A RADIO TRANSMITTER FINGERPRINTING SYSTEM

BY
JASON PAUL TOONSTRA

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

Thesis Advisor: W. Kinsner, Ph. D., P. Eng.

(x+96+A72)=178 pp.

0-612-23529-7

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

A RADIO TRANSMITTER FINGERPRINTING SYSTEM

BY

JASON PAUL TOONSTRA

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis is concerned with an approach to record, process, and classify radio transmitter transients. A radio transmitter transient is emitted by the transmitter when the transmitter's push-to-talk button is depressed. This action engages the transmitter's frequency synthesizer which generates the carrier frequency. The generation of the carrier frequency by the frequency synthesizer is not instantaneous, thus a transient behaviour is exhibited during the carrier frequency acquisition. The capturing of such transient events is achieved by recording the discriminator output of an ICOM R7100 communication receiver. The recording is performed by a Sound Blaster sound card at a sampling rate of 44,100 samples per second and 16 bits per sample accuracy. The recording contains a noise component followed by a transient. The transient is separated from noise by a variance fractal dimension trajectory analysis. Once the transient has been localized, multiresolution wavelet analysis and genetic algorithms select the critical features of the transient used to classify the transient. Multiresolution analysis provides a set of wavelet coefficients that represents the transient features independently, allowing the genetic algorithm to select those that are most critical. The features are classified by a multilayer neural network with 64 inputs and 12 hidden neurons. The average classification rate achieved is 96% for an experimental set of 6 transmitters consisting of four Kenwood transmitters and two Yaesu transmitters. For this experimental transmitter set, the results show that this system classifies transients generated by transmitters of the same manufacturer and model, as well as transients generated by transmitters from different manufacturers.

ACKNOWLEDGMENTS

First, I would like to thank Dr. Kinsner, who as my advisor has provided guidance and support throughout the research. Dr. Kinsner has introduced me to all the topics developed in this research, including wavelets, fractal analysis, and neural networks. I would also like to thank him for generously supplying a computer to develop and test the software written for this thesis.

I would like to thank Mr. Sherman Chow, Mr. Robyn Jackmann, Dr. Nur Serinken and Mr. Kerry Ellis of the *Communication Research Centre* (*CRC*) for providing us with the modified ICOM R7100 communications receiver along with all the transients used to test the system. I would also like to acknowledge the financial support provided by the *Communication Research Centre* during a part of this work.

I would also like to thank my friends and colleagues, namely; Armein Langi, Ken Ferens, Richard Danserau, Hongjin Chen, Don Shaw, Randy Allen, Warren Grieder, Eric Jang, Pradeepa Yahampath, Rasekh Rifaat, and Fan Mo for useful discussions.

Finally, I would like to thank my parents, Paul and Maureen Toonstra who have been a source of constant support and have always been there for me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Background and Motivation

As with humans, it is claimed that low power radio transmitters have unique fingerprints. The collection and classification of radio transmitter fingerprints are important for ensuring the appropriate use of the radio frequency spectrum. If an operator is using a particular frequency inappropriately, the system developed for this thesis could identify the operator's transmitter by its transient fingerprint and the necessary action to restore proper use of the frequency can be taken. The fingerprints take the form of unique features contained in the transient generated by the radio transmitter as it attempts to lock upon the selected carrier frequency. This uniqueness is due to the differing designs and tolerances of components used in the construction of the transmitters. The signal to be communicated by the transmitter is modulated upon the carrier frequency, and thus a transient is generated each time the transmitter is engaged. To engage a transmitter, the user usually has to push a button called the push-to-talk button that informs the transmitter a transmission is to occur. Once the transmitter is aware of the fact that a transmission is to occur, its phase-locked loop (PLL) begins to generate the carrier frequency and in the process, a transient is generated and transmitted. To generate the carrier, a voltage controlled oscillator (VCO), a phase comparator, and a lowpass filter are used, forming the phase-locked loop. Figure 1.1 shows the configuration of the components to construct a PLL. The reference carrier signal frequency is compared to the current output of the VCO. If the frequency generated by the

VCO is greater than the desired carrier frequency, the phase comparator generates a negative voltage. If the generated frequency is less than the carrier, the phase comparator feeds the VCO a positive voltage. If the VCO sees a negative voltage from the phase comparator it drops the frequency it generates, while if a positive voltage is detected, the VCO increases the frequency it is generating. If the frequency generated by the VCO matches the carrier, the phase detector's output is zero volts, and thus the desired carrier has been reached.

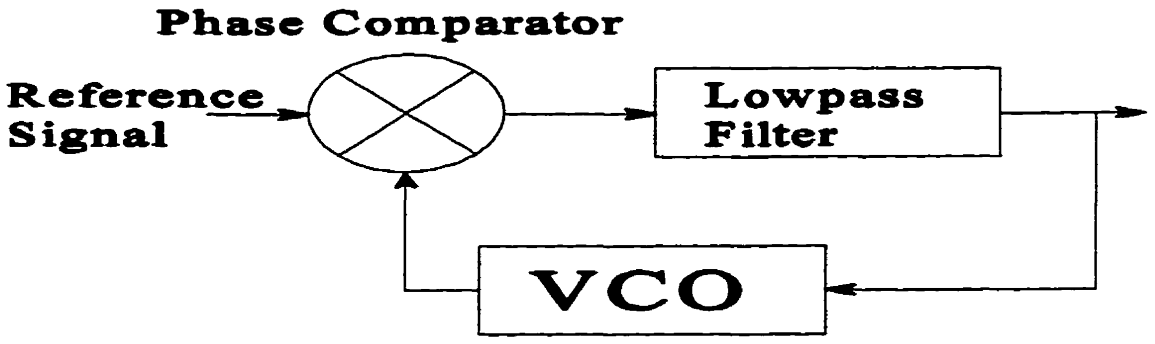


Fig. 1.1. A phase-locked loop.

The PLL described above is not quite what is needed to generate the carrier since the carrier is already required as a reference for the phase comparator. By implementing two frequency dividers as shown in Fig. 1.2, a frequency synthesizer [Couc93] is constructed. This allows a variety of carrier frequencies by adjusting M and N. The output frequency is described by

$$f_o=(\frac{M}{N})f_r \tag{1.1}$$

where f_r is the reference frequency. By implementing frequency dividers, the reference frequency does not need to be the carrier frequency.

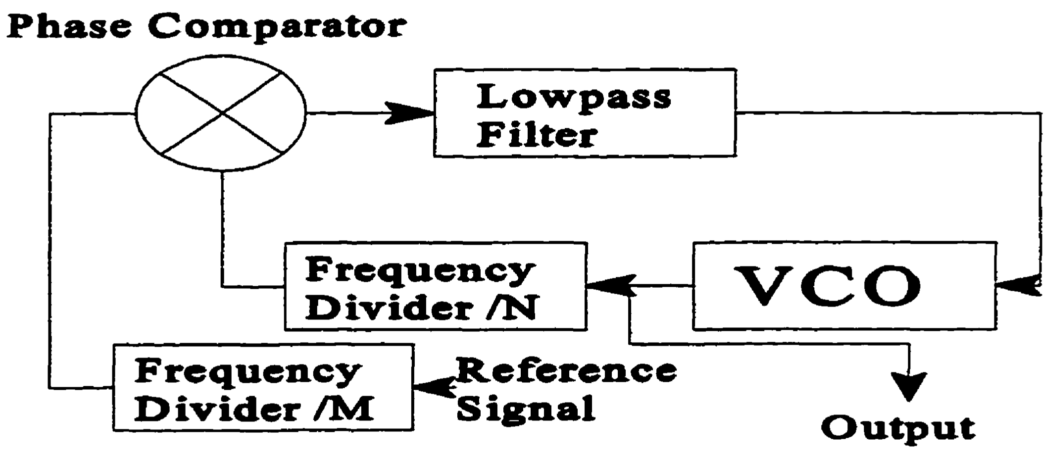


Fig. 1.2. A frequency synthesizer.

In order to analyze and classify the transients, a means of recording the transients is required. This thesis addresses the collection issue by developing a system that is based upon *Creative Lab's* Sound Blaster. The transients are recorded with a sampling rate of 44,100 samples per second and 16 bits per sample accuracy. The AM bandwidth of the receiver is 6 kHz which requires a minimum sampling rate of 12,000 samples per second by Nyquist's Theorem. Thus, with a sampling rate of 44,100 samples per second, it is ensured that all the transient's frequency components are recorded. The high sampling frequency (over 7 times the speech bandwidth) is required to accommodate larger bandwidth receivers, as well as provide high quality recordings that could stand in a court of law. The high frequency sampling eliminates the need for separate analog recordings of the transient.

Once the transients have been collected, the significant features of the transients are selected using wavelet analysis and genetic algorithms. The selected features are used to determine which transmitter is responsible for the transient generation. The classification is performed by a neural network trained on transients previously collected. We shall show that the neural network is able to distinguish between transients generated by transmitters built by different manufacturers as well as transmitters built by the same manufacturer. Beyond distinguishing transmitters of differing manufacturers, we shall also show that the neural network is able to correctly classify transients generated by transmitters of the same model type. This level of classification requires that transients generated by transmitters of the same model type be unique, just as each person has their own fingerprint. This thesis will show that given a set of transients, this level of classification is possible.

MoTron Electronics has developed and manufactured the TXiD transmitter fingerprinting system. However, the TXiD has a maximum sampling rate of 10,000 samples per second, requiring a separate stereo cassette recorder to record the transients that are to be used in a court of law. Classification by the TXiD system involves visual comparison of the features between transients to determine which class the transient belongs to. The system developed by this thesis uses feature selection and neural networks to objectively classify the transients that eliminates the visual comparison of the transient features.

This concept of transmitter transient analysis and implementation has been proposed by Kinsner in 12 internal reports, starting from 1993, and different parts of the system have been studied by his students [Ande95], [Diet94], [Khan95], [Kwok95], [Ruda94], {Shaw94], [Toon95]. There is also another implementation of the system being completed in his research group.

1.2 Thesis Overview

This thesis is organized in a fashion that represents the order in which each technique is applied to analyze the transients. Chapters 2 through 6 end with a discussion of the computer implementation of the technique described in the chapter. The computer code for each component written in the *C* computer language is contained in Appendix A.

Chapter 2 provides a discussion on how the transients are recorded using the Sound Blaster. The concepts of how the acquisition system works is described along with the Sound Blaster recording initialization procedure.

Chapter 3 describes how the noise is separated from the transient using fractal analysis. The separation is required since the acquisition system writes a noise component as well as the transient component to the acquisition output file.

Chapter 4 describes the wavelet analysis technique called multiresolutional signal decomposition which generates a set of wavelet coefficients that represent the independent features of the transient.

A genetic algorithm is implemented to select the critical features represented by the wavelet coefficients. Chapter 5 covers the implementation and discusses the components of the genetic algorithm. The selected wavelet coefficients are used to classify the transient.

The classification of the transient is performed by a neural network. Chapter 6 describes the neural network and the training procedure used in this study.

Chapter 7 covers the results obtained in the experimental portion of this thesis. The effects of the number of features selected and the classification accuracy is studied in this chapter.

Finally, Chapter 8 completes the thesis with conclusions, recommendations and contributions.

CHAPTER 2
TRANSIENT ACQUISITION

2.1 Introduction

In order to analyze radio transmitter transients, a means of recording the transient behaviour is required. For such a system to be useful, it must have a high sampling rate and should have 16 bit resolution. A good candidate for recording the transients is the Sound Blaster 16 audio board developed by Creative Labs. The maximum sampling rate of the Sound Blaster is 44,100 samples per second and contains a 16 bit analog to digital converter. However, a software driver for detecting and capturing transients is not known to exist. This chapter will discuss such a driver based on the software developed by Ethan Brodsky [Brod95] for 16 bit stereo recording. Two key modifications need to be added to enable efficient and accurate recording of radio transmitter transients. These modifications include implementation of a circular buffer [Kins87] to store the recorded samples and a means of detecting when a transient has occurred. These modifications require that the original recording procedure be completely removed and replaced with the new transient recording procedure. The result is a high speed and accurate means of recording radio transmitter transients.

2.2 Receiver Requirements

To record the transmitter transients for legal purposes, the output of a receiver's discriminator is usually recorded on an analog audio or video tape. Our system eliminates the analog taping by providing a high-quality digital recording. The signal from the discriminator required for the analog or digital recording is nonstandard, requiring a modification to the receiver. The receiver used in the acquisition of all the transients used in this study was an ICOM R7100 wideband receiver. The discriminator output access point is pin 1 on a 15 pin connector, located at the back of the receiver. The discriminator output is a voltage proportional to the frequency deviation from the carrier frequency. The ICOM R7100 discriminator characteristics were measured by Robyn Jackmann and Kerry Ellis of the *Communication Research Center* in Ottawa, Ontario. A frequency generator was used to generate a carrier frequency of 150 MHz. The voltage at the output of the discriminator was measured using a Fluke 23 digital voltage meter. The frequency was swept from 137 to 165 MHz with the voltage at the discriminator's output measured at appropriate intervals. Figure 2.1 shows the plot obtained with the x axis labels being offsets relative to 150 MHz.

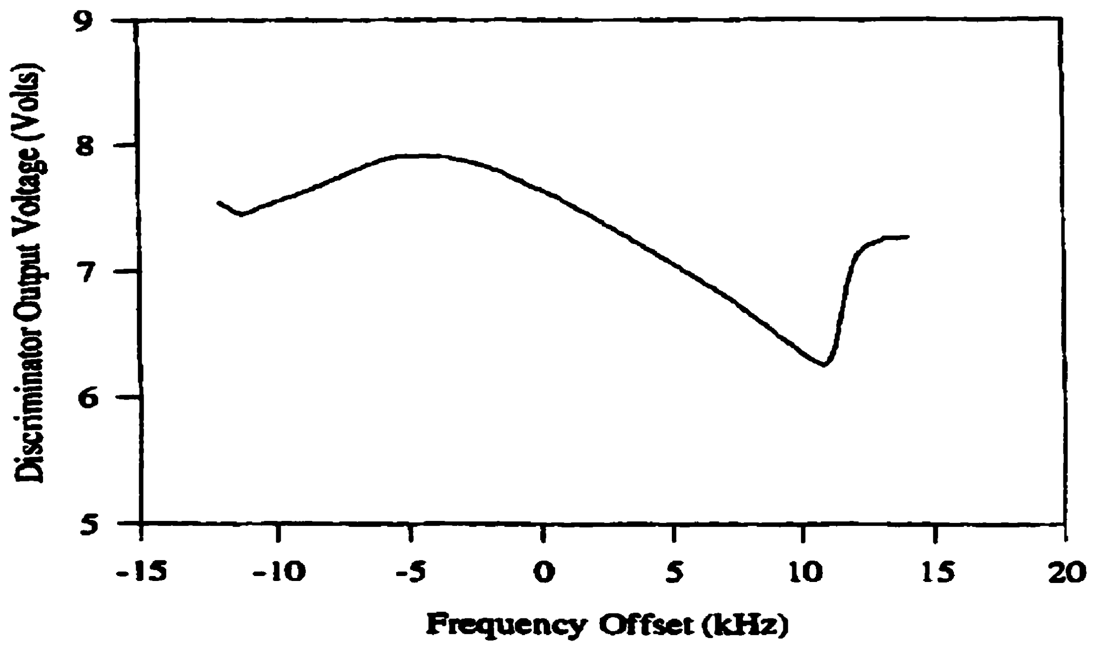


Fig. 2.1. Discriminator output voltage vs. frequency offset.

It is noted that the voltage fluctuation from peak to peak is about 1.75 volts which is within the Sound Blaster's input sensitivity of 2 volts peak to peak. Deviations exceeding the above range are not useful since they are outside the receiver's passband filter. A squelch level control and speaker output jack used to detect when a transient has occurred is also required by the acquisition system. The ICOM R7100 has such facilities making it a good candidate for use in recording radio transmitter transients.

2.3 Overview of Recording Technique

Since a transient is a short duration event (approximately 10 to 50 ms), all the information recorded before the transient need not be stored. However, the Sound Blaster needs to be set up and the recording must start before the transient event occurs. This is required because initialization of the Sound Blaster takes a few seconds, and the transient would be missed due to the time requirements of the setup. Thus, to limit the amount of memory used to record this short duration event and have the Sound Blaster continually recording, a circular buffer is implemented [Kins87]. To implement a circular buffer, a block of memory is allocated, and data is written to this block. When the end of the block is met, the pointer indicating the next location to be written to is repositioned at the beginning of the buffer. The data currently in the buffer location is overwritten by the new data.

Figure 2.2 pictorially shows the operation of a circular buffer. However, to ensure that the transient is not overwritten, a means of detecting the transient needs to be implemented. A good approach is to monitor the speaker output of the receiver with the squelch level set to suppress the noise. When a transmitter engages in a communication, the receiver's squelch circuit opens and permits the signal to pass to the receiver's output to be heard by the user. Thus, by instructing the Sound Blaster to discontinue recording after a

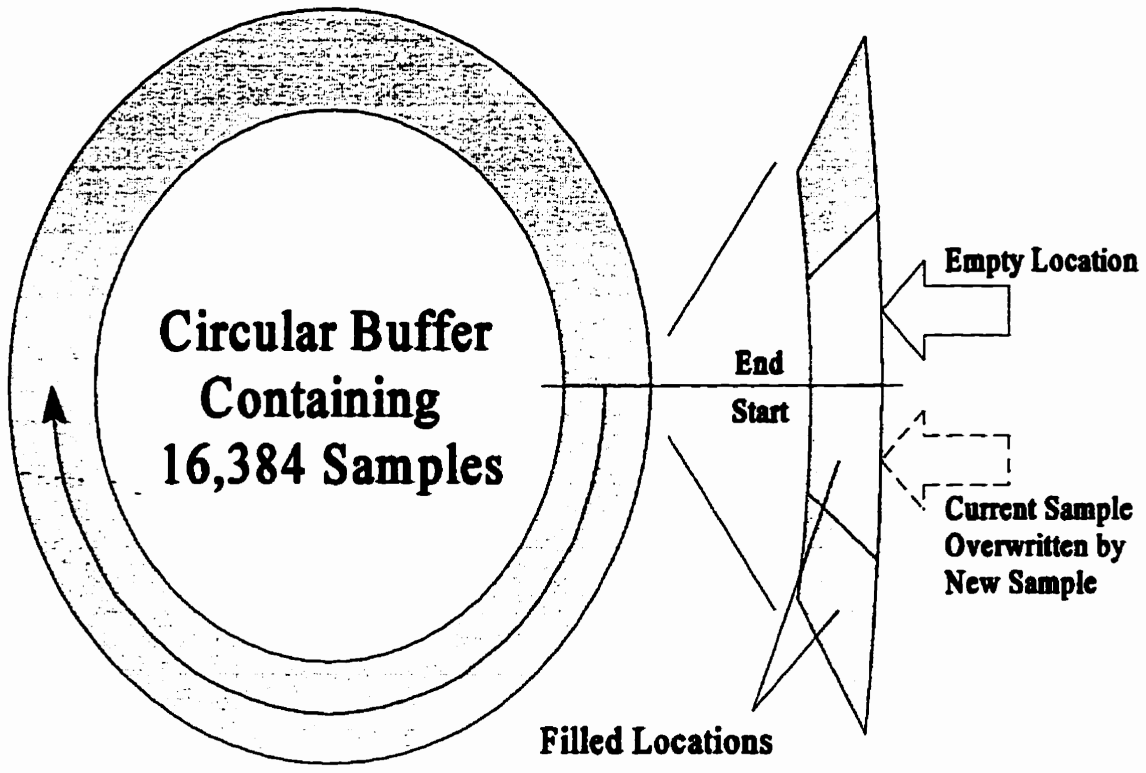

Circular Buffer Containing 16,384 Samples
Empty Location
End
Start
Current Sample Overwritten by New Sample
Filled Locations

Fig. 2.2. Operation of circular buffer.

certain number of samples once the receiver's output becomes active, prevents the transient from being overwritten.

Figure 2.3 shows how this is achieved. Once the transient has be detected, the Sound Blaster shuts down and the transient is written to a file. A more detailed description of the recording process is provided next.

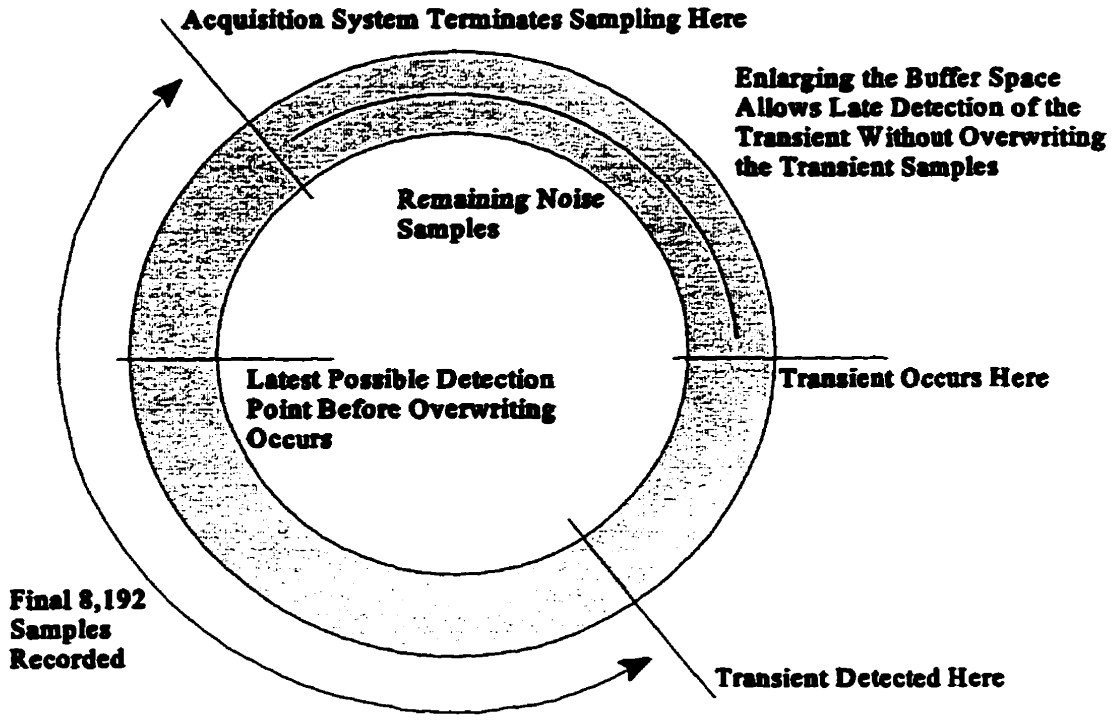

Acquisition System Terminates Sampling Here
Enlarging the Buffer Space Allows Late Detection of the Transient Without Overwriting the Transient Samples
Remaining Noise Samples
Latest Possible Detection Point Before Overwriting Occurs
Transient Occurs Here
Final 8,192 Samples Recorded
Transient Detected Here

Fig. 2.3. Transient detection and its effect on recording.

