2);
       }
       else if (s == 3)
       {
         total_lpc_for_digits3 += t_words;
         wrong_lpc_for_digits3 += distance_measure;
         cum_per_corr = (total_lpc_for_digits3 - wrong_lpc_for_digits3) /
                        total_lpc_for_digits3;
         printf ("\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr) ;
         fprintf (flog, "\n    correct = %5.3f    cum_correct = %5.3f\n",
              percent_corr, cum_per_corr);
 printf ("    Total wrong fusion words are %3.2f from a total of %3.2f words\n\n"
,
        wrong_lpc_for_digits3, total_lpc_for_digits3);
 fprintf (flog,"    Total wrong fusion words are %3.2f from a total of %3.2f word
s\n\n",
        wrong_lpc_for_digits3, total_lpc_for_digits3);
       }


 ) /** end for s **/
 ) /** end check for num_utt <= 7 **/
} /** end fusion routine **/

/************************ ***************************************************
       THIS ROUTINE, CALLED BY GET_VECTORS, IS USED IN PROCESSING
       LPC DATA.                                                          **/

mindist (net, inp, close)
       double          inp[16];
       int             close[2];
       float           net[20][20][16];
{
       register int            i, c, r;
       double                  dot_product;
       double                   maximum = 0.0;

       for (r = 0 ; r < ysize ; r++)
        {
         for (c = 0 ; c < xsize ; c++)
          {
           dot_product = 0.0 ;
            for (i = 0 ; i < number_inputs ; i++)
               dot_product += inp[i] * map[r][c][i];
```



```
              if (dot_product > maximum)
               {
                maximum = dot_product ;
                close[0] = c ;
                close[1] = r ;
               } /** end if **/
          } /** end for **/
        } /** end for **/
} /*** end mindist routine ***/

/*****************************************************************************
       THIS ROUTINE PROCESSES LPC TRAJECTORIES FROM THE KOHONEN
       NETWORK.  EACH TRAJECTORY IS REDUCED AS FOLLOWS:  EACH VECTOR
       SET REPRESENTING A TRAJECTORY IS PASSED THROUGH A TWO-STEP
       REDUCTION ALGORITHM.  IN THE FIRST STEP, A POINT ON THE TRAJECTORY,
       NOT WITHIN TWO NODE UNITS ON ANOTHER POINT ON THE TRAJECTORY WITHIN
       TWO TIME SLICES, IS ELIMINATED.  IN THE SECOND STEP, ALL POINTS
       ON THE TRAJECTORY WHICH ARE NOT PART OF THREE CONSECUTINE POINTS
       ARE ELIMINATED.                                                   **/

get_vectors (name, array, length)
       char    name[30] ;
       int     array[3000][2] ;
       int     *length;
{
       int     flag, r, c, i, j, k ;
       double  in[16], d1, d2, d3, d4, d5 ;
       float   element ;

       int     loc2[3000][2], loc3[3000][2] ;
       FILE    *fsnd ;
       int     sound, point, x[5], y[5] ;
       int     max_pts ;

       fsnd = fopen (name, "r") ;
       sound = 0 ;
       i = 0 ;
       flag = 0 ;
       while (flag != 1)
         {
           fscanf (fsnd, "%f", &element) ;
           if (feof(fsnd) != 0)
            {
              flag = 1 ;
            }
           else
            {
              in[i] = (double) element;
              i++ ;
            } /** end if **/
          /** Changed input number from 15 to number_inputs.
           **/
          if ((i == number_inputs) && (flag == 0))
           {
               mindist (map, in, &loc2[sound][0]);
               i = 0;
               sound++;
          } /** end if **/
        } /** end while **/
       fclose (fsnd) ;
       for (i = 0 ; i < sound ; i++) {
              point = loc2[i][0] + loc2[i][1] * xsize ;
              }
       for (i = 0 ; i < sound ; i++) {
              loc3[i][0] = loc2[i][0] ;
              loc3[i][1] = loc2[i][1] ;
```