2.4 Recording Details

With a preliminary understanding of how the acquisition is to occur, the details can be described. The Sound Blaster allows for two channels to be recorded simultaneously at 44,100 samples per second with 16 bit accuracy. We have selected the left channel to record the receiver's discriminator output, and the right channel to record the receiver's speaker output. Both the left and the right channels are written to a temporary buffer where the signals are analyzed. The discriminator samples are written to the circular buffer, while the speaker output samples are analyzed for activity. If the speaker output samples deviate from their default values of zero, a transient event is said to have occurred and the search is terminated.

The circular buffer is 16,384 samples in size which translates into 32,768 bytes of memory. The size of the transients written to the output file is 8,192 samples, allowing a maximum of 8,192 samples of noise to be recorded. Including a large number of samples of noise allows for a delay in the detecting activity on the speaker output. The squelch circuit was found to a have a delay exceeding the transient duration, thus requiring extra samples to be included in the buffer to ensure the transient is not over written as shown in Fig. 2.3. Including the extra samples allows for the squelch circuit's delayed reaction to the incoming communication. However, the squelch circuit must react within 185 milliseconds of the recording of the transient's first sample. The ICOM R7100's squelch circuit does react within this time constraint ensuring accurate transient acquisition. Once the noise and the transient have been acquired, they are written to a file in unsigned integer format ranging from 0 to 65,536 and with the value 32,768, denoting the signal's D.C. voltage. This file is passed on to the noise separation procedure to extract the transient for analysis and classification.

The connectivity between the Sound Blaster and the receiver consists of a 15 pin connector, a mono 1/8 inch jack and a 1/8 inch stereo jack. Pin 1 on the 15 pin connector is connected to the left channel connection on the stereo jack while the right channel connection is connected to the mono jack. The stereo jack is inserted into the Sound Blaster's line-in input while the mono jack is inserted in the speaker output and the 15 pin connector is connected to the receiver's disciminator output access point. Figure 2.4 shows the correct connection configuration.

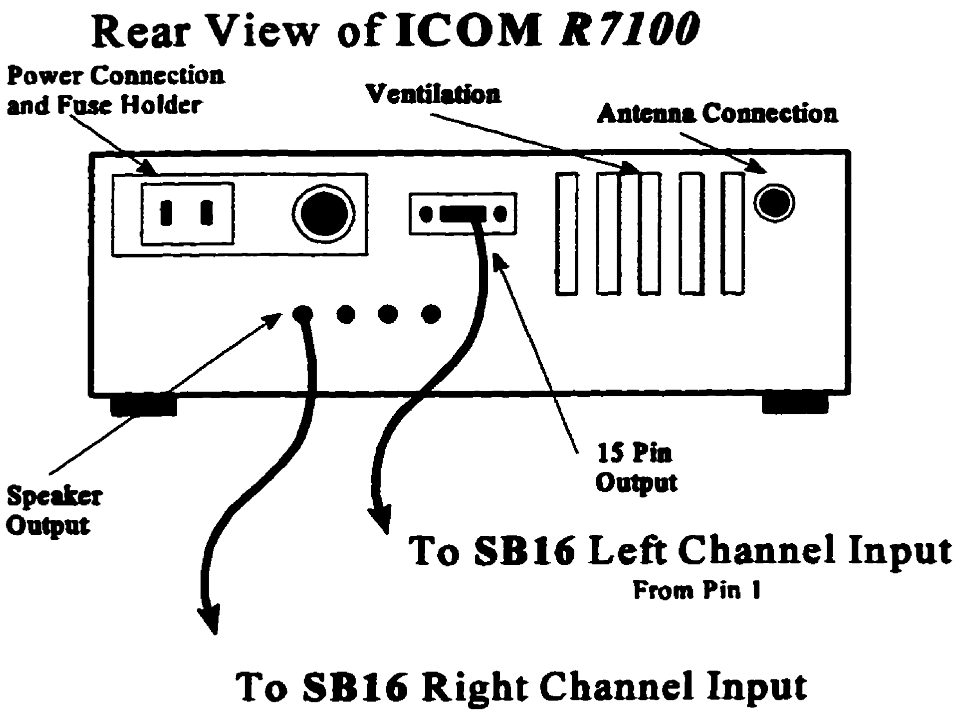


Fig. 2.4. Connections to receiver.

2.5 Sound Blaster Programming

The first step in recording a source using the Sound Blaster is initialization. To begin, the correct installation of the Sound Blaster and base I/O address selection must be verified by resetting the Sound Blaster's digital signal processor (DSP). The reset port is offset by the base I/O address whose default value is 0x220h. To reset the Sound Blaster, the reset port, offset by the base I/O address is written a reset sequence. If the DSP does not respond within 100 microseconds after being reset, the Sound Blaster could be installed incorrectly. However, if the DSP does respond, the initialization can continue by passing the interrupt request number (IRQ) and the direct memory access (DMA) controller channel number to the initialization subroutine for use in setting up the Sound Blaster for recording. The default setting for the IRQ is 7 and the DMA channel for 16 bit recordings is set to 5. The IRQ determines which interrupt service routine (ISR) is called when the Sound Blaster issues an interrupt. The DMA controls the transfers of large amounts of data between the I/O devices and memory by bypassing the CPU [Kins87]. With these parameters in hand, an interrupt service routine can be installed and the DMA can be programmed for the transfer of data from the Sound Blaster to memory. The interrupt service routine contains instructions on how to handle the incoming data and controls the remaining number of samples to be recorded. The ISR also is responsible for the detection of a transient using the data recorded on the right channel containing the receiver's speaker output. To install an ISR, all interrupts are temporarily disabled with the exception of the non-maskable interrupts (NMI). To change the pointer to the ISR to be used for acquisition system, the priority interrupt controller (PIC) [Kins87] is accessed. A mask is constructed from the IRQ defined above to disable the current interrupt that is to call the new acquisition ISR. Once the interrupt is disabled, a copy of the current ISR pointer is made for reinstallation when the acquisition ISR is no longer needed. Next, the pointer to the acquistion ISR is moved into the location previously held by the old ISR and a new mask is constructed to enable the interrupt corresponding to the acquisition ISR.

With the interrupt service routine in place, the DMA controller is programmed to transfer the recorded signals from the Sound Blaster to a memory location. The type of DMA transfer used by the Sound Blaster is the single byte transfer mode. A first in-first out (FIFO) buffer is used by the Sound Blaster to store the recorded samples in the event the DMA transfer is delayed [Brod95]. The first requirement in the programming of the DMA controller is allocating memory for the transfer that does not cross the 64 Kbyte physical page boundary. The size of this buffer that the DMA will write to is 256 samples or 512 bytes, since each sample requires two bytes. Once this memory has been allocated, the DMA channel selected for use by the Sound Blaster is disabled by writing a mask constructed from the channel number to the DMA's mask port. The next step in the programming is the resetting of the DMA's byte pointer by writing any value to the DMA's byte pointer port. Resetting the byte pointer clears the way for setting the transfer mode to be implemented by the DMA controller. The transfer mode selected is the auto-initialized recording mode which is written to the DMA's mode port. This mode allows the DMA controller to use the transfer memory as a circular buffer. If the transfer mode is not auto-initialized, samples will be lost due to the fact that the controller ends the transfer once the end of the transfer memory is encountered. The ISR can start the next transfer. However, there will be numerous samples missed between the last sample transferred and the next sample to be transferred which will compromise the quality of the recording. By implementing auto-initialized transfers, the DMA when it encounters the end of the transfer memory, continues the transfer by writing the samples to the beginning of the transfer memory. Thus, the DMA when in auto-initialized mode does not stop at the end of the transfer and does not require restarting by the ISR. The use of auto-initialized transfers requires that the two interrupts be issued by the Sound Blaster for each DMA transfer. One interrupt occurs at the midpoint of the transfer and the other at the end of the transfer. This allows the ISR to recover the samples in one half of the transfer memory while the DMA controller is transferring data into the other. If only one interrupt were issued by the Sound Blaster at the end of the transfer, the samples would be lost, since the DMA controller treats the transfer memory as a circular buffer and begins writing to the beginning of the block once the end of the block is encountered. Figure 2.5 shows when the interrupts are issued and the responsibilities of the ISR. The next step in the programming is informing the DMA controller where the offset from the page boundary, the first location of the transfer buffer resides.

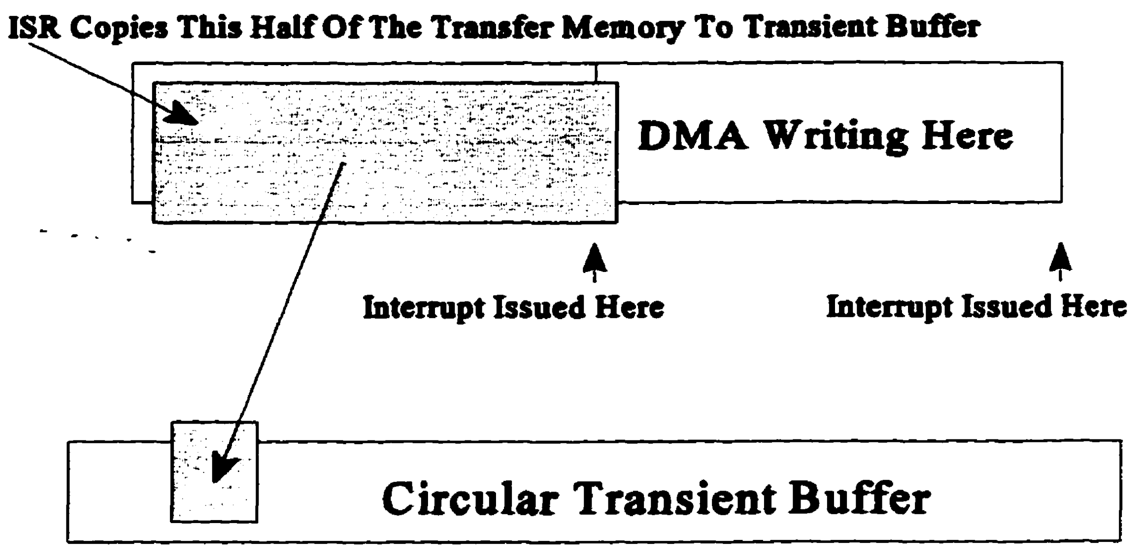

ISR Copies This Half Of The Transfer Memory To Transient Buffer
DMA Writing Here
Interrupt Issued Here
Interrupt Issued Here
Circular Transient Buffer

Fig. 2.5. Issuing of interrupts and ISR responsibilities.

The computer's memory is mapped into pages that reduces the effects of memory fragmentation [SiPG92]. The page number and page offset are combined to determine the physical address of a memory location. The offset is measured in words, thus requiring the low byte to be written first, followed by the high byte. Once the offset has been written to the controller, the transfer length is communicated to the controller by writing to the DMA's count port. The low byte of the transfer length is written first, followed by the high byte as was done for the transfer memory offset. Since the transfer memory is 256 samples in size, 255 in hexadecimal is written to the low byte and 0 is written to the high byte. The hexadecimal equivalent of 255 is used since the transfer length is measured in the number of samples, minus one. The next programming step is the programming of the transfer memory's page location. Again, as before, the low byte is written first, followed by the high byte into the DMA's page port. Once the transfer memory's page has been written to the DMA, the DMA programming is complete and the channel is enabled by clearing the mask bit assigned to the channel.

With the DMA controller programmed, the sampling rate to be used by the Sound Blaster can be set along with the play or record mode selection. These selections are written to the Sound Blaster's DSP. The sampling rate is set to 44,100 samples per second by writing the high byte of the sampling rate followed by the low byte. The record mode is selected as being an unsigned 16 bit stereo which is encoded into a single byte and written to the DSP. The final parameter to be passed to the DSP is the number of samples to be collected before an interrupt is called. As required by the auto-initialized DMA transfers, the number of samples required before an interrupt is issued is one- half the number held by the transfer memory. The number of samples between interrupts is written to the DSP with the low byte first, followed by the high byte. Once the high byte is written, the recording begins.

2.6 The Acquisition Interrupt Service Routine

The acquisition interrupt service routine is responsible for copying the contents of the transfer memory to the transient buffer and detecting whether a transient has occurred. Once the ISR has been called, a determination of which half of the transfer memory is to be operated on is made, since the DMA's auto-initialized transfer mode is implemented. After the determination is made, the ISR looks at the right channel samples to determine whether or not there is any activity at the speaker output. If activity is detected, a variable is set to indicate that the final 8,192 samples should be collected, and the Sound Blaster be shut down. Whether or not activity has been detected, the samples recorded from the output of the receiver's discriminator are written to the transient buffer. A buffer pointer points to the current location in which to write the samples being copied from the transfer memory. As was mentioned previously, when the end of the buffer is encountered, the pointer is repositioned at the beginning of the buffer and the previous samples are overwritten. If a transient has been detected, the number of samples that remain to be recorded, 8,192 is decremented by the number of samples just written to the transient buffer. The next time the ISR is entered, the search for activity on the right channel is bypassed since the transient has already been detected and the number of samples remaining further decremented. If the decrement has left the number of samples remaining equal to zero, the ISR issues a command to the Sound Blaster to discontinue collecting samples. If a transient was not detected, the number of samples remaining is left unchanged and the Sound Blaster continues its recording. The final responsibility of the acquisition ISR is to acknowledge the interrupt to both the interrupt controller and the Sound Blaster. Once the interrupts have been acknowledged, the ISR is exited until the next interrupt is issued.

2.7 Sound Blaster Shut Down

The first step in the shut down of the Sound Blaster is the final acknowledgment of any hanging interrupts. Once the acknowledgment has been completed, the DMA channel is disabled by writing a mask to the DMA's mask port. This terminates the ongoing DMA transfers and releases the channel for other devices to use. Finally, the old interrupt is reinstalled, replacing the acquisition interrupt by the same method described for replacing the old interrupt. With the shutdown complete, the file supplied by the user for writing the transient to is opened and the transient buffer contents are written to the file. Within the file the transient along with the noise occurring prior to the transient is stored.

2.8 Summary

This chapter described a system for recording radio transmitter transients using the Sound Blaster. The sampling rate achieved by the system is 44,100 samples per second with 16 bits per sample accuracy. The initialization of the Sound Blaster and the interrupt service routine implemented to record the transient were described. A file consisting of 16,384 samples is written containing the transient and the noise occurring prior to the transient, once the recording by the Sound Blaster is complete.

CHAPTER 3
NOISE SEPARATION

3.1 Introduction

Once the transient has been captured using the acquisition system, the transient and noise need to be separated. Separation is required since the noise portion of the signal does not contain unique information that can be used in classifying the transient. In order to separate the noise from the transient, the variance fractal dimension [Kins95] is implemented. The use of fractality allows for the characterization of the complexity of the noise-transient signal. To measure the fractality of an object, the concept of fractal dimensions is employed. A fractal dimension characterizes the self similarity and the irregularity of an object in question. If the object is highly regular, non-fractal, the fractal dimension equals its topological dimension. For example, if the object is a line, its fractal dimension is 1 and if the object is a surface, its fractal dimension is two. However, as the object becomes more irregular, the fractal dimension increases passed its topological dimension to some non-integer value. These objects with non-integer values can be characterized as being fractal. The fractal dimension for an object has a limit; the dimension of an object cannot exceed the object's embedding dimension equal to the topological dimension plus one. The topological dimension (or Euclidean dimension) is the integer dimension of natural objects such as a surface. Therefore, a fractal surface cannot exceed its embedding dimension of three. In total there are over 19 fractal dimensions that are described in a unified framework by Kinsner [Kins94]. These fractal dimensions can be characterized as being morphological, entropy, variance, or spectral. Of these dimensions, the variance dimension is an approach that produces reasonable results. However, since it is based on heuristics, it will require additional work to establish the bounds for optimum separation of the transient from the noise.

3.2 Variance Fractal Dimension

The variance dimension calculates the spread of signal amplitudes between two samples separated by some increment. If $B(t)$ denotes the samples of a signal, the variance between samples can be expressed by

$$Var[B(t_2)-B(t_1)] \tag{3.1}$$

Let us assume that the following power law relationship holds

$$Var[B(t_2)-B(t_1)]\sim|t_2-t_1|^{2H} \tag{3.2}$$

where the parameter H, is the Hurst exponent. In general, the power law relationship between variables is of the form

$$t=ch^{d} \tag{3.3}$$

where d is an exponent of h and c is a constant. If the log of each side of Eq. 3.3 is taken, the following is the result

$$\log(t)=\log(c)+d\log(h) \tag{3.4}$$

Thus, to determine d, the slope of the line generated by drawing t on a log-log plot is found.

For notational convenience, set

$$\Delta t = |t_2 - t_1| \tag{3.5}$$

and

$$(\Delta B)_{\Delta t} = B(t_2) - B(t_1) \tag{3.6}$$

Using the same procedure used in 3.4, and equating Eqs. 3.1 and 3.2, the following is found

$$H = \lim_{\Delta t \to 0} \frac{1}{2} \frac{\log[Var(\Delta B)_{\Delta t}]}{\log(\Delta t)} \tag{3.7}$$

where *d* is equal to *2H*, *h* is equal to Eq. 3.5 and *t* represents Eq. 3.6. The Hurst exponent's range is between zero and one. When the Hurst exponent is closer to 1, the signal is smooth without a lot of detail. However, when the Hurst exponent is closer to 0, the signal is very coarse with lots of detail. To compute the variance dimension, the following formula is used

$$D_\sigma = E + 1 - H \tag{3.8}$$

where H is the Hurst exponent and E is the Euclidean dimension. For the noise separation, E is equal to 1. Therefore, to find the variance dimension, the slope found by Eq. 3.7 is computed and used in dimension calculation described by Eq. 3.8.

3.3 Computation of the Hurst Exponent

The computation of the Hurst exponent can be efficiently calculated using the above described power law. A real time analysis extension has also been developed by Kinsner [Kins94] however, the requirement for real time analysis by this system is not an issue. The first step in the analysis is determination of several important parameters. The first of these parameters is the number of samples to be analyzed. Since the signals being analyzed are discrete, the number of samples is finite and defined as N_T. The next parameter to be found is the maximum number of intervals used to determine the Hurst exponent. The maximum number of intervals defined by a b-adyic sequence (b^0, b^1, b^2,...) is denoted by K_{max} and is found by

$$K_{\max}=floor\left(\frac{\log(N_T)}{\log(b)}\right) \tag{3.9}$$

However, to ensure a good covering of the signal by the intervals, the size of the intervals should be reduced such that more than 30 intervals will fit within the signal. If this is achieved, the variance is statistically valid. To find the largest number of intervals under this criterion, the following is used

$$K_{hi}=K_{\max}-K_{buf} \tag{3.10}$$

where

$$b^{kbuf}\geq 30 \tag{3.11}$$

To select the value of b, the type of analysis must be known. If a high degree of detail about the fractality of a signal is required, the dyadic sequence is best suited. By selecting b equal to 2, the above process is called fractal amplification [Kins94]. However, we only require the fractal dimension to determine a point at which the noise ends and the signal begins. By selecting b equal to 1, those portions of the signal corresponding to noise will have a higher fractal dimension and the portions corresponding to the transient will have a lower fractal dimension. This phenomenon is due to the fact that by having the intervals with smaller values, the correlation between these samples is more significant than if the intervals were larger. Thus when the signal is highly uncorrelated, the dimension is higher than if the signal were more correlated. However, when b is equal to 1, the parameter calculation for K_{max} is indeterminate. To combat this situation, the parameter calculations should proceed with b equal to 2 and K_{hi} found using

$$K_{hi}=2^{(Kmax-Kbuf)} \tag{3.12}$$

The variance dimension operates in a loop from K_{hi} to K_{low} where the variance between samples is computed for varying intervals governed by the loop index, k. K_{low} is defined as an integer greater than one to avoid calculating the variance between adjacent samples. Once in the loop structure, two new parameters are computed with each decrement of the loop index. The first of these parameters is the size of intervals, n_k, where the subscript k is the loop index. The following is used to compute n_k

$$n_k=b^{\,k} \tag{3.13}$$

If b is set to 1, n_k is equal to k. Now that the interval size has been defined, the number of intervals required for complete or near complete coverage of the signal is found by dividing the number of signal samples by n_k. The number of intervals is denoted by N_k. A complete coverage is obtained if the interval size exactly divides the number of signal samples. A near complete coverage results if the interval size does not exactly divide the number of samples. If a near complete coverage occurs, the last interval is ignored. In the worst case, 8 samples are ignored for signals with samples sizes of 512 when b is equal to one. This represents a total of 1.56 percent of the samples being ignored. Thus, the error due to ignoring the last interval is negligible.

Once these loop parameters have been computed, the actual variance calculation can be performed. The variance is found using

$$Var(\Delta B)_k=\frac{1}{N_k-1}\left[\sum_{j=1}^{N_k}(\Delta B)_{jk}^2-\frac{1}{N_k}\left(\sum_{j=1}^{N_k}(\Delta B)_{jk}\right)^2\right] \tag{3.14}$$

Once the variance for n_k is calculated, the logarithms (logs) of the interval size and variance are calculated as

$$\begin{aligned} X_k&=\log(n_k) \\ Y_k&=\log(Var(\Delta B)_k) \end{aligned} \tag{3.15}$$

The logs are taken in accordance to equation 3.4 as required for the log-log plot to determine the power law. This completes the loop for a particular value of k. The value of k is decremented by b in order to determine the new loop parameters. Once the parameters have been found, the variance is calculated and the logs are taken. This procedure is continued until k reaches K_{low}.

To find the slope of the line found in the log-log plot, the method of least squares is employed. The formula for the computation of the method of least squares is

$$s=\frac{K\sum_{i=1}^{K}X_iY_i-\sum_{i=1}^{K}X_i\sum_{i=1}^{K}Y_i}{K\sum_{i=1}^{K}X_i^2-\left(\sum_{i=1}^{K}X_i\right)^2} \tag{3.16}$$

where X_i and Y_i are defined above in Eq 3.15. To obtain the Hurst exponent, the slope computed in Eq. 3.16 is divided by 2 since the power law's exponent is $2H$. With the Hurst exponent in hand, the variance dimension is found by

$$D_\sigma=2-H \tag{3.17}$$

for the signals operated upon in this study.

3.4 Implementation of the Variance Fractal Dimension Trajectory

Using the technique for computing the variance dimension above, a software implementation for noise separation can be developed. The variance dimension is well suited for discriminating between noise and signal since the noise component is very uncorrelated whereas the transient portion is highly correlated. However, to find this junction, the variance dimension cannot be applied to the input signal as a whole. In order to find the noise-transient location, the variance dimension must be measured within windows containing a piece of the signal consisting of both the transient and noise. When the window contains mostly noise, the variance dimension is high, near 2. If the window contains both the transient and noise, the variance dimension is usually between 1.3 and 1.8 and if the window contains only the transient, the variance dimension is below 1.3. It is important that the window is not too large nor too small. If the window is too large, the noise-transient transition point cannot be localized efficiently since the window could contain a small section of the noise and a large piece of the transient. The variance dimension will indicate that the transition point has been reached when in fact the window contains noise. If the window is too small, the statical validity of the variance calculations is brought into question. As was mentioned above in the previous section, the number of intervals covering a signal should exceed 30. This study utilizes a window size of 512 samples. This ensures that the variance calculations are valid by having all interval coverings consisting of 30 or more intervals. Using a window size of 512 also allows for accurate noise-transient transition localization. Another consideration for good transition localization is window overlap. If the windows do not overlap, the transition point may occur within one of the windows but the variance dimension will not indicate this unless the window size is very small. However, since the window size has been chosen to be 512, the transition localization will be very poor if the transition point occurs in the middle of a window. With this in mind, it is clear that the windows should heavily overlap one another. In fact, the variance dimension window is only incremented one sample to guarantee good transition localization.