```
        }
max_pts = sound ;
j = 0 ;

if (j != 0)
        max_pts = j ;
j = 0 ;
for (k = 0 ; k < 3 ; k++) {
        x[k] = loc3[k][0] ;
        y[k] = loc3[k][1] ;
        }
d1 = pow((double)(x[0]-x[2]), 2.0) +
     pow((double)(y[0]-y[2]), 2.0) ;
d2 = pow((double)(x[0]-x[1]), 2.0) +
     pow((double)(y[0]-y[1]), 2.0) ;
if ((d1 < 4.1) || (d2 < 4.1)) {
        loc2[j][0] = x[0] ;
        loc2[j][1] = y[0] ;
        j++ ;
        }
for (i = 1 ; i < max_pts-1 ; i++) {
        if (i == 1){
                for (k = 1 ; k < 5 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
                d2 = pow((double)(x[1]-x[2]), 2.0) +
                     pow((double)(y[1]-y[2]), 2.0) ;
                d3 = pow((double)(x[3]-x[2]), 2.0) +
                     pow((double)(y[3]-y[2]), 2.0) ;
                d4 = 5.0 ;
                d5 = pow((double)(x[4]-x[2]), 2.0) +
                     pow((double)(y[4]-y[2]), 2.0) ;
                }
        else if (i == max_pts-2) {
                for (k = 0 ; k < 4 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
                d2 = pow((double)(x[1]-x[2]), 2.0) +
                     pow((double)(y[1]-y[2]), 2.0) ;
                d3 = pow((double)(x[3]-x[2]), 2.0) +
                     pow((double)(y[3]-y[2]), 2.0) ;
                d4 = pow((double)(x[0]-x[2]), 2.0) +
                     pow((double)(y[0]-y[2]), 2.0) ;
                d5 = 5.0 ;
                }
        else {
                for (k = 0 ; k < 5 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
                d2 = pow((double)(x[2]-x[1]), 2.0) +
                     pow((double)(y[2]-y[1]), 2.0) ;
                d3 = pow((double)(x[3]-x[2]), 2.0) +
                     pow((double)(y[3]-y[2]), 2.0) ;
                d4 = pow((double)(x[2]-x[0]), 2.0) +
                     pow((double)(y[2]-y[0]), 2.0) ;
                d5 = pow((double)(x[4]-x[2]), 2.0) +
                     pow((double)(y[4]-y[2]), 2.0) ;
                }
        if ((d2 < 4.1) || (d3 < 4.1) || (d4 < 4.1)
           || (d5 < 4.1)) {
                loc2[j][0] = x[2] ;
                loc2[j][1] = y[2] ;
                j++ ;
```



```
                }
        }
for (k = 0 ; k < 3 ; k++) {
        x[k] = loc3[max_pts+k-3][0] ;
        y[k] = loc3[max_pts+k-3][1] ;
        }
d1 = pow((double)(x[0]-x[2]), 2.0) +
     pow((double)(y[0]-y[2]), 2.0) ;
d3 = pow((double)(x[1]-x[2]), 2.0) +
     pow((double)(y[1]-y[2]), 2.0) ;
if ((d1 < 4.1) || (d3 < 4.1)) {
        loc2[j][0] = x[2] ;
        loc2[j][1] = y[2] ;
        j++ ;
        }
for (i = 0 ; i < j ; i++) {
        point = loc2[i][0] + loc2[i][1] * xsize ;
        }
for (i = 0 ; i < j ; i++) {
        loc3[i][0] = loc2[i][0] ;
        loc3[i][1] = loc2[i][1] ;
        }

max_pts = j ;
j = 0 ;
for (i = 0 ; i < max_pts ; i++) {
        if (i == 0) {
                for (k = 2 ; k < 5 ; k++) {
                        x[k] = loc3[k-2][0] ;
                        y[k] = loc3[k-2][1] ;
                        }
                d1 = 5.0 ; d2 = 5.0 ;
                d3 = pow((double)(x[3]-x[2]), 2.0) +
                     pow((double)(y[3]-y[2]), 2.0) ;
                d4 = pow((double)(x[4]-x[2]), 2.0) +
                     pow((double)(y[4]-y[2]), 2.0) ;
                }
        else if (i == max_pts-1) {
                for (k = 0 ; k < 3 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
                d1 = pow((double)(x[0]-x[2]), 2.0) +
                     pow((double)(y[0]-y[2]), 2.0) ;
                d2 = pow((double)(x[1]-x[2]), 2.0) +
                     pow((double)(y[1]-y[2]), 2.0) ;
                d3 = 5.0 ; d4 = 5.0 ;
                }
        else if (i == 1) {
                for (k = 1 ; k < 5 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
                d1 = pow((double)(x[1]-x[2]), 2.0) +
                     pow((double)(y[1]-y[2]), 2.0) ;
                d2 = 5.0 ;
                d3 = pow((double)(x[3]-x[2]), 2.0) +
                     pow((double)(y[3]-y[2]), 2.0) ;
                d4 = pow((double)(x[4]-x[2]), 2.0) +
                     pow((double)(y[4]-y[2]), 2.0) ;
                }
        else if (i == max_pts-2) {
                for (k = 0 ; k < 4 ; k++) {
                        x[k] = loc3[i+k-2][0] ;
                        y[k] = loc3[i+k-2][1] ;
                        }
```