With the window size and window increment size set, the noise separation utilizing the variance dimension can begin. The first step in the implementation is the reading in of the file generated by the acquisition system. Once the file has been read in, the size of the file is reduced from 16,384 samples to 4,096 samples. The reduction is achieved by averaging four adjacent samples. This averaging is justified by the fact that further averaging will take place once the transient has been localized. By reducing the size of the signal, the separation process is accelerated by having less samples to analyze. With 4,096 samples, the detail contained in the signal is not significantly affected and the transition point localization remains accurate. Figure 3.1 shows the contents of the circular buffer and the variance fractal dimension of the transient and noise components. It should be noted that the noise occurring after the transient in Fig. 3.1 is actually a part of the noise recorded prior to the onset of the transient. The location of the noise occurring after the transient is due to the fact that the circular buffer is written without realigning the noise to its proper position before the transient.

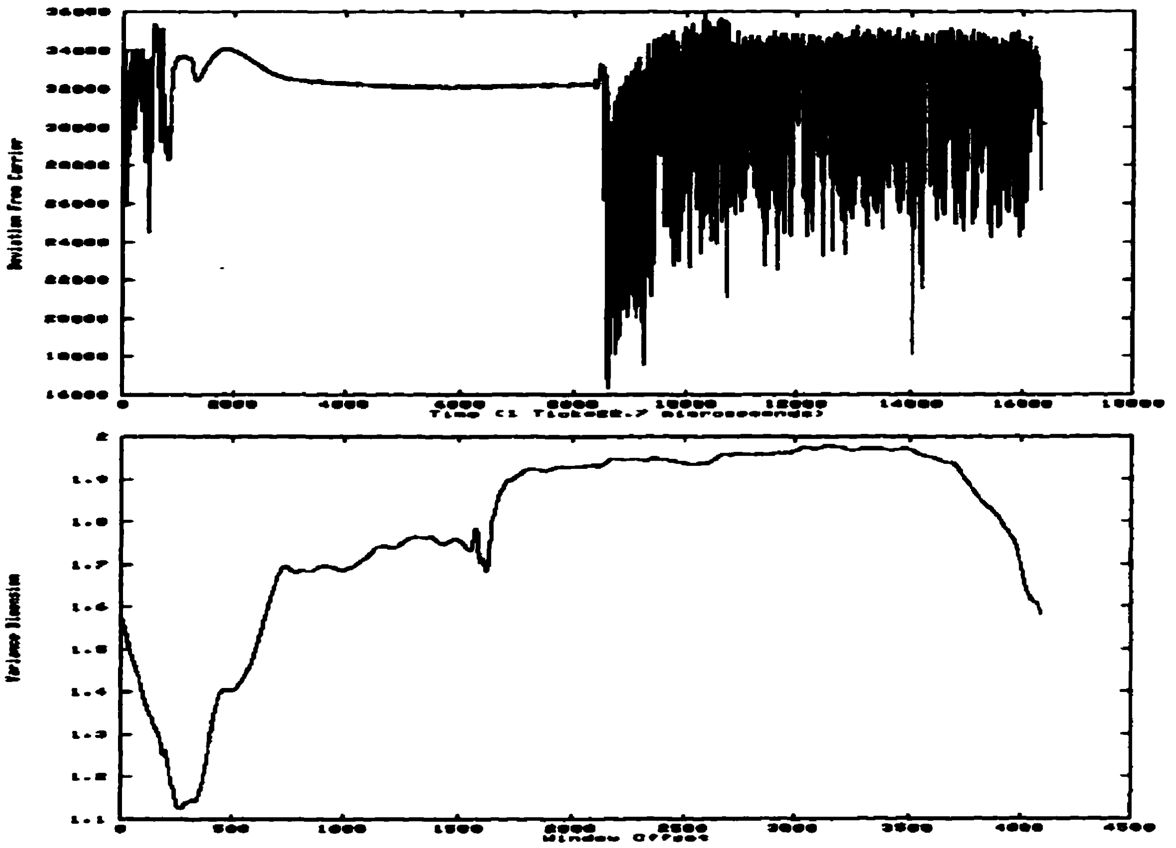


Fig. 3.1. Variance fractal dimension of a recorded transient.

The variance dimension is calculated in a separate function that is passed a window containing 512 samples of the signal. The variance dimension function returns the variance dimension which is stored in an array. The value of b is set to 1 with K_{max} equal to 16 and K_{min} set to 2. The loop index k is set to the value of K_{max} and is decremented after each variance calculation. The variance calculation used in the implementation is identical to Eq. 3.14. To compute the variance dimension, the method of least squares is used to find the slope of a line that best fits the points representing the log of the variance that is computed in each loop iteration. Once the slope has been found, it is divided by 2. The slope divided by 2 equals the Hurst exponent which is then subtracted from 2 to determine the variance dimension. Once the variance dimension has been returned for the current window, the next window is constructed from the current window by incrementing its position by one sample. This new window is passed to the variance dimension function for analysis. This process is repeated until the window has incremented through every sample. However, when the window reaches the 3,585th sample, the window's size exceeds the size of the signal. To remedy this situation, a wraparound procedure is put in place. Thus, when the window exceeds the signal size, those samples in excess are filled with the signal samples starting at the first sample. Figure 3.2 gives a visual interpretation of this procedure. This procedure is valid due to the fact that a circular buffer is used in acquiring the signal. By implementing a wraparound policy, a complete coverage of the variance dimension is assured.

With the variance dimension measured for each window, the noise separation can proceed. The first step in the noise separation is the localization of the minimum dimension. For most signals, the minimum dimension indicates the transition from noise to transient. The localization of the minimum dimension consists of a complete search of the array containing the variance dimensions for each window.

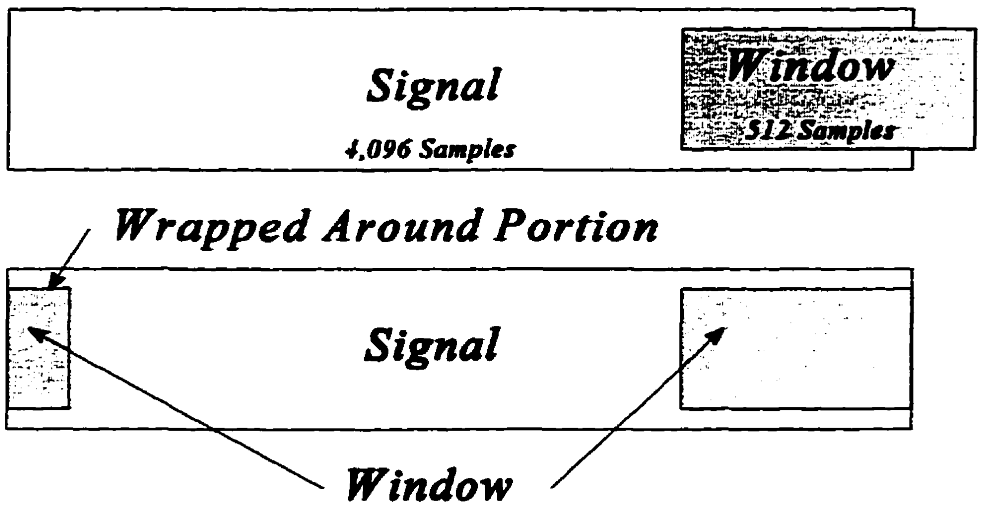


Fig. 3.2. Wraparound procedure.

When a local minimum dimension is located, the minimum's position within the array is stored. With each new local minimum found, the previous minimum is overwritten and the new location is stored. When the search is complete, the minimum currently stored becomes the global minimum representing the transition point. However, the minimum dimension does not always indicate the transition from noise to transient, thus requiring additional analysis. It is known that as the window passes over the noise and into the transient, the dimension begins to drop. By taking the derivative of the slope of the drop, another indicator results which can be used to detect the beginning of all transients. However, the drop in the variance dimension is not monotonically decreasing. In fact, there are many subtle changes in the direction of the slope. To smooth out these subtle changes a lowpass filter is passed over the variance dimension array. The filter takes 101 samples and averages these points. The sample upon which the filter is centered is replaced with the average value. Each sample contained in the variance dimension array is filtered to smooth out the subtle details. A wraparound policy is again implemented when the filter exceeds the dimensions of the variance dimension array. To calculate the derivative, a search is initiated to locate where the dimension begins to indicate the onset of a transient. This is achieved by locating the point in which the dimension drops below 1.8. When this point is found, the derivative is calculated to measure the change in the variance dimension. The derivative is calculated for 532 samples past the point at which the dimension drops below 1.8. In order to emphasize the derivatives most likely to indicate a transition point, a weighting of the derivatives is put into place. Since derivatives between 1.3 and 1.0 are most likely to indicate the beginning of the transient, these are derivatives that are emphasized the most. The weighting of the derivatives is defined by

$$\textit{weighted derivative}[i]=50^{(2-\textit{variance dimension}[i])}*\textit{derivative}[i] \qquad (3.18)$$

where i ranges between 1 and 532. Weighting the derivatives as they near 1.0, ensures that more negative changes occurring between higher dimensions does not lessen the accuracy of localizing the transition point.

There are two cases when the maximum negative slope is used to indicate a transition from noise to transient: (i) when the distance in samples between the minimum dimension and maximum negative derivative is exceedingly large (greater than 532 samples), and (ii) when the transient does not begin at the minimum dimension. The first case occurs when the minimum dimension reaches a minimum but continues to drop at a rate much slower than the rate found when the variance dimension drops from approximately 2.0 as shown in Fig. 3.3.

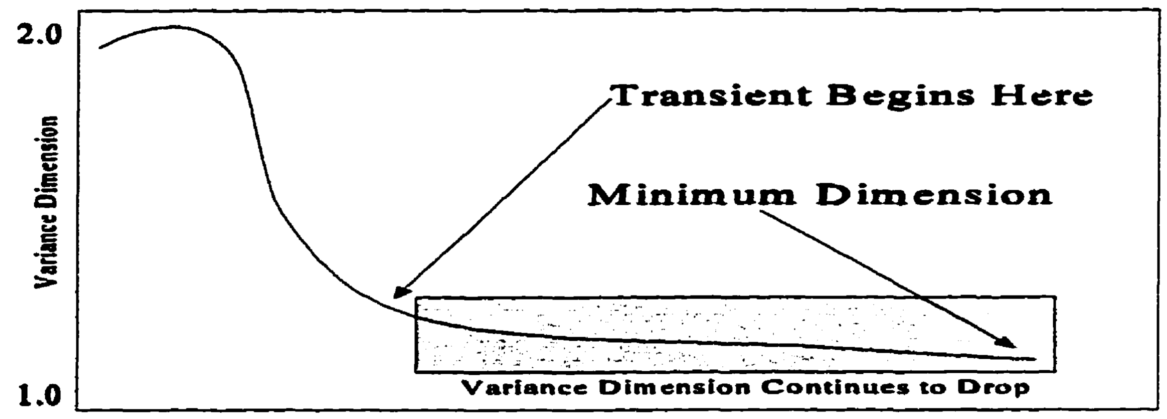


Fig. 3.3. Slow monotonic decrease in variance dimension.

The second case is indicated by a plateau as the variance dimension drops as shown in Fig. 3.4.

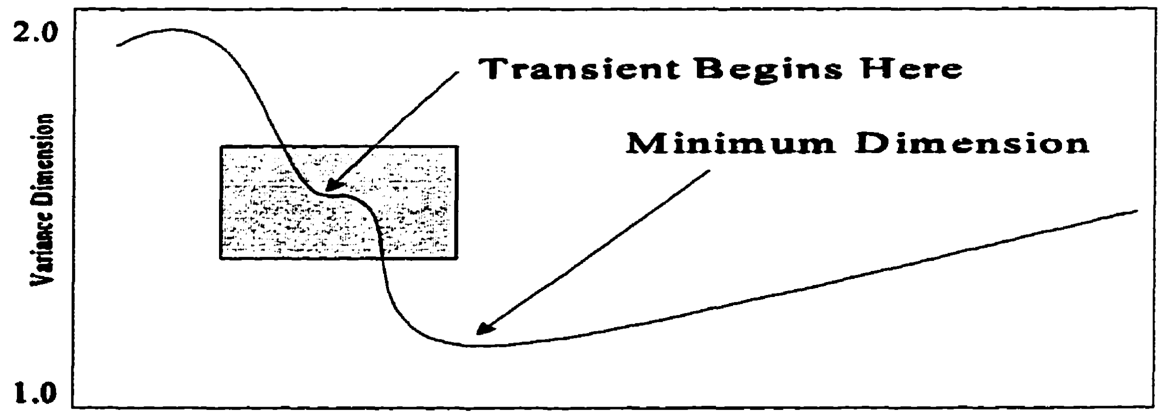


Fig. 3.4. Plateau indicating the onset of the transient.

The first case is easy to detect, if the minimum dimension location is found after the slope changes from being negative to positive or zero, the minimum dimension occurs too late to indicate a transition point. Therefore, the maximum negative derivative should be used as the indicator. The second case requires that the positive derivatives be monitored and used in finding the transition point. To locate the plateau, the zero crossing point of the derivative is recorded. If the maximum negative derivative occurs after this point, the maximum negative derivative before the zero crossing is taken as the transition point between the noise and the transient. Figure 3.5 gives a visual description of this second case.

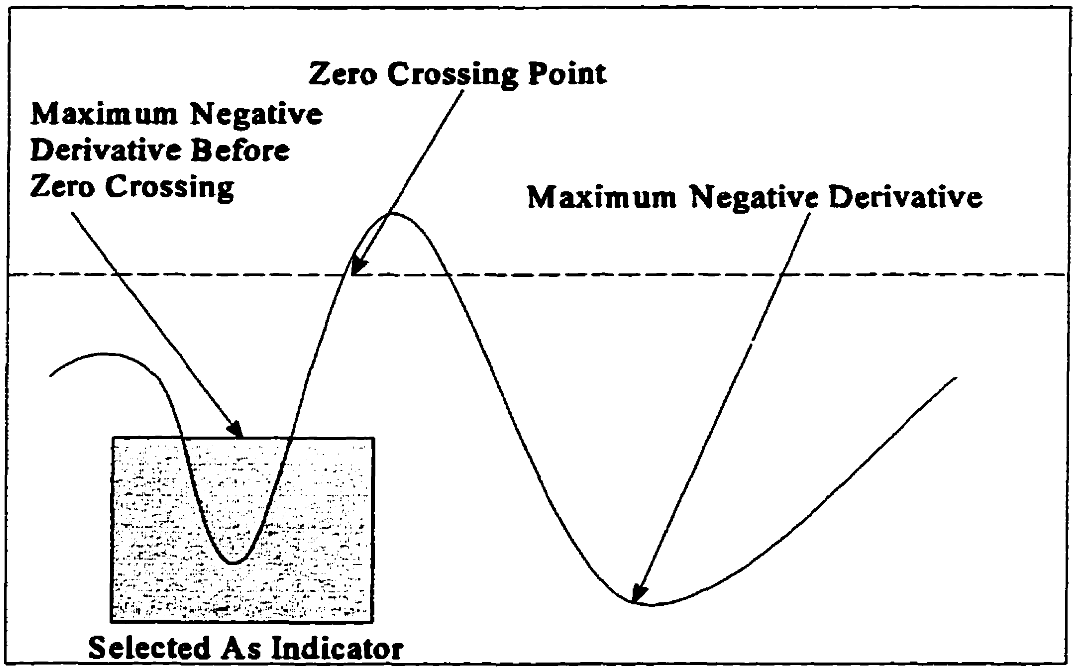

Zero Crossing Point
Maximum Negative Derivative Before Zero Crossing
Maximum Negative Derivative
Selected As Indicator

Fig 3.5. Selected indicator for the onset of the transient.

Since the window in which the variance dimension is calculated is incremented by only one sample for the next calculation, the locations of the dimension or the derivatives recorded represent the transition point. To extract the signal from the noise, 2,048 samples beginning at the recorded location are placed in a new array. This array now contains the transient which is further processed for the feature selection procedure discussed later. The feature selection procedure requires further reduction in the transient's size, to a total of 64 samples. Each of these 64 samples are super-samples constructed from averaging 32 consecutive samples from the transient array. Figure 3.6 shows this procedure. On completion of the generation of the super-samples, the software implementation writes the super-samples to disk for use by the feature selection procedure.

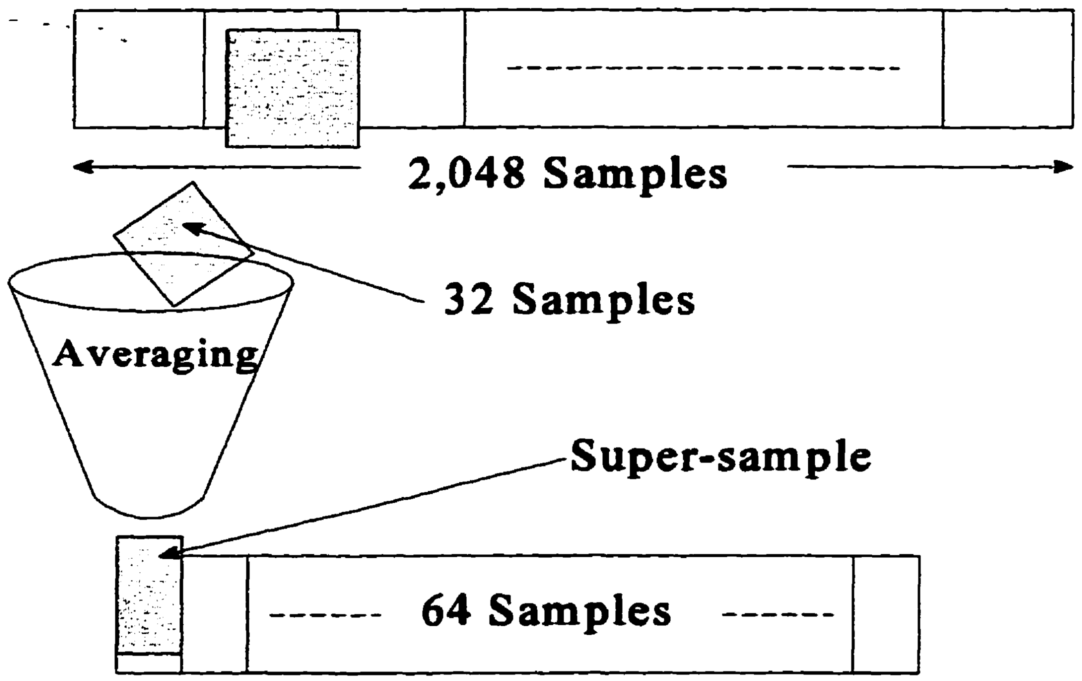


Fig. 3.6. Creation of the super-samples.

3.5 Summary

This chapter has described a technique for separating in time the noise component from the transient by implementing the variance dimension. The variance dimension is computed within a window which is passed over the noise and transient. A minimum in the variance dimension or a large negative derivative indicates that the transient has been located. The offset of the window in which the minimum variance dimension or negative derivative occurs, localizes the beginning of the transient.

CHAPTER 4
MULTIRESOLUTION ANALYSIS

4.1 Intoduction

This chapter will deal with the issue of feature selection. Feature selection is responsible for selecting the features of the transient that best represent it. These features in turn, are used to identify the transmitter responsible for the transient's generation. In order for efficient feature selection, a means of representing the features is required. The representation should be compact, and must include the reconstruction of the signal from the representation. A further restriction on the representation is that it must handle the non-stationary nature of the transients. An excellent candidate is wavelet analysis, [Chui92] [Daub94] [Youn93], and in particular, the multiresolution analysis [Mall89]. In the time domain, a wavelet is a signal with two special properties: it must be oscillatory and have compact support, meaning the time duration of the wavelet is limited. The wavelet analysis is similar to Fourier analysis in its function and operation. As with Fourier analysis, wavelet analysis transforms a signal in both continuous and discrete contexts. However, unlike the Fourier domain, the wavelet domain is two dimensional with a time and scale axis. The two dimensions are required to accommodate the compact support of the wavelet. Wavelets are scaled and translated to transform a signal from the time to the wavelet domain. As the wavelet is scaled, wavelet analysis measures increasingly more detail in the signal. The translations of the wavelet ensure a complete coverage of the signal due to its compact support. This is in comparison to the one dimensional Fourier frequency domain which represents the frequency composition of a signal. Thus, wavelet analysis allows the localization of both time and scale in representing a signal.

4.2 Multiresolution Analysis

Several techniques exist for transforming signals from the time domain to the wavelet domain in both continuous and discrete applications. However, a particularly useful technique for generating a compact representation of a signal for feature selection is multiresolution analysis. The basic idea behind multiresolution analysis is assuming there exists a set of spaces such that

$$\ldots \subset V_2 \subset V_1 \subset V_0 \subset V_{-1} \subset V_{-2} \ldots \tag{4.1}$$

with the following property, if

$$f(x) \in V_i \tag{4.2}$$

where *f(x)* is some function, then

$$f(x-2^{-i}k) \in V_i \tag{4.3}$$

and

$$f(2x) \in V_{i-1} \tag{4.4}$$

Also required in the analysis, is an orthogonal complement of V_i in V_{i-1} called W_i. Thus,

$$V_{i-1} = V_i \oplus W_i \tag{4.5}$$

which indicates that W_i contains the added detail to get from V_i to V_{i-1}. By expanding Eq. 4.5, the following results

$$V_{i-1} = W_i \oplus W_{i+1} \oplus W_{i+2} \oplus W_{i+3} \oplus \ldots \tag{4.6}$$

Thus, any resolution can be constructed from the sum of added details. By calling the complement space the wavelet space, and having a wavelet function defined as the orthogonal basis that adheres to the properties listed above, the multiresolution analysis technique is somewhat intuitive. If a signal is projected on both the V_i and W_i spaces, V_i will contain the signal at a lower resolution while W_i will contain the detail removed from the signal due to its projection on V_i. With V_i in hand, V_i is projected on V_{i+1} and W_{i+1}. This time V_{i+1} contains the signal at even a lower resolution while V_{i+1} contains the detail corresponding to this projection. This decomposition is continued until the minimum resolution is met, usually the D.C. component of the signal. Once the decomposition is complete, Eq. 4.6 has been satisfied by the generating of all the added detail components and hence the contribution of each scaled and translated wavelet function to the signal has been determined. The wavelet function's contribution is represented by a wavelet coefficient. The larger the coefficients value, the greater the contribution while smaller coefficients denote lower contributions. However, one issue is left unresolved, the means of getting from one resolution to the next. To achieve this, a scaling function, ϕ, is implemented to remove the detail from the signal making it more and more coarse as the decomposition progresses. Both wavelet and scaling functions remain unchanged throughout the entire decomposition with the exception of their scale. The functions are scaled to analyze the signal as its resolution is decreased.

With the discussion focusing on the detail of a signal and the reduction in resolution of a signal, it seems natural to implement filters to perform these actions. A lowpass filter could smooth a signal and remove the detail, while a highpass filter could be used to retain the detail. The application of filters in multiresolution analysis is based on the formation of a series of perfect half-band filters. The first filter application operates on the entire signal bandwidth. The signal's bandwidth is split into two, a low frequency and a high frequency band. The low frequency band is split again while the high frequency band is placed aside. The splitting of the low frequency band at each stage is continued until the D.C. component is reached just as was described above. At each stage, the signal at the output of the lowpass filter contains less and less detail, thus the lowpass filter can be considered the scaling function. While on the other hand, the highpass filter output contains the detail of the signal at the corresponding resolution, thus it can be considered the wavelet function.

Figure 4.1 shows how the bandwidth is split at each stage. Figure 4.1 shows an important property of wavelet analysis, that is, both scale and time localization cannot be simultaneously achieved. From Fig. 4.1, the initial splitting of the signal's entire bandwidth yields large high and low frequency bands. At this stage the wavelet analysis is looking at the signal's most detailed components. In other words, these are the highest frequency components of the signal which have a very compact duration. As the resolution decreases, the signal's detailed components become less compact and are composed of lower frequencies. It is also noticed that the bandwidth of the components is becoming increasingly

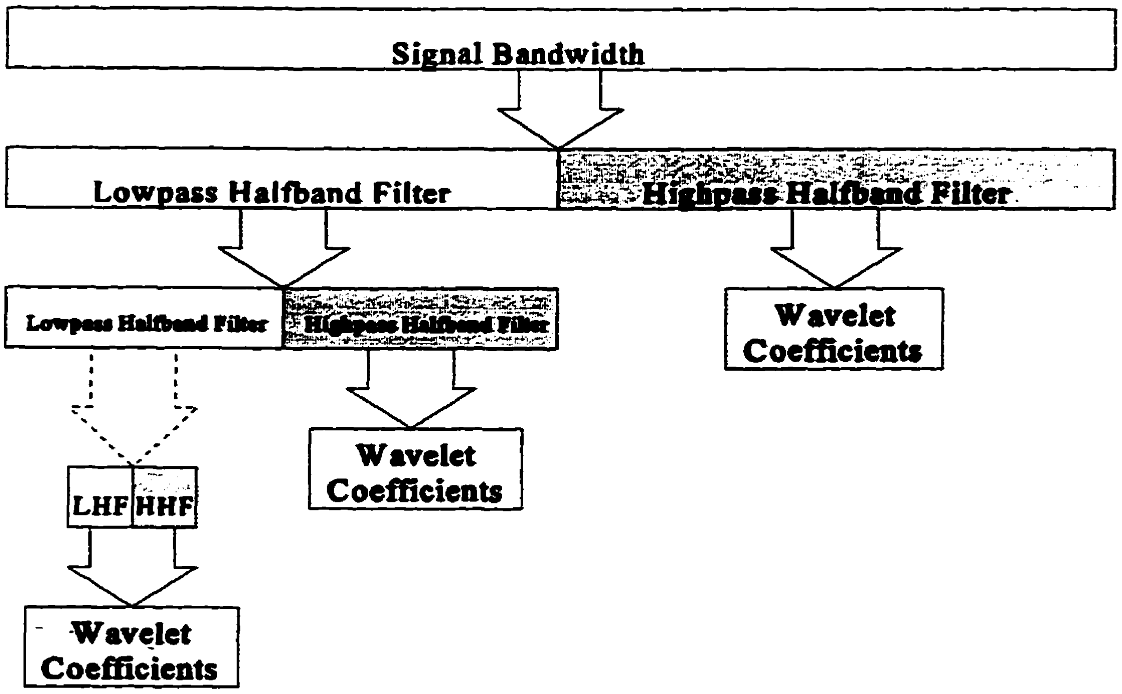


Fig. 4.1. Bandwidth of filters used in the signal decomposition.

narrow. As the bandwidth of the filters become increasingly narrow as the analysis progresses, the accuracy of determining the spectral components of the signal is greater due to the fact that the narrow bandwidth filters pass smaller numbers of possible spectral components as compared to the larger bandwidth filters. Thus, there exists a correlation between the bandwidth of the filters used at a particular scale in the analysis and the accuracy of determining the spectral components of the signal at that scale as shown in Fig. 4.2.

Of utmost importance for feature selection is that the signal representation must be orthogonal. By choosing an orthogonal wavelet basis function, the projections on the wavelet space provide coefficients that represent a single component of the signal. Thus, by removing wavelet coefficients, that feature of the signal is effectively removed. Therefore, to select the

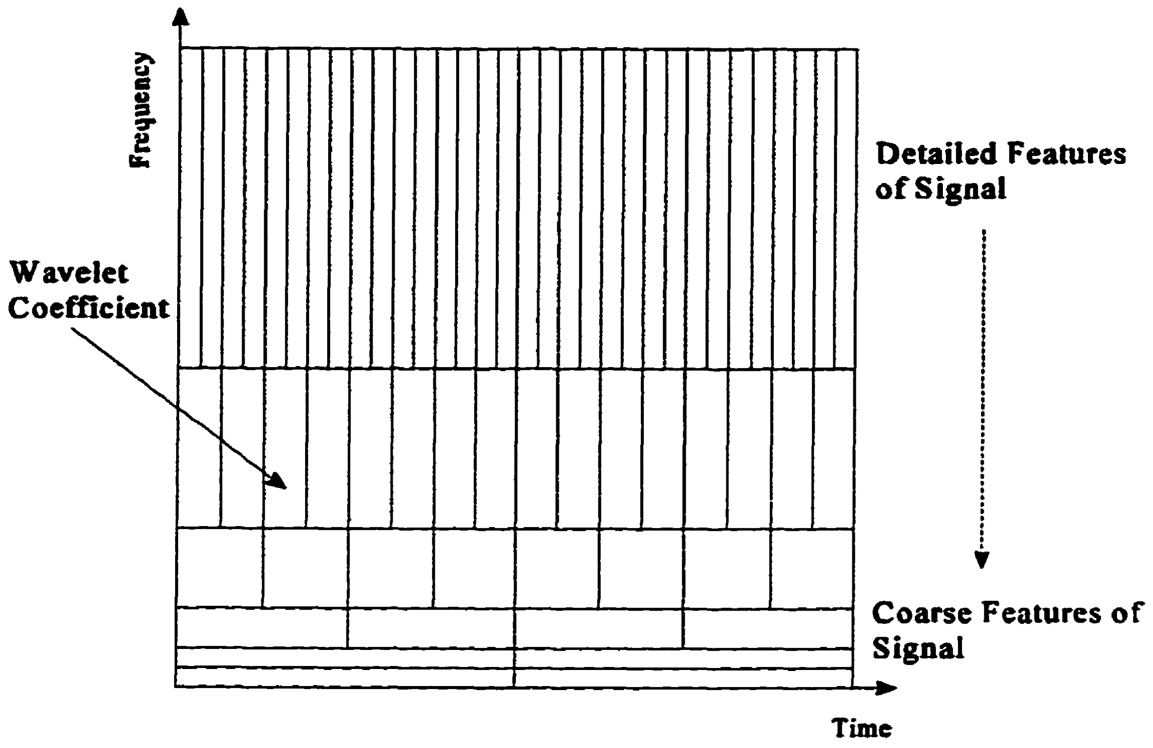


Fig. 4.2. The wavelet time-frequency domain.

critical features of a signal, the corresponding wavelet coefficients are chosen to represent the signal. By using the wavelet coefficients, a very efficient and compact means of representing the features has been acquired. The analysis implemented in this study used the Daub 4-tap filter pair which is orthogonal in both scale and translation. In addition to orthogonality, it is necessary that the signal representation be reversible. That is, it is possible to transform the representation back to the original signal. This is required to ensure that the features selected accurately represent the signal. To test the accuracy, the selected features are transformed back to the time domain and compared to the original signal. If the representation's accuracy is too low, further coefficients need to be added. To have perfect reconstruction, a set of four filters are needed. Two filters are used to decompose the signal while the second pair are responsible for the reconstruction of the signal. Figure 4.3 shows a single stage decomposition/reconstruction system. The filter pair used in decomposing the signal are labeled *G* and *H* while the reconstruction pair are labeled G^* and H^*. The lowpass filters, *G* and G^* and the highpass filters *H* and H^* must possess two important properties. The first of these properties requires that the decomposition/reconstruction system be an identity system or that

$$GG^{*} + HH^{*} = I \tag{3.7}$$

In addition to the above property, the filters must be orthogonal

$$GH^{*} = 0 \tag{3.8}$$

and

$$HG^{*} = 0 \tag{3.9}$$

If these conditions are met, perfect reconstruction is attainable.

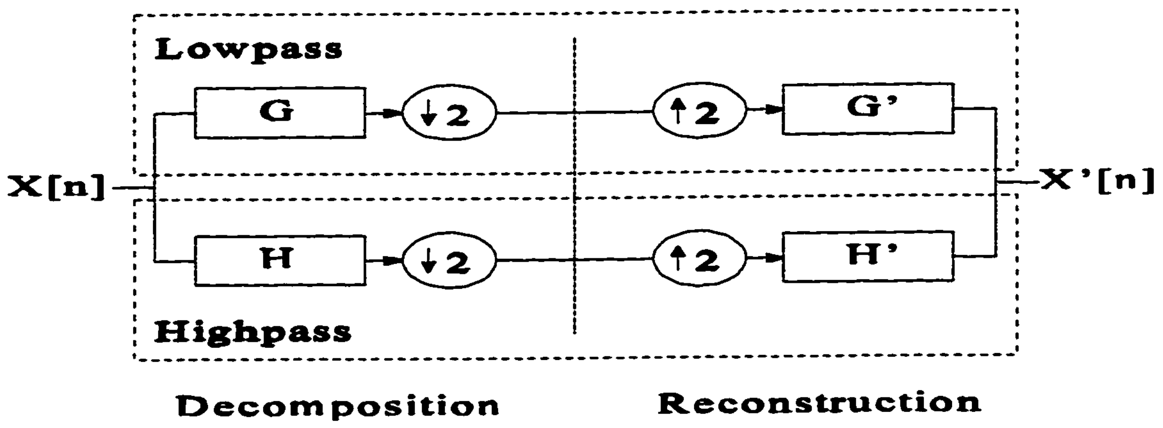


Fig. 4.3. Single stage decomposition and reconstruction.

The reconstruction phase is similar to decomposition stage except that the analysis begins at the D.C. component. The wavelet coefficients corresponding with the lowest resolution are passed to G^* and H^*. The signals are summed together and passed to the next stage. The resultant signal is passed through the next pair of G^* and H^* filters and summed together. This analysis is continued until the signal is reconstructed. Figures 4.4 and 4.5 summarize the decomposition and reconstruction of a signal. The down-sampling or up-sampling is required to scale the signal appropriately for the next stage. The down-sampling is used in the decomposition while the reconstruction phase requires up-sampling. To down-sample a signal, every second sample is thrown away while up-sampling is achieved by interleaving a zero between each sample. Figures 4.4 and 4.5 can be performed by a

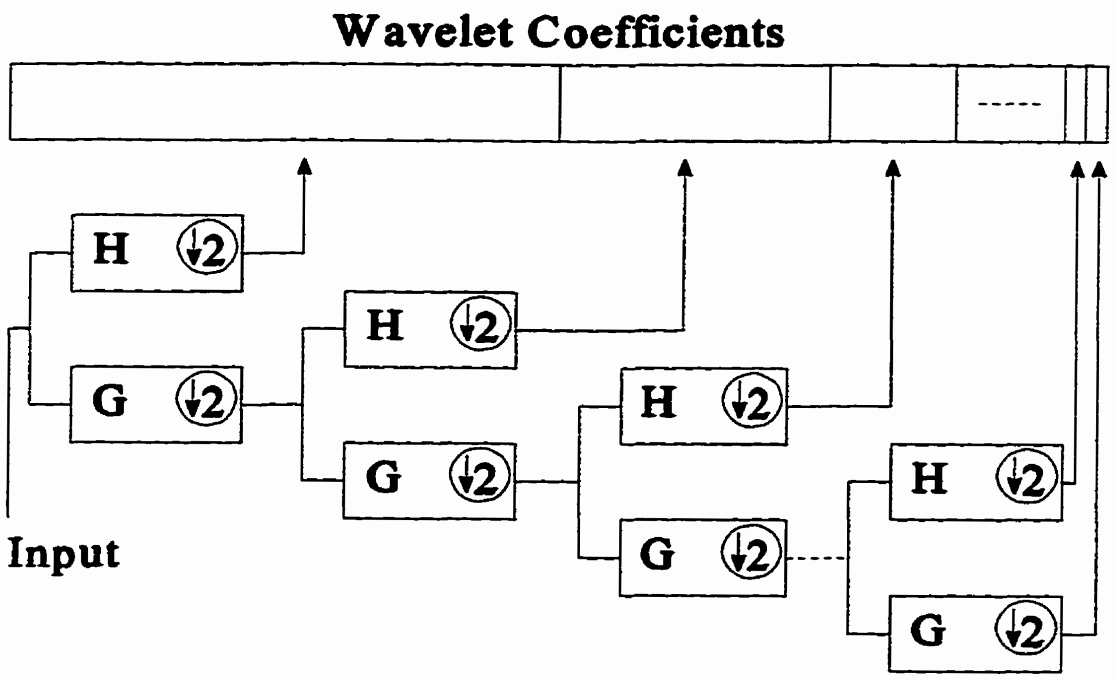


Fig. 4.4. Signal decomposition.

computer as described in the next section.

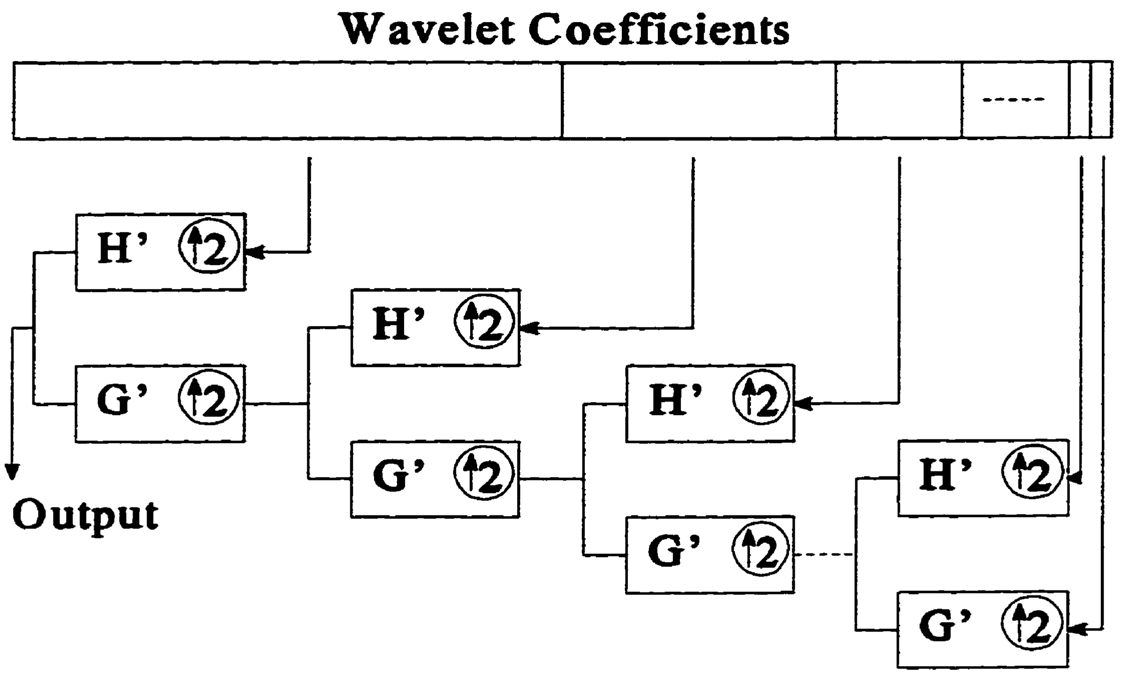


Fig. 4.5. Signal reconstruction.

4.3 Software Implementation

The software implementation of the wavelet transform is developed in the book *Numerical Recipes in C* [PTVF92]. The code developed is extremely efficient in terms of memory use and execution time. To decompose a signal into its corresponding wavelet coefficients, the impulse responses of the low and highpass filters are convolved with the signal. The output of the highpass filter are the wavelet coefficients at the analysis scale. The output of the lowpass filter is scaled by down-sampling and passed onto the next stage. The impulse response of the Daubechies 4-tap filter contains 4 coefficients defined as

$$c_0 = \frac{1+\sqrt{3}}{4\sqrt{2}} \tag{3.10}$$

$$c_1 = \frac{3+\sqrt{3}}{4\sqrt{2}} \tag{3.11}$$

$$c_2 = \frac{3-\sqrt{3}}{4\sqrt{2}} \tag{3.12}$$

and finally,

$$\frac{1-\sqrt{3}}{4\sqrt{2}} \tag{3.13}$$

For the lowpass filter, all the coefficients are positive and when a highpass filter is desired, c_0 and c_2 are negative. To perform the convolution between the signal and the filter impulse response, a matrix is constructed consisting of the filter coefficients. The first row contains the filter coefficients in their lowpass configuration as does the third, fifth and all odd rows. However, each odd row entry is offset two positions from the previous odd row entry's location. All other matrix entries are set to zero. Figure 4.6 shows this configuration. Inspection of Fig. 4.6 indicates the highpass filter coefficients are contained in the even rows of the matrix with a similar offset pattern. This interleaving of filter coefficients is permitted since every second row of separate low and highpass filter convolution matrices would be empty. The same type of convolution matrix is used for the reconstruction of the signal except the matrix shown in Fig. 4.6 is inverted. With the constraint of orthogonality placed on the matrix, the inverted matrix is equal to the transpose of Fig. 4.6. To guarantee orthogonality, the following two constraints must be satisfied

$$c_0^2 + c_1^2 + c_2^2 + c_3^2 = 1 \tag{3.14}$$

and

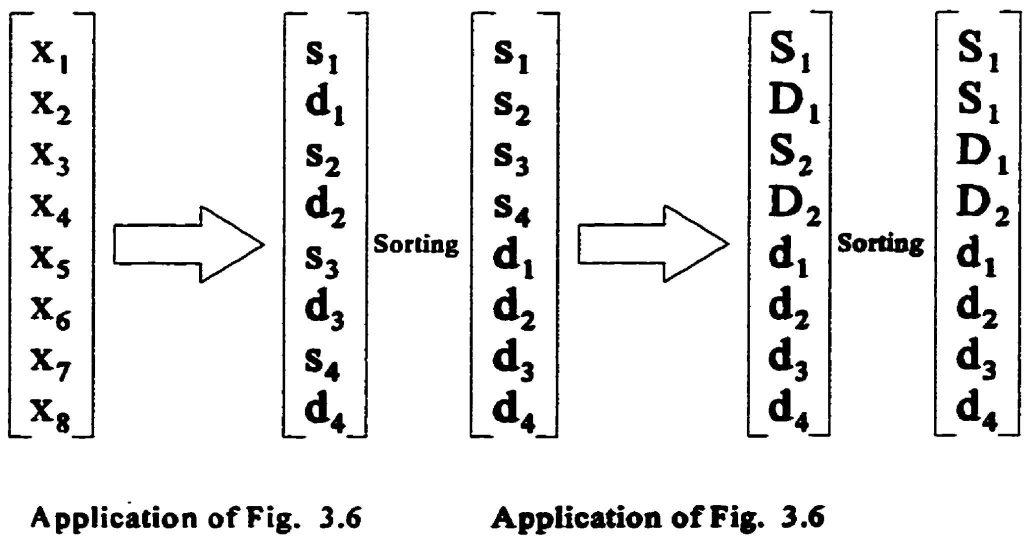
$$c_2 c_0 + c_3 c_1 = 0 \tag{3.15}$$

Using the coefficient values given above, it is noted that both of these constraints are met allowing the transposition of the matrix.

$$\begin{bmatrix}
C_0 & C_1 & C_2 & C_3 & & & & & & \\
C_3 & -C_2 & C_1 & -C_0 & & & & & & \\
 & & C_0 & C_1 & C_2 & C_3 & & & & \\
 & & C_3 & -C_2 & C_1 & -C_0 & & & & \\
 & & & & & & \ddots & & & \\
 & & & & & & C_0 & C_1 & C_2 & C_3 \\
 & & & & & & C_3 & -C_2 & C_1 & -C_0 \\
C_2 & C_3 & & & & & & & C_0 & C_1 \\
C_3 & -C_2 & & & & & & & C_3 & -C_2
\end{bmatrix}$$

Fig. 4.6. Matrix used to decompose the signal.

To decompose a signal of vector length *N*, the vector is multiplied by the convolution matrix. The result is a second vector of length *N* containing *N/2* wavelet coefficients and *N/2* smooth data. The two data types are interleaved and require sorting. The odd vector elements contain the smooth information while the even elements contain the wavelet coefficients. The vector is sorted with the smooth information residing in the first *N/2* locations and the wavelet coefficients placed in the remaining *N/2* locations. The convolution matrix is reapplied to smooth information resulting in *N/4* new wavelet coefficients and *N/4* smooth elements. Once again the data is interleaved and requires sorting in the same fashion used above. The application of the convolution matrix is continued until only two smooth components remain, at which time the vector contains all the wavelet coefficients for the signal. Figure 4.7 shows the interleaving and sorting procedure for an input vector of size 8.



x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
s_1 d_1 s_2 d_2 s_3 d_3 s_4 d_4
Sorting
s_1 s_2 s_3 s_4 d_1 d_2 d_3 d_4
S_1 D_1 S_2 D_2 d_1 d_2 d_3 d_4
Sorting
S_1 S_1 D_1 D_2 d_1 d_2 d_3 d_4
Application of Fig. 3.6
Application of Fig. 3.6

Fig. 4.7. Coefficient ordering as a result of applying the matrix found in Fig. 4.6.

The algorithm requires inputs of power 2, but does not restrict the size of the exponent. The maximum size vector is restricted by the computer's memory. The inverse transform is similar except that the transpose of the matrix shown in Fig. 4.6 is used.

Multiresolution analysis provides an excellent framework for representing the features of a signal in a compact manner. The transient input to the above described software implementation is the output of the noise separation component described in the previous chapter. The vector size of the input is 64 samples and output of the wavelet analysis component is 64 coefficients. If reconstruction of a transient is required, the input is 64 coefficients and the output is a 64 sample signal.

4.4 Summary

This chapter provides a description of multiresolution analysis. Multiresolution analysis provides an excellent means of independently representing the features of the transients. To select the critical features of the transient, the corresponding wavelet coefficients are selected. The process is completely reversible allowing the reconstruction of the transient from the selected coefficients.

CHAPTER 5
GENETIC ALGORITHMS

5.1 Introduction

By using multiresolution analysis to transform a transient into a set of independent coefficients, a compact means of representing the features of the transient is obtained. However, not all of these features are significant and can be removed, leaving only those features that are most representative of the signal. One possible technique for feature selection is the genetic algorithm [Davi91] since it looks at many permutations of selected features and selects the best model of the original transient based on a selected measure of fitness. Genetic algorithms mimic natural evolution of species by the implementation of crossover, mutation, and selection of the fittest operations. It would be most desirable for the genetic algorithm to operate realistically like natural evolution, however, the computer has finite limitations. These limitations include, limited memory to hold a very large number of species and processing speed to operate on these large number of species in a finite amount of time. In the following sections, each of the operators will be described along with some heuristical features that ensure correct operation.