```
                        d1 = pow((double)(x[1]-x[2]), 2.0) +
                             pow((double)(y[1]-y[2]), 2.0) ;
                        d2 = pow((double)(x[0]-x[2]), 2.0) +
                             pow((double)(y[0]-y[2]), 2.0) ;
                        d4 = 5.0 ;
                        d3 = pow((double)(x[3]-x[2]), 2.0) +
                             pow((double)(y[3]-y[2]), 2.0) ;
                        }
                else {
                        for (k = 0 ; k < 5 ; k++) {
                                x[k] = loc3[i+k-2][0] ;
                                y[k] = loc3[i+k-2][1] ;
                                }
                        d1 = pow((double)(x[1]-x[2]), 2.0) +
                             pow((double)(y[1]-y[2]), 2.0) ;
                        d2 = pow((double)(x[0]-x[2]), 2.0) +
                             pow((double)(y[0]-y[2]), 2.0) ;
                        d3 = pow((double)(x[3]-x[2]), 2.0) +
                             pow((double)(y[3]-y[2]), 2.0) ;
                        d4 = pow((double)(x[4]-x[2]), 2.0) +
                             pow((double)(y[4]-y[2]), 2.0) ;
                        }
                if (((d1 < 4.1) && (d3 < 4.1)) ||
                    ((d1 < 4.1) && (d2 < 4.1)) ||
                    ((d3 < 4.1) && (d4 < 4.1))) {
                        loc2[j][0] = x[2] ;
                        loc2[j][1] = y[2] ;
                        j++ ;
                        }
                }
        *length = j ;
        for (i = 0 ; i < j ; i++) {
                array[i][0] = loc2[i][0] ;
                array[i][1] = loc2[i][1] ;
                }
}

/*****************************************************************************
        THIS ROUTINE GRADES THE DTW RESULTS . IT GIVES A +1 FOR A CORRECT
        DIGIT, -1 FOR A WRONG DIGIT, -1 FOR A DELETED DIGIT, AND -.5 FOR
        AN ADDITIONAL DIGIT.                                              **/

grader (template_array, utt_array, tt_words, utt_words, min_dist)
        int     template_array[library_size], utt_array[library_size] ;
        int     tt_words, utt_words;
        float   *min_dist;
{
        int     i, j, ptr, b_ptr ;
        float   back_ptr[2][library_size] ;
        float   d, min, dist, back ;

        back = 0.0 ;
        ptr = 1 ;
        for (i = 0 ; i < utt_words ; i++) {
                if (ptr == 0){
                        b_ptr = 0 ;
                        ptr = 1 ;
                        }
                else {
                        b_ptr = 1 ;
                        ptr = 0 ;
                        }
                if (i == 0) {
                        for (j = 0 ; j < tt_words ; j++) {
                                if (template_array[j] == utt_array[0])
                                        dist = 0.0 + back ;
```