5.2 Terminology

Before the genetic algorithm is discussed in detail, some terminology is described to aid in the understanding of the genetic algorithms operators. The genetic algorithm operates on chromosomes which store the selected wavelet coefficients for a particular evolving solution. Genes represent the individual wavelet coefficients which make up the chromosome. The gene is a binary operand since it contains a one if the wavelet coefficient is selected and a zero if it does not. The chromosomes are stored in an array called the genetic pool. The genetic pool is passed between genetic modules, modified according to the function of the module. After each application of the genetic modules, a new generation of evolving solutions is created. The number of generations allows for a means in determining when to terminate evolution. When the genetic algorithm is complete, a genetic solution is returned. This is the best solution found in the genetic pool when the genetic algorithm is terminated.

5.3 Genetic Pool Initialization

The genetic pool is the foundation upon which the genetic algorithm is built and is where the evolving solutions reside and are operated on. The dimensions of the genetic pool are 20 chromosomes by 64 genes. Figure 5.1 pictorially shows the genetic pool.

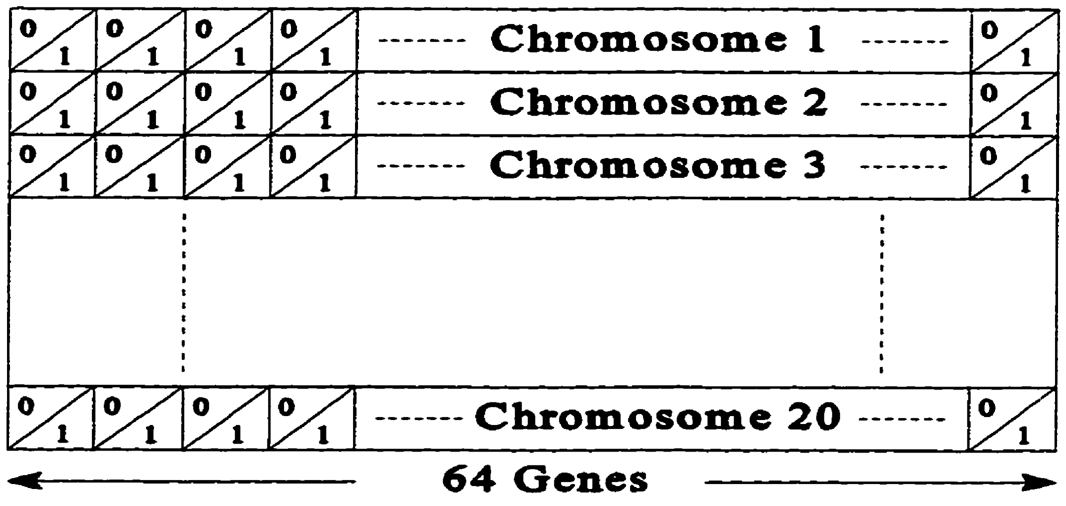


Fig. 5.1. The genetic pool.

The first step in the initialization is performing the wavelet transform on the input signal and storing the coefficients in a vector equal in size to the number of coefficients. It has been observed from the study of transients that the first eight wavelet coefficients contribute a significant amount of information to the signal. Therefore they should always be included in all evolving solutions. These coefficients describe the D.C. and low frequency information of the signal which is required for proper feature selection. To accommodate this requirement, the genetic pool initialization procedure adds these eight coefficients to the chromosomes by placing a one in each of the first 8 genes. Once this is complete, a random placement of the remaining allocated coefficients is performed. The remaining number of coefficients for allocation is equal to the total number of coefficients provided to represent the features to be selected, subtracted by 8. The random placement is required since the crossover and mutation operators do not add coefficients but just move coefficients to new locations that improve the fitness of the evolving solution. In order to perform the random placement, the chromosomes are split into two pieces for reasons discussed in the crossover section. The first section contains 24 genes (8^{th} to 32^{nd} genes) while the second section contains 32 genes (33^{rd} to 64^{th} genes). Each section is assigned a certain number of coefficients from the remaining allocated coefficients. The number of assigned coefficients in each section remains constant throughout the entire evolutionary process. The assignment of the coefficients is determined by the total energy of all the coefficients contained in each portion, the more energy, the more coefficients are allocated to the section to represent the features of the transient. The formula used for the section coefficient allocation is:

$$\textit{Section Coefficient Allocation} = \frac{\textit{Section Coefficient Energy}}{\textit{Total Energy}} * (\textit{Remaining Allocated Coefficients}) \qquad (5.1)$$

where

$$\textit{Total Energy} = \textit{First Section Coefficient Energy} + \textit{Second Section Coefficient Energy} \qquad (5.2)$$

The section coefficient allocation is performed for both the first and second sections. Once the allocation has been determined, the genes are modified accordingly. The modification is performed in the following manner and is shown in Fig. 5.2:

i. A random number in the appropriate section range is generated.

ii. The gene that corresponds to the random number is set to one.

iii. The above two steps are repeated *n-1* times where *n* represents the section allocation.

iv. The above steps are repeated for the second section.

After performing these procedures, the genetic pool has been initialized and the genetic algorithm can begin evolution.

5.4 Crossover

As in nature, the genetic algorithm attempts to take the best attributes of the previous generation and passes these attributes to the next generation. The task of this operation is performed by the genetic algorithm's crossover module. As the name implies, crossover takes

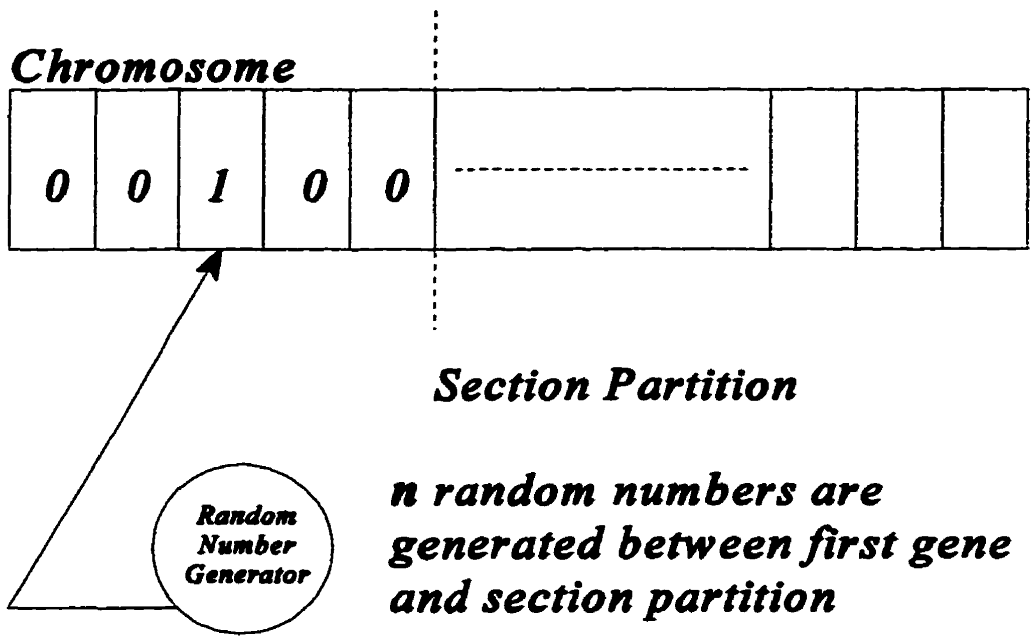
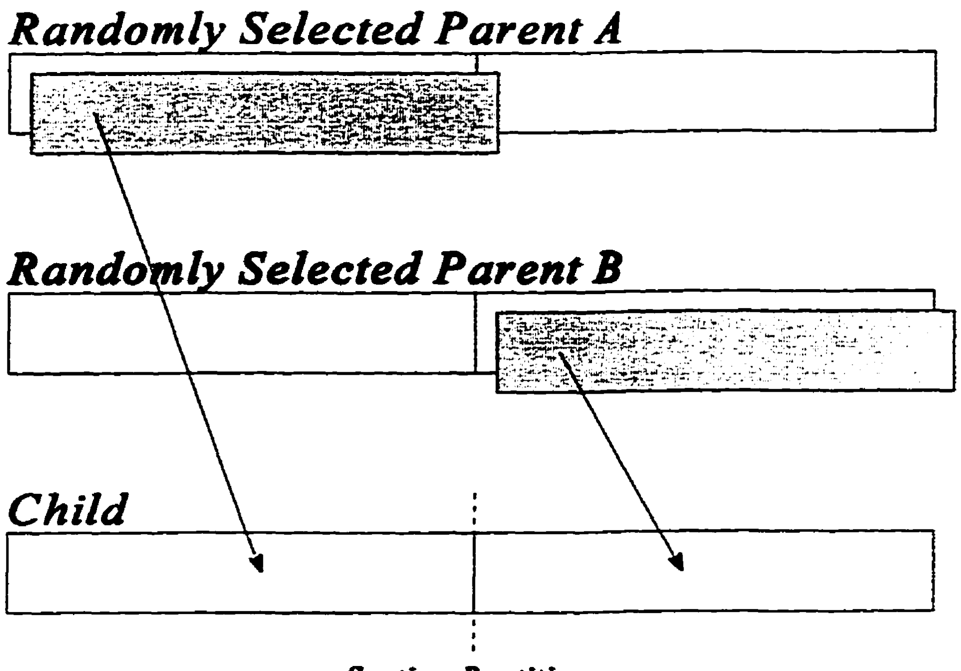

Chromosome
0
0
1
0
0
Section Partition
Random Number Generator
n random numbers are generated between first gene and section partition

Fig. 5.2. Initialization of the genetic pool.

two portions of separate chromosomes and combines the portions into a new chromosome as shown in Fig. 5.3. The hope is by taking the components of the previous generation (parents) and combining them together (child), a better evolving solution will result.

This implementation of the genetic algorithm requires that the total number of coefficients selected does not exceed 32. For this to occur, the crossover point must remain fixed and the number of coefficients contained in each section must also remain fixed. As was mentioned in the previous section, the partition point between the two sections was between the 32nd and 33rd genes. This partition point is in fact the crossover point. If the crossover point is not fixed, the number of coefficients will increase past the limit imposed by the genetic algorithm. For example, consider the scenario analyzed in Table 5.1.



Section Partition

Fig. 5.3. Crossover.

Table 5.1. Two chromosomes are selected for crossover and the crossover point is set at 32.

Chromosome 1	**24 Coefficients are contained in this first section**	**4 Coefficients are contained in this second section**
Chromosome 2	**16 Coefficients are contained in this first section**	**10 Coefficients are contained in this second section**

Now using the crossover scheme shown in Fig. 5.3, the total number of coefficients selected and contained in the child chromosome is 34, exceeding the limit by 2 coefficients. Hence, by fixing the crossover point and the number of coefficients contained in the sections, the coefficient limit can never be exceeded.

Now that the requirements for the crossover procedure have been defined, the operation of the crossover procedure can be described. The first stage of crossover is the random selection of two chromosomes which become the parents. The random numbers used by this genetic algorithm implementation are generated by the built in multiplicative congruential random number generator provided in the *C* programming language. To perform crossover, the first coefficient section of the first parent is copied to a vector while the second section of the second parent is also copied to this vector. This vector is called the child and randomly replaces one of the parent chromosomes. This copy and replacement operation is repeated 20 times before the crossover procedure is complete. The crossover procedure is performed once every generation.

5.5 Mutation

To further evolve a solution, mutation is performed on the genetic pool. When mutation is performed on a chromosome, it is hoped that the chromosome being mutated will further evolve the chromosome. To perform mutation, a chromosome is selected at random and with this chromosome two randomly selected genes are also selected. The value of each gene is evaluated and if it is found they are complementary, their values are switched. If it is found that their values are not complementary; i.e., both are zeros or ones, no switching is performed. This implementation is used to ensure that the number of selected wavelet coefficients does not exceed 32 as was the case in the crossover procedure. Figure 5.4 shows the randomly selected chromosome and one of the genes being mutated. It is assumed that in Fig. 5.4 the other gene being mutated contained a one. The arrows show that a zero is taken from the randomly selected gene's location, complemented, and replaced. It should also be noted that mutation can only take place within crossover ranges defined in the previous section to keep the number of selected coefficients constant in each range throughout the genetic algorithm implementation. The maximum number of mutations performed per generation is set at 8.

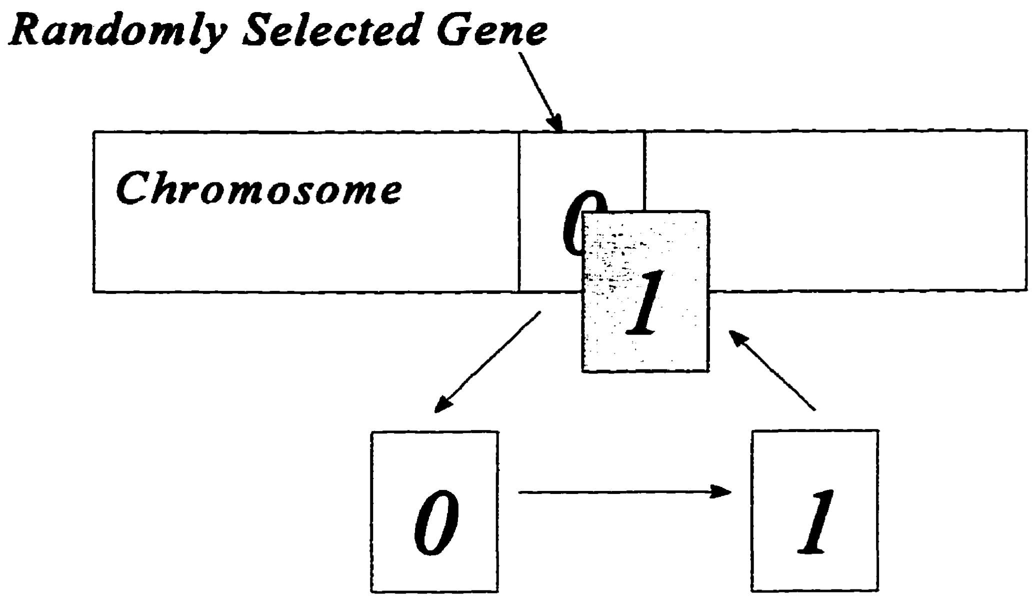


Fig. 5.4. Mutation.

It is important not to disturb the chromosomes by over mutating the genes. If too many genes are mutated, it is possible to degrade the evolving solutions instead of enhancing it, thus increasing the generations required to find a satisfactory solution. Therefore, it is advisable to keep a number of mutations performed per generation to a very limited number. In this study, only a little over half a percent of the genes are mutated per generation, thus allowing for a smooth evolution of the chromosomes.

5.6 Selection

The final module performed during each generation is selection. Selection is responsible for selecting the most fit evolving solutions and passing them on to the next generation. Those evolving solutions not fit enough are not included and are disposed of. To determine the fitness of a solution, the energy of the selected coefficients are compared to the energy of all the coefficients. The result of this comparison is a number between zero and one. To determine the energy of the evolving solutions represented by the chromosomes, a weighting scheme is used. It has been observed from the study of transients that different regions of wavelet coefficients typically have different effects on the quality of the model being generated. Table 5.2 summarizes these observations.

Table 5.2. Effect of inclusion of wavelet coefficients on model quality.

Coefficient Range	Effect of Inclusion on Model
8th to 16th	Great
16th to 32nd	Large
32nd to 64th	Small

Using the information found in Table 5.2, the following formula has been derived:

Table 5.3. Contribution made to total energy by each coefficient range.

Coefficient Range	Contribution To Total Energy
8th to 16th E_1	100%
16th to 32nd E_2	90%
32nd to 64th E_3	50%
Total Energy =	$E_1+0.90*E_2+0.50*E_3$

Once the total energy of the selected coefficients has been determined for each chromosome, it is divided by the total energy of all the coefficients. These normalized fitness values are again normalized to the total of the normalized fitness values. A genetic roulette wheel is constructed from the normalized values:

$$Roulette[i]=\sum_{j=1}^{i} Normalized\text{-}Normalized\ Fitness \qquad i=1,..,20 \tag{5.3}$$

where

$$\sum_{i=1}^{20} Roulette[i]=1 \tag{5.4}$$

A random number between zero and one is generated and compared to *Roulette[i]* for each *i*. When the random number is greater than *Roulette[i]*, *i* points to the chromosome to be selected for the next generation. The selected chromosome is copied to a temporary genetic pool and another random number is generated for the next selection. The random number generation and comparison is performed 20 times in total in order to fill the temporary genetic pool. Once the selection is complete, the temporary pool replaces the original genetic pool and the next generation has been established. The whole idea behind the roulette selection is that the greater the fitness of an evolving solution, the greater the chance of selection and being passed on to the next generation.

Since the number of selected coefficients is limited by the coefficient allocation provided to represent the transient features, the energy of a model's selected coefficients never reaches that of the original transient represented by all the coefficients. It is the responsibility of the genetic algorithm to select those coefficients that maximize the energy of the allocated number of coefficients provided. Models with a greater level of maximization tend to proceed to the next generation for possible further evolution.

Once the selection is complete, the whole process of crossover, mutation, and selection is repeated until the termination signal is given. The termination signal is given once the 100th generation has been completed. By the 100th generation, no further evolution was exhibited by the chromosomes, thus new generations will not enhance a solution any further. Once the termination signal has been received, the chromosome with the highest fitness is selected as the genetic model and is prepared for classification. The preparation for classification involves the normalization of the selected coefficients between zero and one by dividing the coefficient by the total energy of its wavelet scale. These normalized wavelet coefficients are written to a file and used by the neural network for training or classification. If a wavelet coefficient is not included in the model, its value is set to zero which is also written to the file.

5.7 Summary

This chapter describes the genetic algorithm which is a very powerful feature selection tool. Selection is achieved by implementing the four modules of a genetic algorithm. The first module is the initialization of the genetic pool followed by the repeated application of crossover, mutation, and selection of the fittest. The genetically evolved solution containing the most significant features at the end of the evolutionary process is selected as the output. This output is either used to a train neural network or to be classified by a neural network.

CHAPTER 6
NEURAL NETWORKS

6.1 Introduction

In the realm of computing, it would be desirable to have a computer learn in a similar way that our brain learns. A computer cannot learn through experience, nor can it generalize as the human brain can. The brain is highly complex, nonlinear and has a parallel structure. It is made up of in the order of 10 billion neurons and 60 trillion connections (synapses) [Hayk94]. To mimic this structure, a software implementation of a greatly simplified brain called a neural network has been developed. The software implementation uses neurons constructed from linear or nonlinear elements, connections called weights and the means of training the network to learn. Neural networks range in complexity from the very simple to the very large with a complicated topology. However, since this thesis does not focus on neural networks, one of the simplest and well known neural networks has been selected to perform the required transient classification.

6.2 Neurons

To begin with, the neuron is described in terms of its components and inputs. The neuron is a simple device that has a variable number of inputs which depends on the network topology. The neuron sums together these inputs and passes the total through some transfer function called the activation function. The transfer function can be both linear or nonlinear depending on the network application. In mathematical terms, the summing component of the neuron is described by

$$u_k=\sum_{j=1}^{p} w_{kj}x_j \tag{6.1}$$

where u is the input to the activation function of the neuron, w denotes the weights connecting the neuron to other neurons or inputs, and x is the output of that neuron or input. The output of the neuron is described by

$$y_k=\phi(u_k-\theta_k) \tag{6.2}$$

where ϕ represents the activation function and θ denotes the threshold term. The threshold allows greater learning potential by allowing the decision boundary to leave the origin. By assigning the threshold a weight and input, Eqs. 6.1 and 6.2 can be rewritten as

$$v_k=\sum_{j=0}^{p} w_{kj}x_j \tag{6.3}$$

and

$$y_k=\phi(v_k) \tag{6.4}$$

where

$$x_0=-1 \tag{6.5}$$

and

$$w_{k0}=\theta_k \tag{6.6}$$

The activation function can be linear such as

$$\phi(v)=av \tag{6.7}$$

where a is some constant or it may be nonlinear such as the logistic function

$$\phi(v)=\frac{1}{1+\exp(-av)} \tag{6.8}$$

where a is the slope parameter. As a increases, the slope of the logistic function becomes steeper and in the limiting case, when a goes to infinity, the logistic function resembles a step function.

6.3 Network Architecture

The architecture of a neural network refers to the configuration of neurons and the connections between them. The neurons in a network are ordered in layers. The number of neurons in layer is not restricted nor are the number of rows. As the number of layers increases, the greater the capacity of the network. The first layer of the network is referred to as the input layer. This layer accepts the inputs to the network and passes them to the next layer. The connection between layers are referred to as weights. The weights are numbers that correspond to the strength of the connection between neurons. The input is propagated through the neuron via the weights to the output layer. The output layer consists of a number of neurons whose outputs represent the classes the input data has been separated into. If one output is much larger than the others, the data is considered to be classified to the class represented by that output. If the network consists of only two layers, the input and output layers, the network is referred to as a single-layer feedforward network. The term feedforward denotes that the input is propagated forward through the network. If the network contains more that two layers, the network is called a multilayer feedforward network. The layer between the input and output layers are called the hidden layers. Figure 6.1 gives a pictorial representation of a single layer network while Fig 6.2 shows a multilayer network.

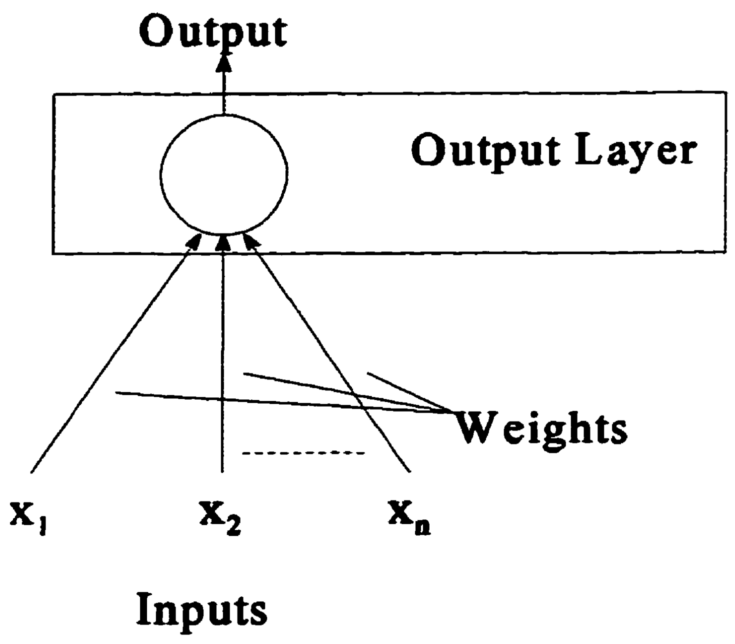


Fig. 6.1. Single layer neural network.

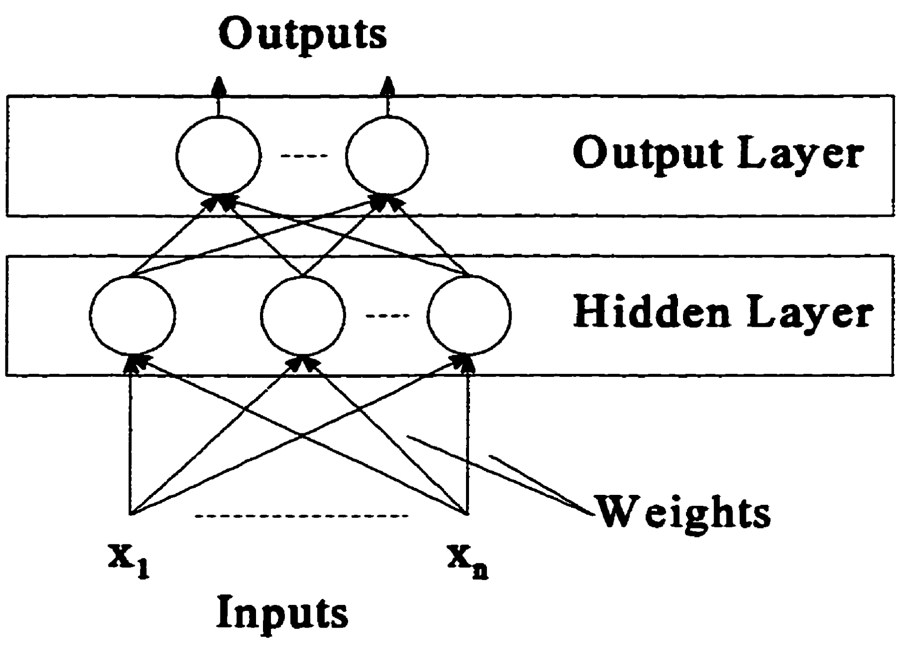


Fig. 6.2. Multilayer neural network.

It is noticed that every neuron contained in each layer is connected to every neuron in the next layer, this type of network is said to be fully connected. If some of these connections are removed, the network is then a partially connected network. Only fully connected multilayer networks are used in this study to classify the transients collected.

6.4 Learning

A neural network learns by showing it an input pattern and the desired output for the particular pattern and adjusting the network weights appropriately. The error between the desired output and actual output that the network computes determines the amount of change required in the network's weights. This type of learning is termed supervised learning since the desired output is provided, giving the network clues as to the right response. However, a means of adjusting the weights from the computed error must be found. The back-propagation training algorithm is the technique for propagating the error back from the output layer to the input layer, adjusting the weights as it progresses. The back-propagation algorithm begins by propagating an input to the output and recording the network output. The error between the output and desired output is measured by

$$E=\frac{1}{2}\sum_i e_i^2 \tag{6.9}$$

where

$$e_i=y_i-d_i \tag{6.10}$$

where *y* is the actual output and *d* denotes the desired output. The factor of a half is included for computational convenience while the square term ensures that the errors do not cancel one another out. With the error known, the network weights can be adjusted to minimize the error. If the error is differentiated with respect to the weight being adjusted, the gradient of the error curve at that weight is found. Figure 6.3 gives an example of such a curve. If the gradient is negative, the weight must be adjusted in the positive direction. On the other hand, if the error gradient is positive, the weight must be reduced. This technique is known as the gradient descent. The adjustment in the weight is proportional to the error gradient given by

$$\Delta w_{ij}=-\eta\frac{\partial E}{\partial w_{ij}} \tag{6.11}$$

where η is the learning rate which is described later. The negative sign in the equation is related to the above discussion on how to adjust the weight.

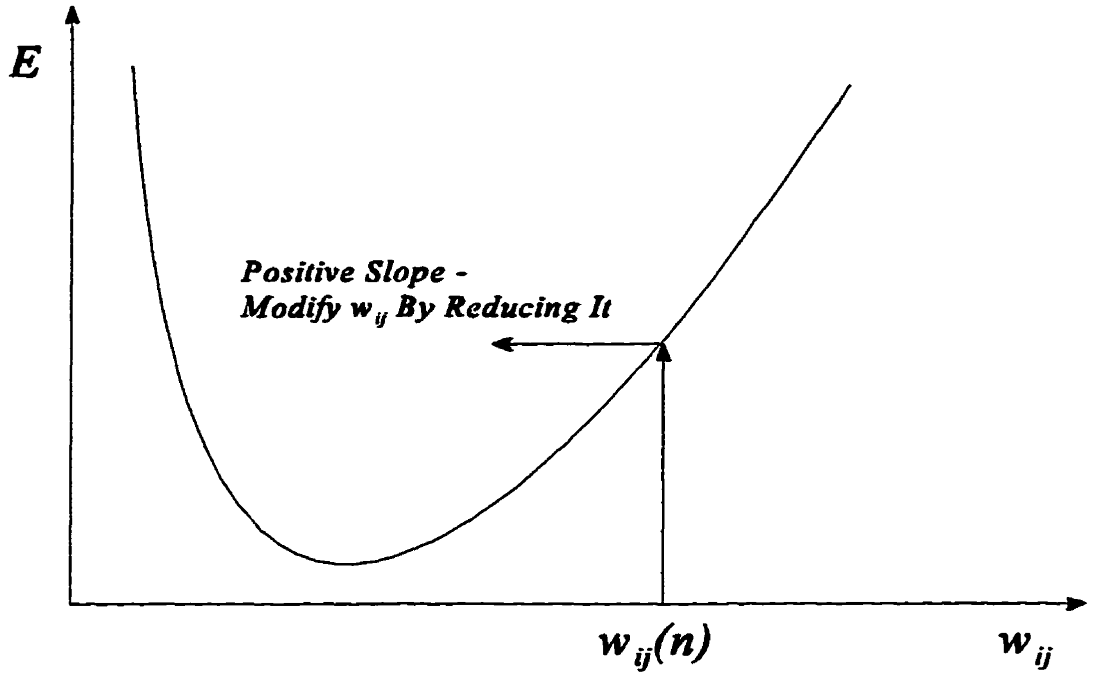


Fig. 6.3. Weight modification with respect to the error.

To evaluate the gradient, the chain rule must be employed since the gradient cannot be evaluated directly. The gradient is given by

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial v_i} \frac{\partial v_i}{\partial w_{ij}} \tag{6.12}$$

To evaluate the first derivative, the error is differentiated with respect to the network output i

$$\frac{\partial \frac{1}{2}\sum_{i}(y_i-d_i)^2}{\partial y_i}=y_i-d_i \tag{6.13}$$

The second derivative is evaluated by differentiating the network output by the input to the logistic activation function with *a* equal to 1.

$$\frac{\partial \frac{1}{1+\exp(v_i)}}{\partial v_i}=y_i(1-y_i) \tag{6.14}$$

The third derivative is found by differentiating the input to the activation function by the weight being adjusted

$$\frac{\partial \sum_{j} x_j w_{ij}}{\partial w_{ij}}=x_j \tag{6.15}$$

where x is an input if the network is single-layered and an output of the neuron one layer down if the network is multilayered. When the thresholds are being updated, the value of x is equal to -1 and the threshold is treated as a regular weight and updated accordingly. The above procedure only operates on the output layer and weights directly connected to output neurons. If the weights are connected to a hidden layer, the procedure must be modified since the error is propagated through all the neurons in the layer above. In other words, there are no desired responses for the hidden neurons, thus requiring the error to be determined recursively in terms of the errors of all the neurons in the layer above. The gradient given in this case is defined by

$$\frac{\partial E}{\partial w_{jk}}=\sum_i \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_i}\frac{\partial v_i}{\partial y_j}\ \frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{jk}} \tag{6.16}$$

where the first and second derivatives are calculated as above with the third derivative found by

$$\frac{\partial \sum_j w_{ij} y_j}{\partial y_j}=w_{ij} \tag{6.17}$$

where y is the output of the neuron connected by w to the neuron in the layer above. The forth and fifth derivatives are found in the same fashion as derivatives found in Eqs. 6.14 and 6.15, except that the weight being modified is connected to neurons in the hidden layer, not the output. As before, when dealing with thresholds, Eq. 6.16 is used with the input being -1. Now summarizing, the error gradient at an output neuron is found by

$$\frac{\partial E}{\partial w_{ij}}=(y_i-d_i)(y_i(1-y_i))(x_j) \tag{6.18}$$

and the gradient at a hidden neuron is

$$\frac{\partial E}{\partial w_{jk}}=\sum_i (y_i-d_i)(y_i(1-y_1))(w_{ij})\ (y_j(1-y_j))(x_k) \tag{6.19}$$

Once the gradient has been found for the appropriate weight, the weight modification can proceed. The weight adjustment is made by the following

$$w_{ij}(n+1)=w_{ij}(n)+\Delta w_{ij}(n) \tag{6.20}$$

With the technique for modifying the weights complete, the rate at which the network learns is still at issue. The learning rate, η controls how quickly the network converges on the minimal error. It should be noted that a high learning rate does not guarantee accelerated learning due to the large changes made to the weights and may in fact prolong the training time. A more appropriate approach is to set the learning rate at a more conservative level to ensure satisfactory training is achieved. If the learning rate is too small, training will occur but at the expense of training time. Another approach to accelerate learning is the application of momentum. Momentum uses the previous weight adjustment and adds it to the current adjustment. The mathematical expression of momentum is

$$\Delta w_{ij}(n)=\alpha\Delta w_{ij}(n-1)+\Delta w_{ij}(n) \tag{6.21}$$

where α is the momentum parameter. As the momentum parameter is increased, the effect of momentum is greater while decreasing it reduces the effect. If the signs of the weight adjustments are the same, the momentum from the previous adjustment increases the current weight adjustment.

6.5 Network Training

To train a neural network using supervised learning, a set of inputs and desired network response to these inputs must be generated. This set is referred to as the training set. Each input is presented to the network and is propagated through the network to the output.

The error is measured, and the weight and threshold corrections are made. However, a single pass through all the inputs usually does not constitute a complete training. The inputs must be presented to the network many times before training is complete. To test the quality of the training being performed, a separate set of input data is set aside to test the network. The testing set is usually very small, roughly 10 percent of the total inputs used for training. This data is used exclusively for testing the network and is never included in the training set. As with the training data, the testing data has the desired outputs determined in order to measure the network error. Since this testing data is not included in the training of the network, the use of the testing data is an objective means of measuring the network's ability of generalize. The testing data can also be used to terminate training. When the average output error of the training data falls below a certain point, the training is discontinued.

6.6 Software Implementation

Before the actual training can begin, the weight matrices, threshold vectors, and neuron output vectors must be allocated. The weight matrix dimensions are governed by the number of neurons directly connected by the weights. The number of rows of the weight matrix are equal to the number of neurons above the weights while the number of columns is equal to the number of neurons below the weights. The threshold vector sizes are equal to number of neurons contained in their respective layer. The same goes for the neuron output vectors which record the output computed by each neuron. Once the memory for the data structures have been allocated, the software reads an input file to determine whether it is to train or classify data. If the network is to classify data, the network weights and threshold are read from a file and loaded in their appropriate location. However, if the training mode is selected, the weight matrices and threshold vectors are loaded with small random numbers. Propagation of an input to the hidden layer requires the application of Eqs. 6.1 and 6.2. The inputs are read from a file and stored in a matrix. The size of the matrix is dependent on the network's mode. If the network is in a training mode, the matrix only has one row, if the network is in classification mode, the number of rows corresponds to the size of the training and testing sets combined. Each neuron and its corresponding row of weights in the weight matrix are multiplied with appropriate input. The neuron's threshold is then subtracted from the sum computed by applying Eq. 6.1. Equation 6.2 is evaluated using the logistic function as the activation function. Each neuron output is stored in the hidden layer output vector. The propagation continues by evaluating the output layer's outputs by the above technique with the inputs to the output layer being the outputs of the hidden layer. The outputs of the output neurons are recorded in the output layer's output vector. If the network is in a classification mode, the outputs of the output neurons are reported to the user and written to a file for later analysis. If the network is in a training mode, the propagation of the inputs works in conjunction with the back-propagation training algorithm. The input propagation is responsible for determining the network errors used by the training algorithm to modify the weights. The back-propagation training algorithm with momentum is used to modify the weights according to the error found. Training is continued until the average error of each output is found to be about 5 to 8 percent. If this error reduction target cannot be met, the algorithm will be suspended after 25,000 iterations. Once the training algorithm has been terminated, all the weights and thresholds are written to a file for later use in classification.

The format of input files required by the software depends on whether the network is to classify data or to be trained. If the network is to be trained, the first line of the input file contains a 1, followed by the network parameters such as the number of inputs, number of hidden layers and outputs, size of training set, and size of testing set. Following the network parameters, the training set is defined by listing the file name and desired outputs of the entire set. To complete the file, the testing set is defined in the same manner as the training set. If the network is to classify data, the first line contains a 2, followed by the network size parameters and the number of inputs to be classified. Following the parameter list, the file names of the data to be classified are listed.

6.7 Summary

This chapter described the neural network used to classify the radio transmitter transients. The topology of the neural network implemented for this study was discussed as well as the back-propagation training algorithm used to train the network. The back-propagation training algorithm uses the error between the desired output and the network computed output to determine the modifications made to the network's weights and thresholds. The next chapter describes the results obtained that show that the neural network can be a useful tool in classifying radio transmitter transient features constructed and selected by wavelet analysis and genetic algorithms. There are other neural networks that may also be good candidates for classification. For example, a probabilistic neural network is being implemented by Kinsner's research group.

CHAPTER 7
EXPERIMENTAL RESULTS

By combining the techniques described in the previous chapters, a means of identifying radio transmitter transients is achieved. The process begins by acquiring the radio transmitter transient, using a radio receiver and Sound Blaster, followed by separating the transient from the noise by using the variance dimension. Once the transient has been separated, the genetic algorithm in combination with multiresolution analysis selects the critical features of the transient to be used in classifying the transient or training a neural network. If training is required, the transient features are added to a training or testing file to "teach" the network to identify the transient's features. If the transient is to be classified, a trained neural network is shown the transient's features and the neural network indicates which transmitter it believes generated the transient based on the features presented to it.

7.1 Transient Acquisition

Initially, we have performed transient acquisition of slow transmitters, including Yaesu, Kenwood, and Radio Shack. Later, the transients used in this study were acquired by the *Communication Research Centre* (*CRC*) in Ottawa. Our acquisition software was provided to the *CRC*, and used to record transients generated by six low power transmitters in a controlled environment. A ICOM R7100 communication receiver was connected to a Sound Blaster in the fashion described in Chapter 2. The transmitters were connected to the receiver via an 89 foot length of RG-58U coaxial cable with a system of variable attenuators inserted in series within the length of the transmission line. The attenuators were used to keep the input signal to the receiver at approximately -70dBm. With the connections made at the receiver and transmitter, the transmission line was routed along an outside wall of a room to diminish the effects of coupling between the transmitters and the measuring equipment. To activate the transmitter's push-to-talk button, each transmitter was modified to enable an operator to remotely operate the push-to-talk button without physically touching the transmitter. The push-to-talk control line was routed beside the coaxial cable connecting the transmitter to the receiver. To initiate the acquisition, the squelch level on the receiver is set to a point such that the speaker output is quiet and the receiver is set to the appropriate transmitter frequency, followed by the acquisition software being started. Next, the acquisition software waits until a transient event occurs which would be triggered by the operator remotely activating the push-to-talk button. When the transient has been detected, the circular transient buffer is written to the disk to be operated on by the next stage. In all, 50 transients were collected from each transmitter, bringing the total transient collection to 300. Table 7.1 lists the transmitters and their designated transmission frequency.

Table 7.1. Transmitters used to collect transients.

Transmitter	Model	Frequency (MHz)
Kenwood 1	TH25AT	147.000
Kenwood 2	TH25AT	147.000
Kenwood 3	TH25AT	147.000
Kenwood 4	TH21AT	147.000
Yaesu 1	Unknown	Unknown
Yaesu 2	FT208R	147.600

An example of each transient is presented in the next section describing the noise separation technique.

7.2 Noise Separation

Since the transient buffer contains a significant number of noise samples, a means of separating the noise from the transient is required. The variance dimension described in Chapter 3 is employed to determine the point at which the transient begins. A window size of 512 samples and a window offset of 1 is used to localize the beginning of the transient.

7.2.1 Kenwood 1 Transmitter

Figure 7.1 shows the recorded transient buffer for the Kenwood 1 transmitter.

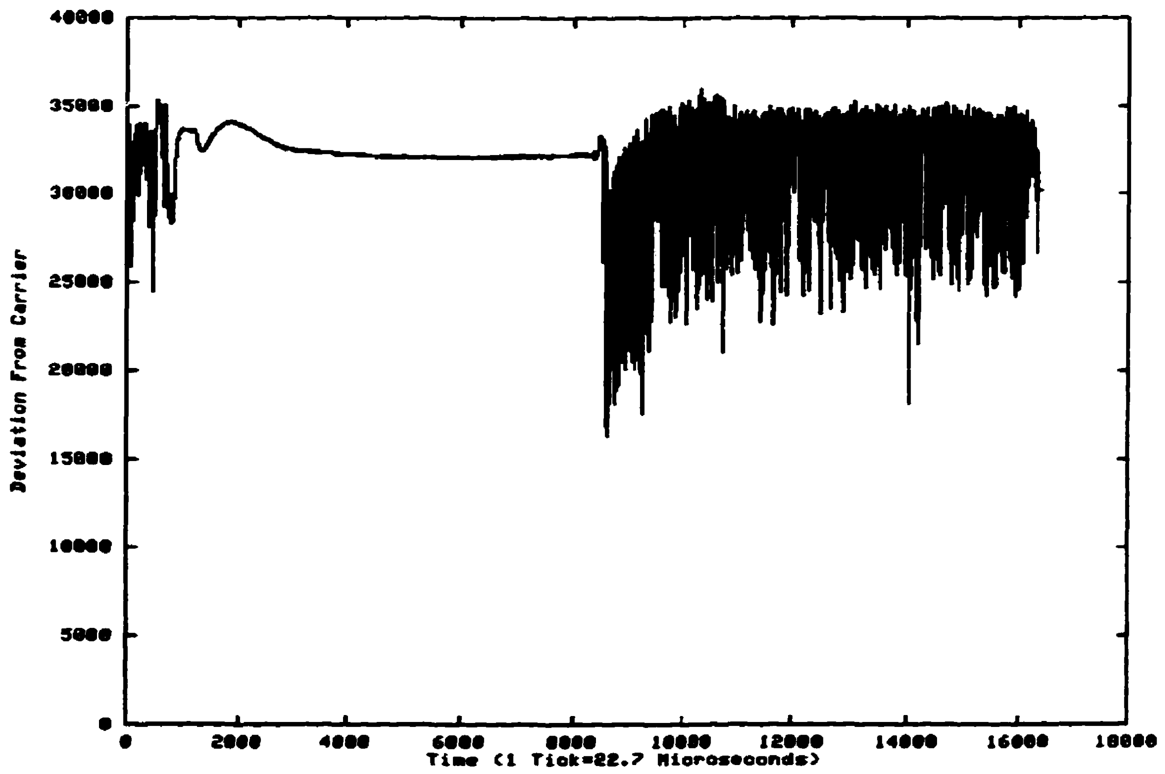


Fig. 7.1. Recorded Kenwood 1 transient and noise.

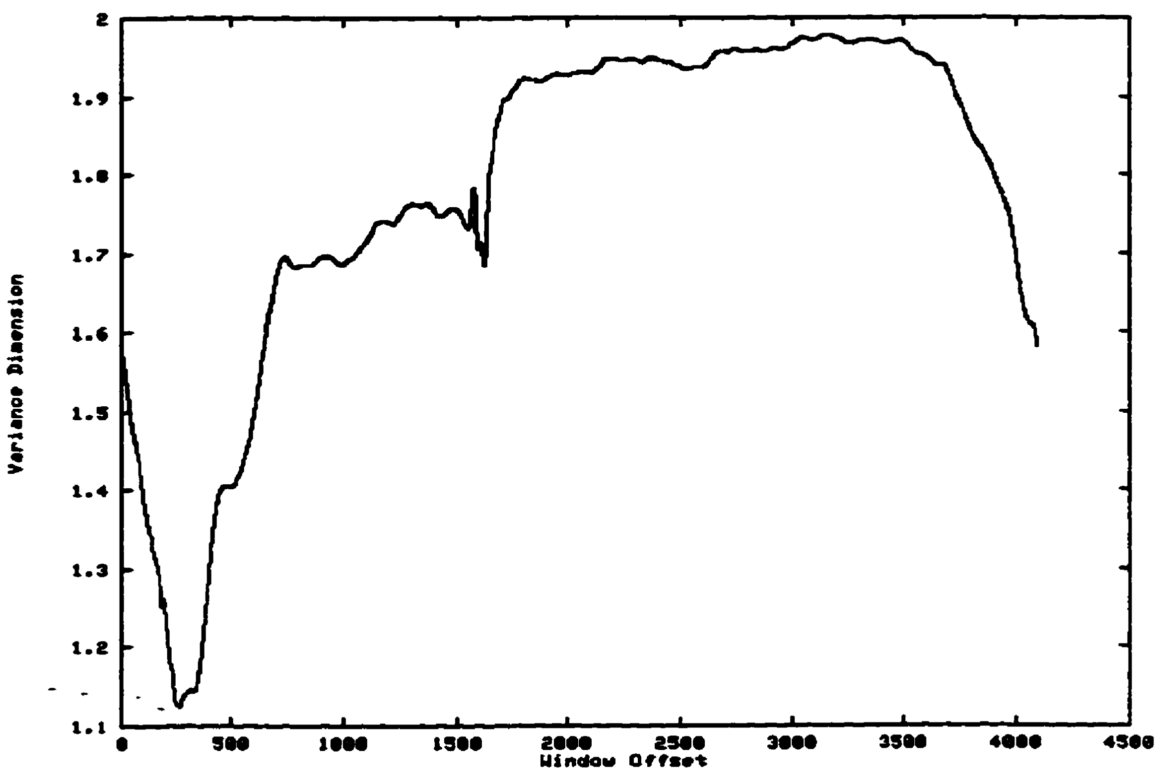


Fig. 7.2. Variance dimension for Fig 7.1.

It is noticed that the transient begins at about the 1,000th sample and continues until about the 8,400th sample. Figure 7.2 shows the variance dimension found for the signal in Fig. 7.1. The x axis is the number of samples the window in which the variance dimension calculated is offset from the first sample in Fig. 7.1. It should be noted that the number of samples used for the variance dimension is reduced to 4,096 by averaging each group of 4 samples, thus the offset ranges from 0 to 4,096. The first variance dimension calculation yields a result of approximately 1.57. This is due to the fact that the variance dimension window contains both noise and transient samples. As the window is further offset, the dimension beings to drop to a minimum value. The minimum value is found when the window is offset by approximately 250 samples. Multiplying the offset of 250 by 4, taking the averaging into account, localizes the transient at about the 1,000th sample. This corresponds to the location of the transient found in Fig 7.1. As the offset continues to increase, the variance dimension continues to increase until the window contains all noise at a window offset of 2,100 samples. Again, multiplying 2,100 by 4 localizes the noise at 8,400th sample in Fig. 7.1. The dimension begins to drop once again near the offset of 3,500 as the windows wraps around to the beginning of the samples. By wrapping the window, a complete picture of the variance dimension of the noise-transient signal is constructed. It is also noted that the transient contained in Fig. 7.1 does not contain 8,192 samples. In fact, it contains only 7,400 samples. This phenomenon is due to the operator not completely depressing the push-to-talk button. By momentarily pressing the push-to-talk button, the transient event occurs, however, the transmitter quickly discontinues communication and releases the channel. This leads to a larger noise component as shown in Fig. 7.1. In normal operation, the user holds down the push-to-talk button long enough (186 milliseconds) to collect the 8,192 samples used to analyze the transient. However, most transients collected for the Kenwood 1 transmitter do not exhibit this type of phenomenon.