```
                                else
                                        dist = 1.0 + back ;
                                back = dist ;
                                back_ptr[ptr][j] = dist ;
                                }
                        }
                else {
                        for (j = 0 ; j < tt_words ; j++) {
                                if (template_array[j] == utt_array[i])
                                        dist = 0.0 ;
                                else
                                        dist = 1.0 ;
                                if (j == 0)
                                        back_ptr[ptr][0] = back_ptr[b_ptr][0] +
                                                (0.5 * dist) ;
                                else {
                                        min = back_ptr[b_ptr][j] + (0.5*dist) ;
                                        d = back_ptr[b_ptr][j-1] + dist ;
                                        if (d < min)
                                                min = d ;
                                        d = back_ptr[ptr][j-1] + dist ;
                                        if (d < min)
                                                min = d ;
                                        back_ptr[ptr][j] = min ;
                                        }
                                }
                        }
                }
        if (utt_words > 0)
          *min_dist = back_ptr[ptr][tt_words - 1];
        else
          *min_dist = tt_words;

          percent_corr = (tt_words - *min_dist) / tt_words;
/**
        if (utt_words > 0)
         {
          *min_dist = back_ptr[ptr][tt_words-1];
          percent_corr = (tt_words - back_ptr[ptr][tt_words-1]) /
                                        tt_words;
         }
        else
         {
          *min_dist = tt_words;
          percent_corr = 0.0;
         }
**/

}
```

Bibliography

1. *SPIRE 17.2 Preliminary User's Guide.* Speech Comminication Group, Research Laboratory of Electronics, Massachussetts Institute of Technology, February 1986.
2. *SPIRE 17.2 Reference Manual.* Speech Comminication Group, Research Laboratory of Electronics, Massachussetts Institute of Technology, February 1986.
3. Gary Dean Barmore. Speech Recognition Using Neural Nets and Dynamic Time Warping. Article submitted for publication.
4. Gary Dean Barmore. *Speech Recognition Using Neural Nets and Dynamic Time Warping.* Master's thesis, AIR FORCE INSTITUTE OF TECHNOLOGY, 1988.
5. Bruno Beck et al. An Assessment of the Technology of Automatic Speech Recognition for Military Applications. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25(4):310–322, August 1977.
6. Peter Brauer and Petter Knagenhjelm. Infrastructure in Kohonen Maps. In *1989 International Conference on Neural Networks*, pages 647–650, IEEE Press, Glasgow Scotland, 1989.
7. Robert G. Dawson. *Spire Based Speaker-Independent Continuous Speech Recognition Using Mixed Feature Sets.* Master's thesis, AIR FORCE INSTITUTE OF TECHNOLOGY, 1987.
8. Duane DeSieno. Adding a Conscience to Competitive Learning. In *1987 International Conference on Neural Networks*, IEEE Press, San Diego California, 1987.
9. James L. Flanagan. The Synthesis of Speech. *Scientific American*, 226(2):48–58, February 1972.
10. David S. Harvey. Talking With Airplanes. *Air Force Magazine*, 71(1):88–96, January 1988.
11. William P. Jones and Josiah Hoskins. Back-Propagation. *BYTE*, 12(11):155–162, October 1987.
12. Robert H. Kassel. *A Users Guide to SPIRE.* MIT Speech Research Group, March 1985.
13. David H. Kauffman. *SPIRE 17 Release Notes.* MIT Speech Research Group, January 1986.
14. Peter Y. Kim. *F-16 Speaker-Independent Speech Recognition System Using Cockpit Commands (70 Words).* Master's thesis, AIR FORCE INSTITUTE OF TECHNOLOGY, 1988.

15. Dennis H. Klatt. *The Problem of Variability In Speech Recognition In Models of Speech Perception*, chapter 14, pages 300–324. Lawrence Erlbaum Associates, 1986.

16. Teuvo Kohonen. An Introduction to Neural Computing. *Neural Networks*, 1:3–16, 1988.

17. Teuvo Kohonen. Learning Vector Quantization and the K-Means Algorithm. In *1988 International Conference on Neural Networks, Tutorial #10, Self-Organizing Feature Maps, Appendix 4: 1-2*, IEEE Press, San Diego, 1988.

18. Teuvo Kohonen. Self-Organizing Feature Maps. In *1988 International Conference on Neural Networks, Tutorial #10, Self-Organizing Feature Maps*, IEEE Press, San Diego, 1988. Information obtained from videotape.

19. Teuvo Kohonen. Self-Organizing Feature Maps. In *1988 International Conference on Neural Networks, Tutorial #10, Self-Organizing Feature Maps*, page 17, IEEE Press, San Diego, 1988.

20. Richard P. Lippmann. An Introduction to Computing with Neural Nets. *IEEE ASSP MAGAZINE*, 4(2):4–18, April 1987.

21. Richard P. Lippmann. Review of Neural Networks for Speech Recognition. Unpublished article sponsored by the Department of the Air Force.

22. Mark K. Lutey. *Problem Specific Applications for Neural Networks.* Master's thesis, AIR FORCE INSTITUTE OF TECHNOLOGY, 1988.

23. Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, 1969.

24. Hermann Ney. The Use of a One-Stage Dynamic Programming Algorithm for Continuous Word Recognition. *IEEE Transactions of Acoustics, Speech, and Signal Ppreessing*, ASSP-32(2):263–271, April 1984.

25. Thomas W. Parsons. *Voice and Speech Processing.* McGraw Hill Book Company, 1987.

26. S. Pruzansky. Pattern-Matching Procedure for Automatic Talker Recognition. *JASA*, 35(3):354–358, March 1963.

27. L. R. Rabiner and B. H. Juang. An Introduction to Hidden Markov Models. *IEEE ASSP MAGAZINE*, 3(1):4–16, Janurary 1986.

28. Lawrence R. Rabiner and Schafer Ronald W. *Digital Processing of Speech Signals.* Prentice Hall Inc., 1978.

29. R. Rosenblatt. *Principles of Neurodynamics.* Spartan Books, New York, 1959.

Vita

Captain Wayne Recla

He entered the Air Force soon after graduation from Boys' Technical High School

His first tour of duty was in Southeast Asia as a radar technician on F-4D aircraft. Capt. Recla progressed through the enlisted ranks and attained the rank of Master Sergeant in 1981 while enrolled in the Airman's Education and Commissioning Program. In 1984, he completed requirements for a Bachelor of Science degree in Electrical Engineering at the University of New Mexico. Capt. Recla then served four years with the 6545th Test Group at Hill AFB, Utah testing unmanned air vehicles before entering the School of Engineering, Air Force Institute of Technology in June 1988.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT APPROVED FOR PUBLIC DISTRIBUTION: DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/89D-41	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 (AU) WRIGHT-PATTERSON AFB, OHIO 45433-6583	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)
A STUDY IN SPEECH RECOGNITION USING A KOHONEN NEURAL NETWORK DYNAMIC PROGRAMMING AND FEATURE-FUSION (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
RECLA, WAYNE, CAPT, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS THESIS	FROM _____ TO _____	1989 DECEMBER	187

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	KOHONEN NEURAL NETWORKS, SPEECH RECOGNITION DYNAMIC PROGRAMMING, NEURAL NETWORKS
25	4		

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

THESIS ADVISOR: DR. MATTHEW KABRISKY
PROFESSOR, ELECTRICAL AND COMPUTER ENGINEERING

(SEE REVERSE)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL DR. MATTHEW KABRISKY, GS-15	22b. TELEPHONE (Include Area Code) 513-255-9267 22c. OFFICE SYMBOL AFIT/ENG

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

Continued from Block 19: Abstract

Usually speech systems are one-dimensional. One feature or representation of speech is processed using a specific design methodology. In contrast, the human perception system is multi-dimensional; humans process more than just the sound of the word. Any speech recognition system that mimics human speech perception will need to be multi-dimensional. This methodology formed the basis for the design approach used in this research effort. Linear Predictive Coefficients (LPC) and formants were used as distinct and independent inputs into a recognition system consisting of a Kohonen neural network and a dynamic programming word classifier. A feature-fusion section and rule-based system were used to integrate the two input feature sets into one output result.

The first half of this research effort involved extensive testing of the Kohonen network. Using a speech input signal, different Kohonen gain reduction methods, initial gain values, and conscience values were tested for various iteration times in an effort to quantify the response and capabilities of the Kohonen network. Three-dimensional Kohonen-Dynamic Programming surfaces were developed that graphically showed the effects of gain, conscience, and iteration time on the speech recognition response of a Kohonen neural network. A new standard iteration time called a *multiple* was used during training of the Kohonen networks.

The results of the basic research on the Kohonen network produced an optimized Kohonen configuration that was used in the multiple-feature recognition system. A 70-word vocabulary of F-16 cockpit commands were used to evaluate the new feature-fusion method. The feature-fusion section performed well and was able to correctly classify words even when each individual input feature gave an incorrect answer