7.2.2 Kenwood 2 Transmitter

The next transient of interest is generated by the Kenwood 2 transmitter. Figure 7.3 shows the transient and noise components as recorded by the acquisition software. The transient contained in Fig. 7.3 exceeds the 8,192 samples required to analyze the transient and begins at roughly the 7,600 sample. Figure 7.4 shows the variance dimension for the signal in Fig. 7.3.

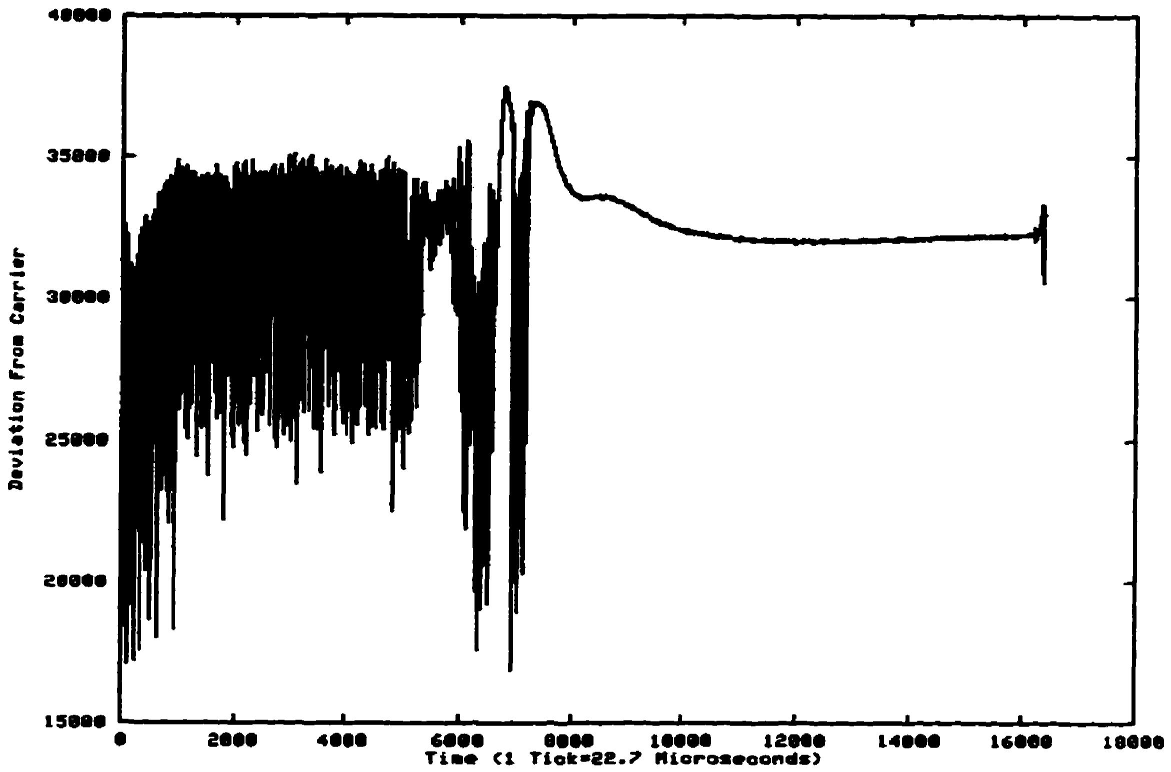


Fig. 7.3. Recorded Kenwood 2 transient and noise.

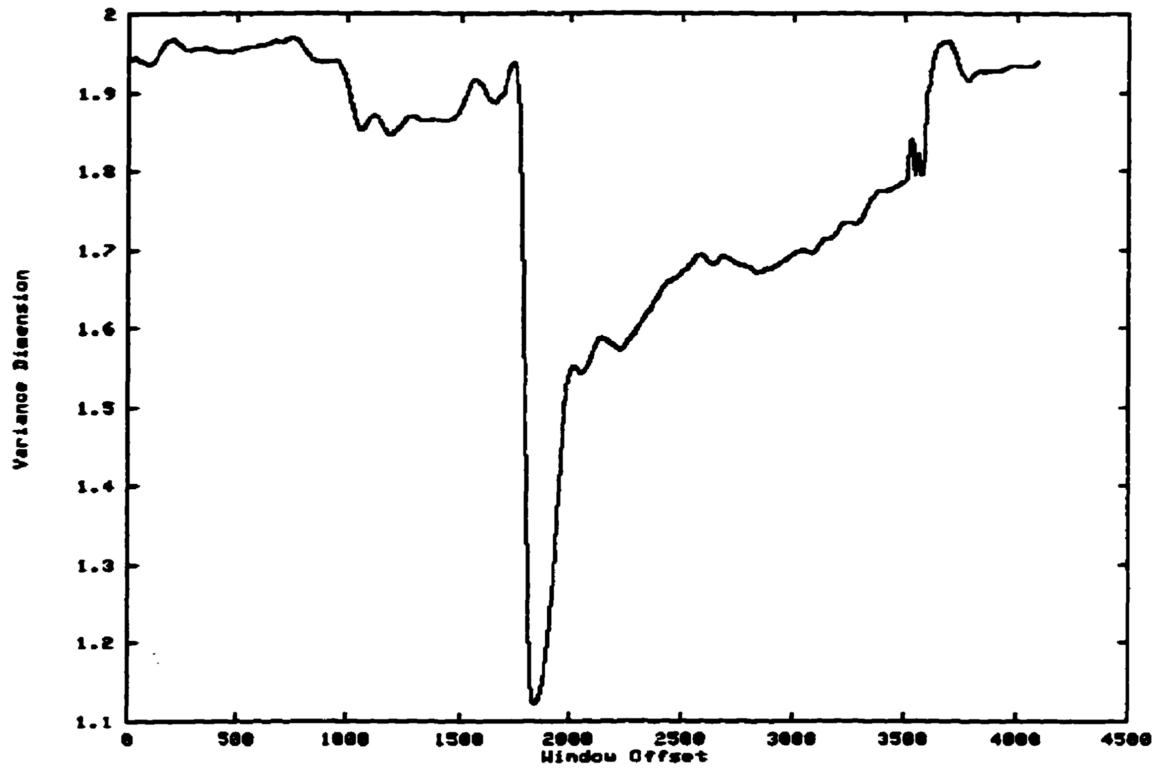


Fig. 7.4. Variance dimension for Fig 7.3.

In Fig 7.4 the variance dimension drops significantly at the window offset of 1,850 with the minimum dimension found at about 1,900. The dimension begins to climb after the beginning of the transient has been localized since the transient itself contains a significant noise component superimposed upon it. When the window reaches an offset of 8,000, the variance dimension calculation begins to measure the superimposed noise and the dimension increases. However, between the beginning of the transient and the offset of 8,000, the changes between samples is greater, giving the impression of improved correlation between samples. It should be noted that there still exists a superimposed noise component in this region but its effect on the variance dimension is diminished. Figure 7.5 shows the transient separated from the noise.

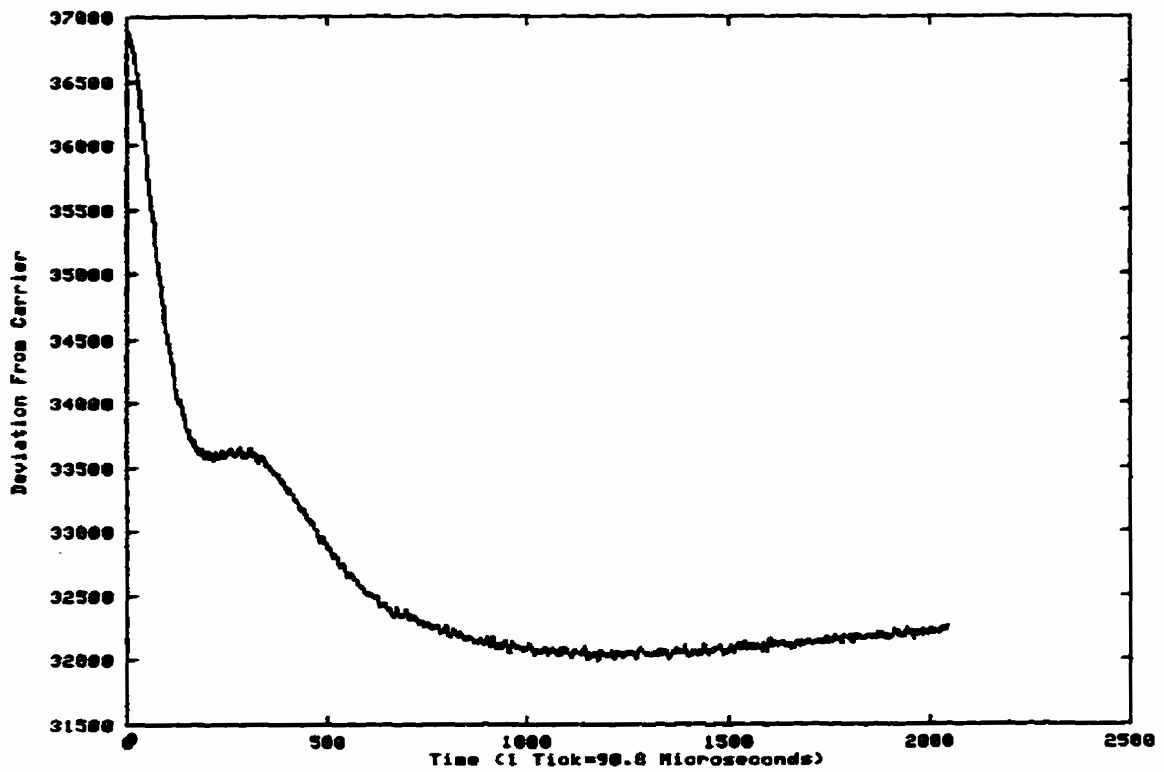


Fig. 7.5. Kenwood 2 transient separated from noise.

The transient separated from the noise begins at the top of the second peak in Fig. 7.3. Since there is a heavy noise component between the two peaks, the variance dimension does not drop at the location of the first peak.

7.2.3 Kenwood 3 and 4 Transmitters

In order to confirm the operation of the variance dimension noise separation, a third transmitter transient is analyzed. Figure 7.6 shows the recorded noise and transient when the Kenwood 3 transmitter is activated.

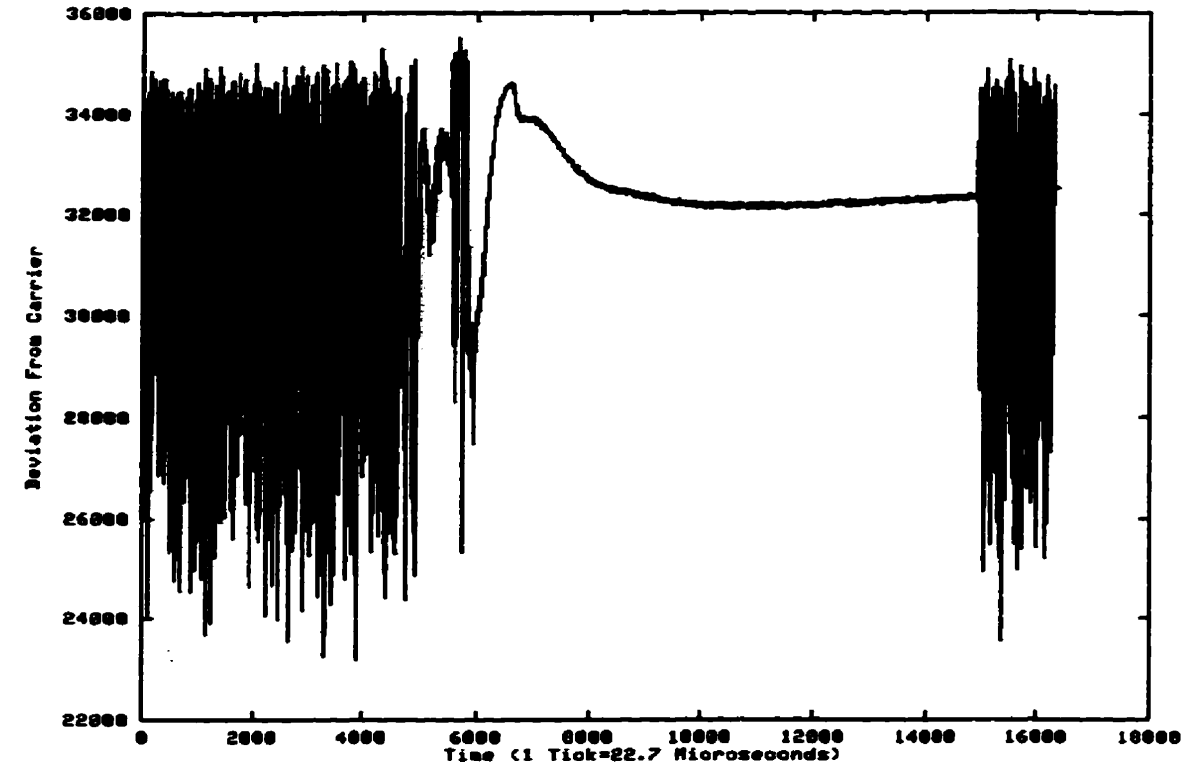


Fig. 7.6. Recorded Kenwood 3 transient and noise.

The separated transient is shown in Fig. 7.7. Again, the minimum dimension is used to localize the onset of the transient. At about the 5,000th sample in Fig. 7.5, the signal begins to look somewhat correlated. However, the dimension is slow to drop before it reaches a minimum at approximately the 6,100th sample. It is also noticed that the Kenwood 2 and Kenwood 3 transmitters are the same model type with very distinct transients. The variability of transients between the same model types is exactly what is required to accurately classify the transients. The variability also exists between different model types as shown in Fig. 7.8. Figure 7.8 contains the transient generated by the Kenwood 4 transmitter with the model designation TH21AT as opposed to the Kenwood 2 and Kenwood 3 which are TH25AT models.

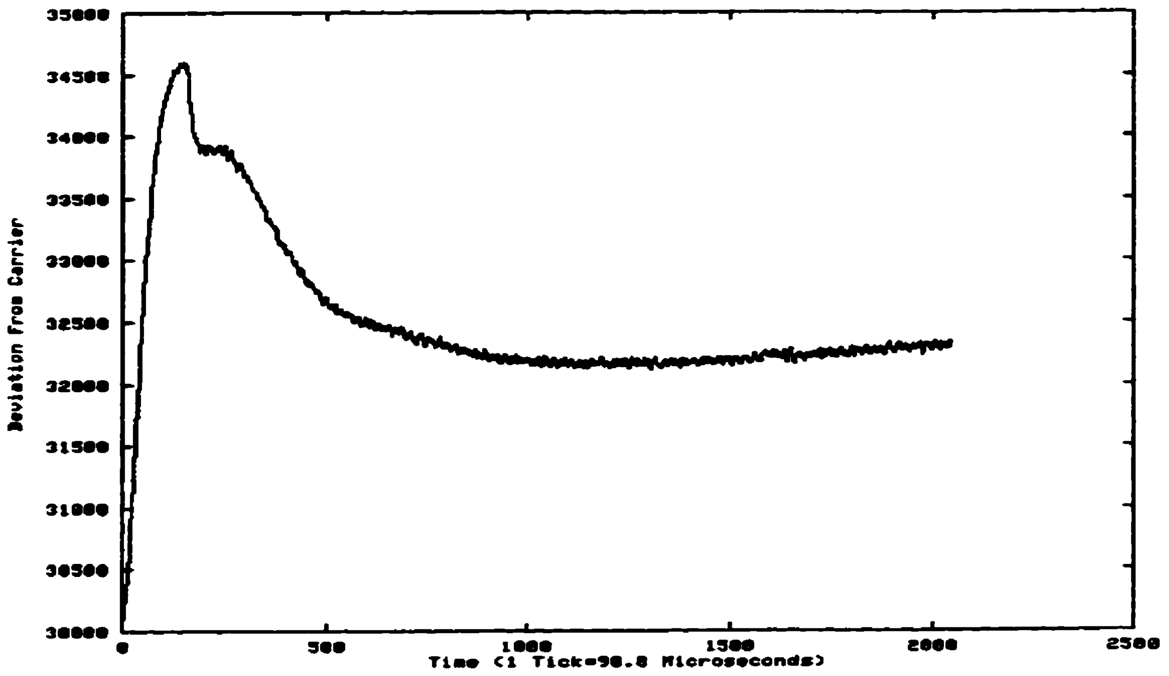


Fig. 7.7. Kenwood 3 transient separated from noise.

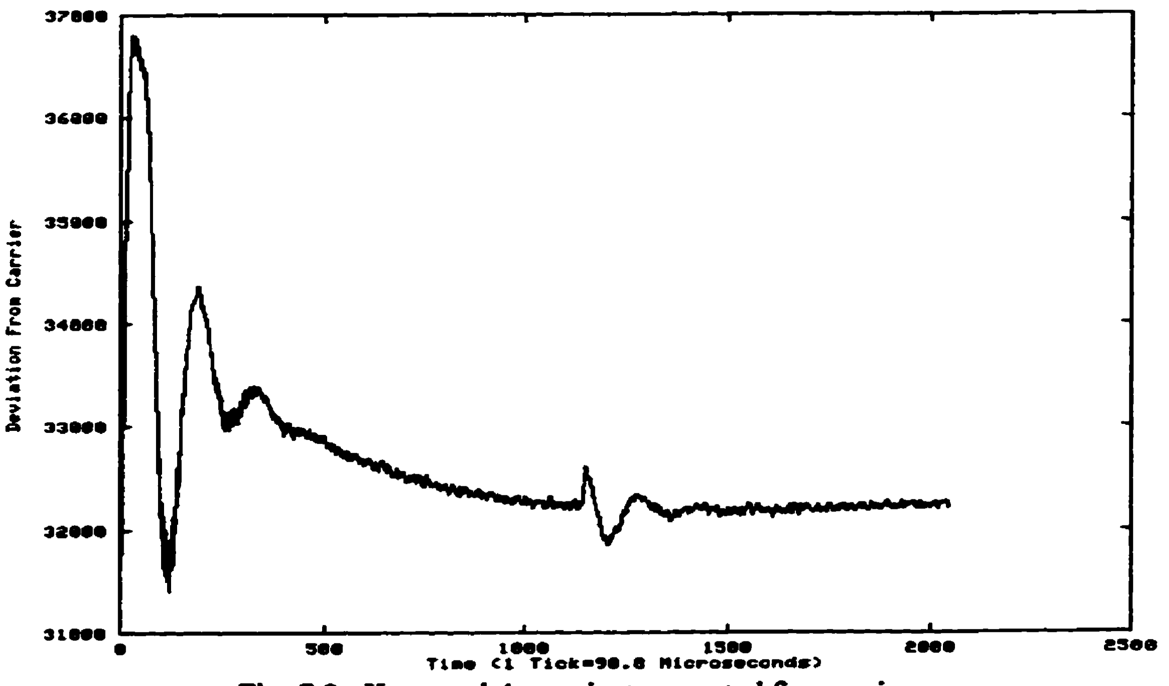


Fig. 7.8. Kenwood 4 transient separated from noise.

7.2.4 Yaesu Transmitters

It is also noticed that transients generated by transmitters of different manufacturers have unique transient features as shown in Fig. 7.9. Figure 7.9 is a transient generated by a Yaesu FT208R designated as Yaesu 2.

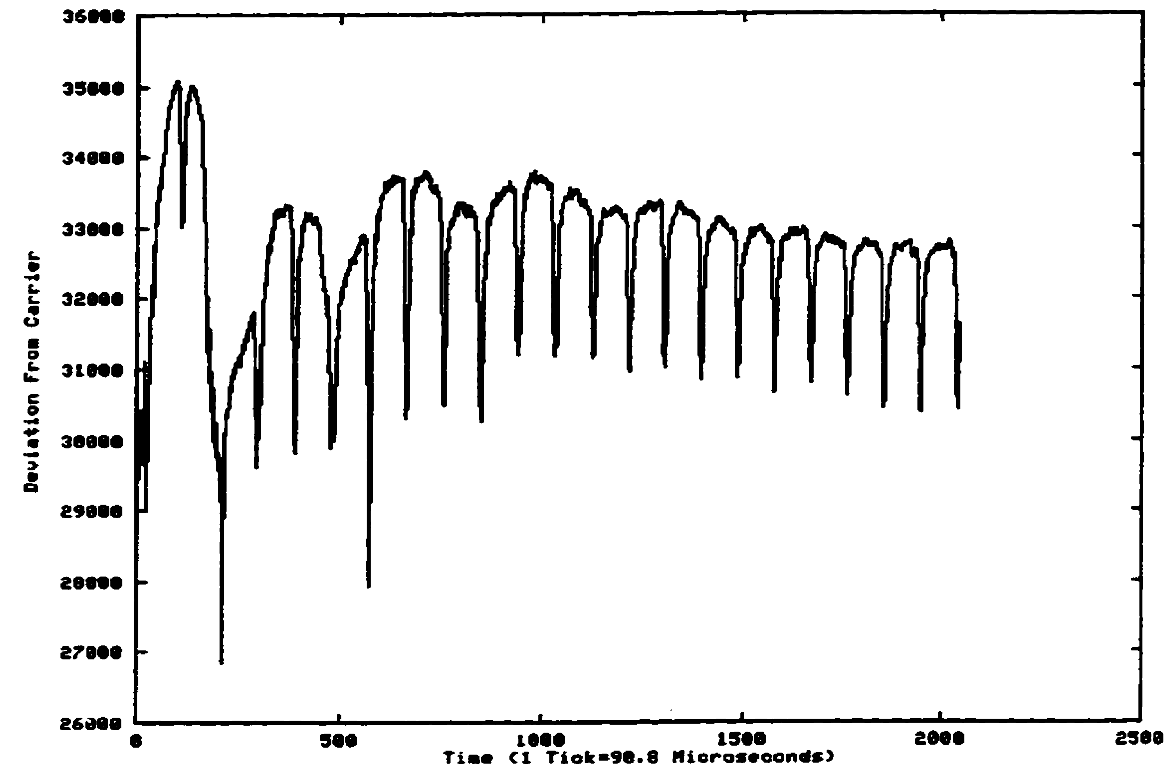


Fig. 7.9. Yaesu 2 transient separated from noise.

The transient generated by the Yaesu 2 requires a different means of localizing the transient. Since the variance dimension analysis does not have a definite minimum dimension, the maximum negative slope is used. If the minimum dimension is selected, the first 250 samples in Fig. 7.9 are truncated which eliminates a significant feature for use in classifying the transient. Figure 7.10 shows the variance dimension plotted versus the window offset. Since a plateau exists, the slope before the plateau is used to localize the transient. By using this

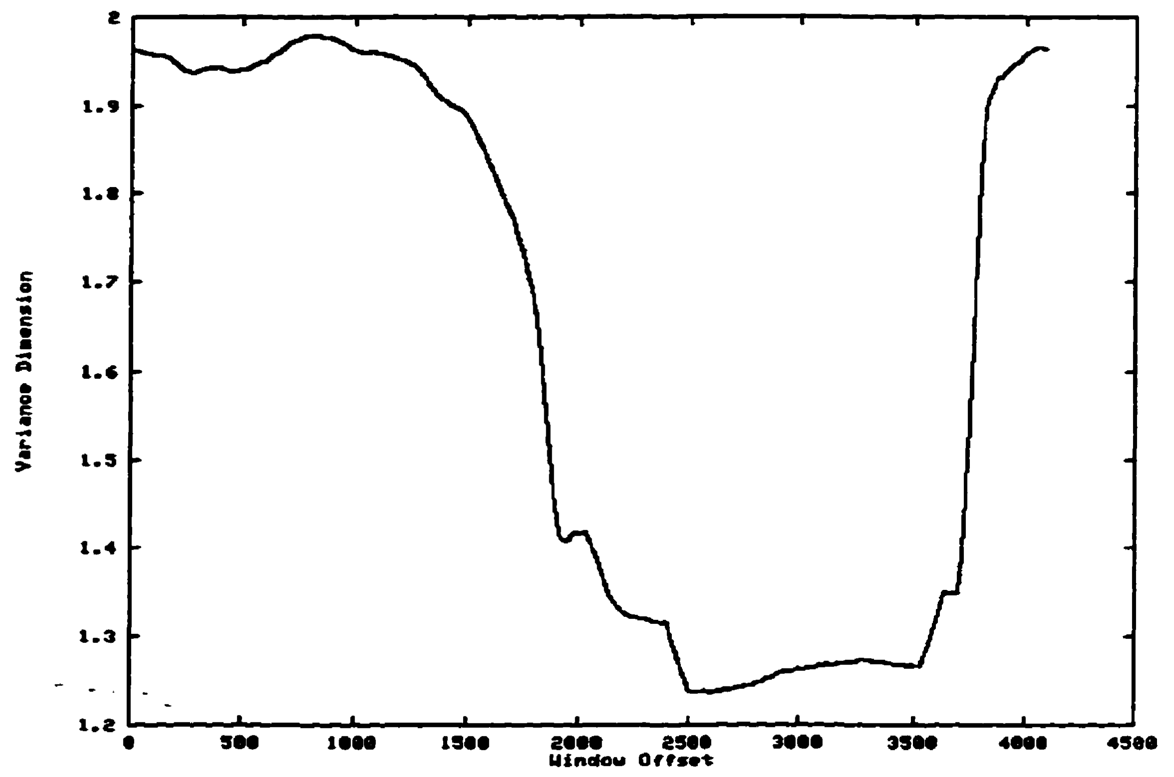


Figure 7.10. Variance dimension used to localize **Yaesu 2** transient.

analysis, the transient in Fig. 7.9 is localized containing all the significant features. The lack of a definite minimum dimension is due to the fact that the transient contains a quasi-periodic component starting at the 1,000th sample. Since this portion is highly correlated and dominates the superimposed noise, the variance dimension remains low unlike the Kenwood generated transients that do not contain the correlated portion.

7.3 Transient Classification

With the transients separated from the noise, the transients are passed to the genetic algorithm for feature selection. The wavelet coefficients that represent the significant features are selected by the genetic algorithm and written to a file. These files containing the selected features are used to either train a neural network or classify the transient. The neural network topology used to classify the transients consists of 64 inputs, 12 hidden neurons and *n* outputs, where *n* is the number of classes the transient can be classified to. Each class is labeled with a transmitter name, indicating which transmitter was responsible for the generation of the input transient. The first network trained contained 6 outputs to classify the four Kenwood and two Yaesu generated transients. The number of coefficients selected by the genetic algorithm is 32. The use of more than 32 coefficients does not improve classification due to the fact that it has been observed that wavelet coefficients representing insignificant features tend to be selected in this case. Therefore, of the 64 neural network inputs, 32 have non-zero coefficient inputs while the others are zero due to the fact that the feature represented by that coefficient was not selected. Six transients from each class are selected to be representative of the entire class. A transient is said to be representative of the entire class if it contains transient features that are consistent with the significant features observed within the class. It is also desired that transients with features that are not consistent with the features of the class be classified as well. Therefore it is necessary to include examples of transients that contain features that are not consistent with the class. An additional two transients are selected to test the network's ability to generalize. The training is terminated when the average error per output is less than 0.05 for the testing patterns. For the set of transients with 32 selected coefficients, 2,000 modifications to the weights were required to train the network. It was noted that all the training patterns and testing patterns were correctly classified by neural network. However, as described in the first section of this chapter, some transients are not correctly generated and must be discarded. Table 7.2 lists the number of discarded transients per class.

Table 7.2. Number of discarded transients per class.

Class	Transients Discarded
Kenwood 1	9
Kenwood 2	0
Kenwood 3	3
Kenwood 4	4
Yaesu 1	0
Yaesu 2	0

The transients which are not discarded are presented to the trained network for classification to determine the performance of the network. Table 7.3 summarizes the results. A transient is said to be correctly classified if the output corresponding to the transmitter that generated the transient is greater than 0.50 and the rest of the outputs are below 0.20. The output excitation thresholds for correct classification are based on heuristics and it will require additional study to determine whether these values are optimal. The closer the output excitation value is to 1.0, the greater confidence the network has in its classification.

Table 7.3. Classification results.

Class	Transient Presented	Correctly Classified	Percentage Correct
Kenwood 1	35	29	83%
Kenwood 2	44	40	91%
Kenwood 3	41	38	93%
Kenwood 4	40	40	100%
Yaesu 1	44	43	98%
Yaesu 2	44	44	100%

It is noticed in Table 7.3 that the neural network performed quite well. The average percentage correct is 94% with the only class significantly lower than the average being the Kenwood 1 class. This difficulty is due to the variability in the transients that the Kenwood 1 transmitter provided. To combat this situation, an additional two transients were added to the training set for the Kenwood 1 transmitter. By including two additional training examples, it is hoped that the network will be able to improve upon the number it correctly classifies. The network was trained as before with the training set for the Kenwood 1 transmitter modified. As before, about 2,000 modifications to the weights were required to train the network to the same error tolerance as before. Once the network has been trained, the remaining transients are presented to the network to assess the network's classification performance. Table 7.4 summarizes the results.

Table 7.4. Classification results.

Class	Transient Presented	Correctly Classified	Percentage Correct
Kenwood 1	33	30	91%
Kenwood 2	44	42	95%
Kenwood 3	41	36	89%
Kenwood 4	40	40	100%
Yaesu 1	44	43	98%
Yaesu 2	44	44	100%

The results presented in Table 7.4 indicate that the number of correctly classified transients has increased in two classes while decreasing in one. The average percentage correct has increased to 96% from 94% by increasing the number of training examples for the Kenwood 1 class. Thus, by ensuring that the training set is representative of the class, the accuracy of the network can be increased.

From Table 7.3 and 7.4 it is noticed that the neural network is able to classify transients generated by different manufacturers, different models, and transients generated by the same manufacturer and model. The Kenwood 1 through Kenwood 3 transmitters are all the same model type while the Kenwood 4 is made from the same manufacturer, but is a different model. The neural network in each case is able to distinguish between each Kenwood class with a great deal of accuracy. This indicates that the transients generated by each of the Kenwood transmitters are unique. This is necessary to be able to pinpoint individual transmitters that may be operating in an inappropriate manner. It is also noted that the neural network is able to distinguish between different manufacturers (Kenwood and Yaesu) with a high level of accuracy. This is expected since different manufacturers use differing designs of frequency synthesizers to generate the carrier frequency. Thus, using 32 coefficients to describe the transients significant features, the network is able to distinguish between transients generated by different manufacturers, same manufacturer but different models, and same manufacturer and model type.

It is also of interest to determine whether fewer wavelet coefficients can be used to classify the transients. The previous experiments used a total of 32 coefficients with good classification results. The next experiment determines whether a total of 24 coefficients will yield the same results. Again, as with the previous experiments, a neural network is trained. However, this time, the training and testing sets consist of the Kenwood transients with 24 selected wavelet coefficients. The network was trained to the same tolerance as the first two networks with each training and testing pattern correctly classified by the neural network.

The classification results are extremely poor as compared to previous networks trained with 32 transient features. The Kenwood 1 class of transients were classified correctly less than 50% of the time. The Kenwood 2 class was classified correctly with about 50% accuracy with the other two classes performing better. The Kenwood 3 and Kenwood 4 classes were classified correctly more than 50% of the time but not near the 90% the previous networks performed at. Figure 7.11 shows the average network outputs for the networks trained on 24 wavelet coefficients and 32 when the remaining Kenwood 1 transients were presented to the network.

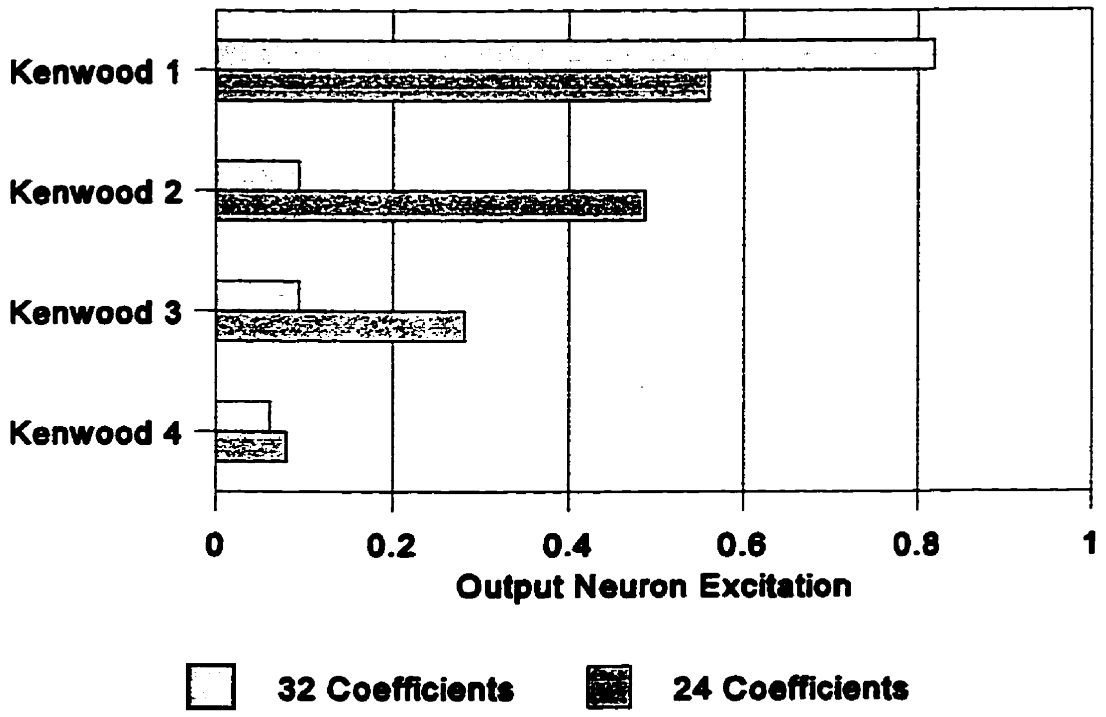


Fig. 7.11. Network outputs when presented Kenwood 1 transients.

Figure 7.11 shows the poor separation between the Kenwood 1 and Kenwood 2 classes when using 24 wavelet coefficients. The average outputs corresponding to the Kenwood 1 and

Kenwood 2 are near equal, indicating that the network is unable to correctly classify the Kenwood 1 transients. Figure 7.12 shows the network outputs when the Kenwood 2 transients are presented to the network. It is noticed that the Kenwood 2 output is greater than the rest but confusion between the Kenwood 2 and Kenwood 3 classes did exist. As with the Kenwood 1 transients, more than 24 wavelet coefficients are needed to classify the transients correctly. Figure 7.13 shows the outputs of the networks when the Kenwood 3 transients are presented.

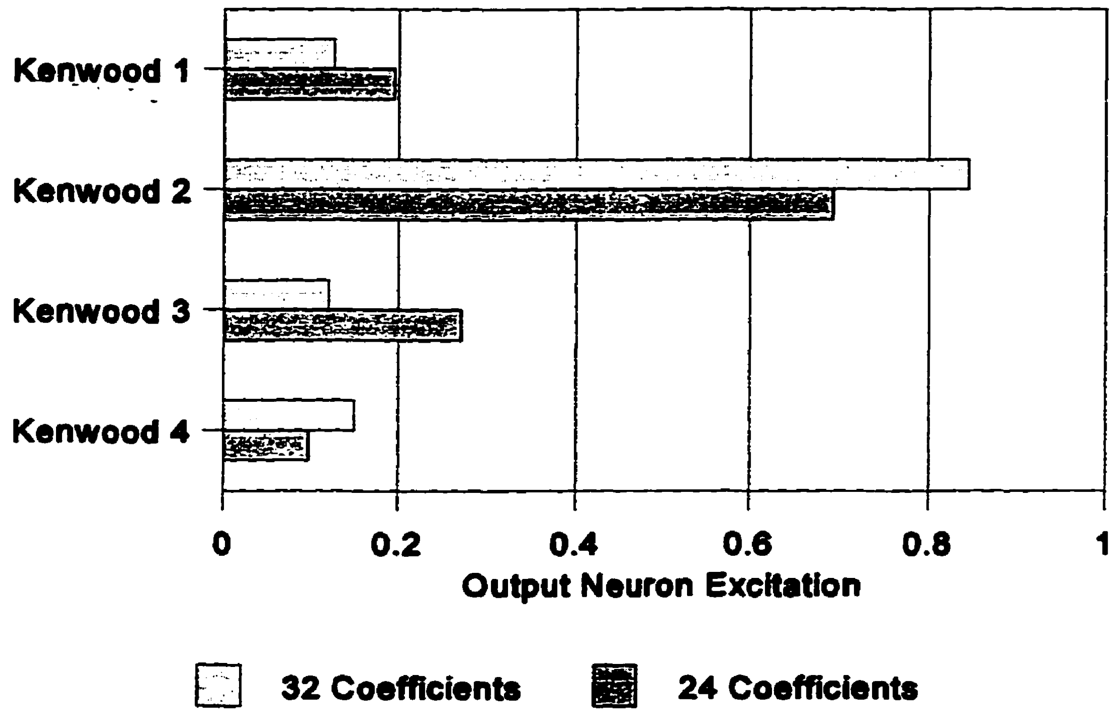


Fig. 7.12. Network outputs when presented Kenwood 2 transients.

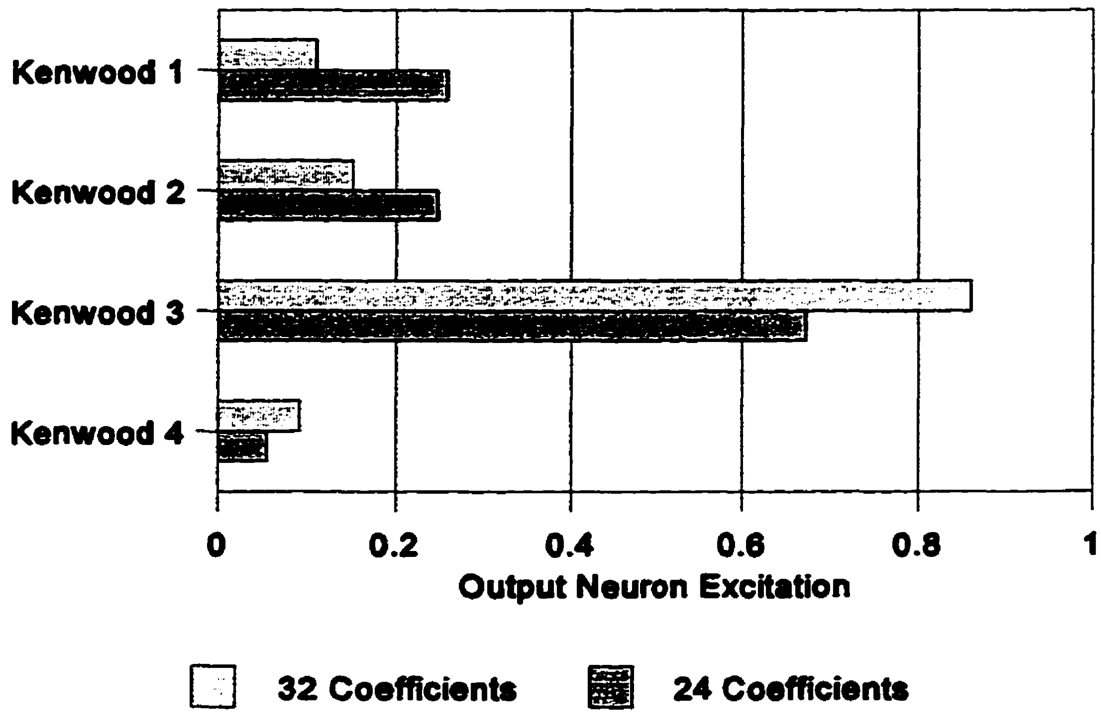


Fig. 7.13. Network outputs when presented Kenwood 3 transients.

Figure 7.13 is quite similar to Fig. 7.12, the correct output is significantly higher than the rest. However, the network trained on 32 coefficients correctly classified more transients than did the network trained on 24. The network trained on 24 coefficients tended to classify the Kenwood 3 transients as either Kenwood 1 or Kenwood 2 when an error in classification was made. This phenomenon indicates that the Kenwood 1, Kenwood 2, and Kenwood 3 share common characteristics, thus requiring more features to separate the classes. Finally, Fig. 7.14 shows the network outputs when the Kenwood 4 transients are presented.

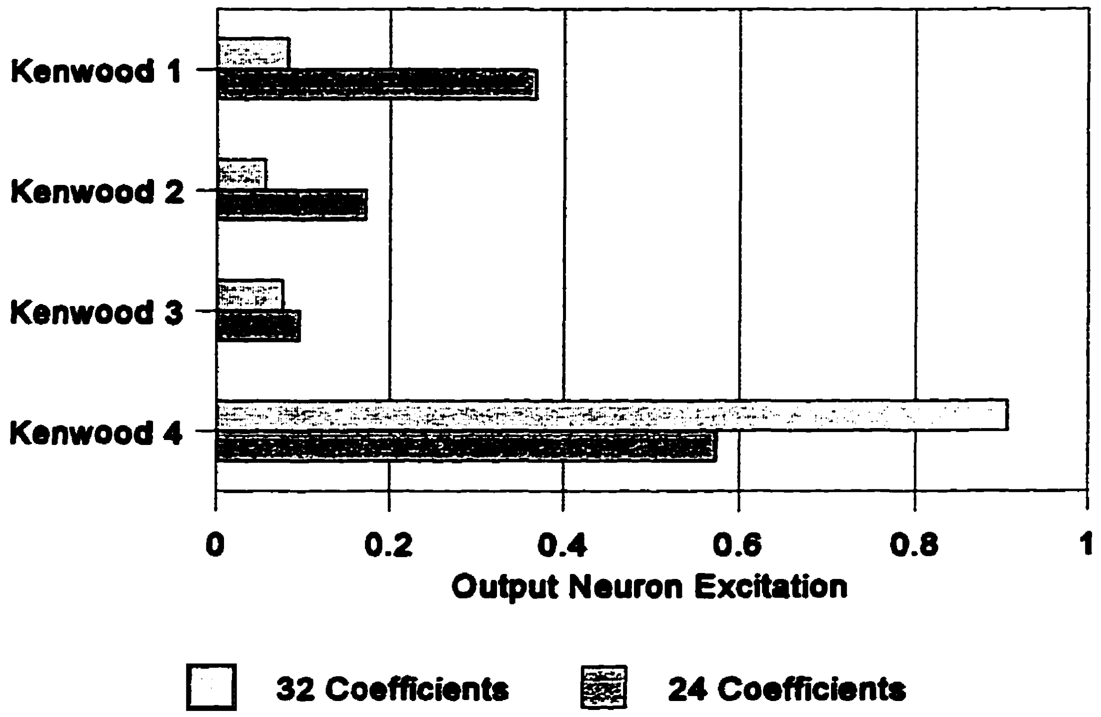


Fig. 7.14. Network outputs when presented Kenwood 4 transients.

Figure 7.14 indicates that the network had difficulties distinguishing between the Kenwood 1 and Kenwood 4 classes. Again, this indicates that the Kenwood 1 and Kenwood 4 classes have similar features and require more wavelet coefficients to separate the classes.

In order to classify the transients accurately, a network should be trained on 32 wavelet coefficients, as the smaller number of coefficients (24) does not provide enough features to separate the classes. When 32 coefficients are used, the classification of the transient classes are very accurate.

CHAPTER 8
CONCLUSIONS AND RECOMMENDATIONS

8.1 Conclusions

This thesis presented a means of recording, processing, and classifying low power radio transmitter transients. The acquisition portion of the system provided excellent recordings of the transients as shown in the figures contained in Chapter 7. The noise separation performed by the variance dimension technique separated the transients with good accuracy as shown again in the figures in Chapter 7. The consistency of the separation can also be confirmed by the average correct classification rate of 96%. It also can be concluded that at least 32 wavelet coefficients representing the signal features are required to accurately classify the transients. When 32 coefficients are used, the maximum classification rate achieved by the neural network was 96%. The average classification rate also depended on the size of the training set. A training set of transients per class provided a classification rate of 94%, while expanding the Kenwood 1 training set to 8 transients to better represent the transients contained in the class provided a classification rate of 96%. It also can be said that the network was able to distinguish between transients generated by transmitters built by differing manufacturers as well as the same manufacturer. The neural network was also able to distinguish between transients generated by the same manufacturer and model type. The network was very accurate in classifying the Kenwood 1, Kenwood 2, and Kenwood 3 transients generated by transmitters of the Kenwood TH25AT model type. If the number of wavelet coefficients is reduced to 24, the neural network lost its ability to distinguish between the Kenwood classes of transients. When the number of coefficients is dropped to 24, the average neural network outputs corresponding to the correct class dropped to output levels of incorrect classes.

8.2 Contributions

The following is a list of contributions believed to be made during the completion of this thesis.

a. A system for recording transients at a sampling rate of 44,100 samples per second and 16 bits per sample accuracy.

b. The implementation of the variance fractal dimension trajectory in transient-noise separation.

c. The use of genetic algorithms for transient feature extraction from wavelet coefficients.

d. Classification of transients generated by low power transmitters built by different manufacturers.

e. Classification of transients generated by low power transmitters built by the same manufacturer.

f. Classification of transients generated by low power transmitters of the same model type.

8.3 Recommendations

Based upon the work completed for this thesis, the following recommendations are made.

a. Obtain more transient recordings to expand the database of transients.

b. Expand the number of transients classified to further test the system.

c. Expand the study to include high power transients.

d. Develop a window based user interface.

e. Test newer versions of the Sound Blaster to determine compatibility with software developed.

f. Study different neural network topologies and training methods to expand the capabilities of the system.

REFERENCES

[Ande95] Darryl Anderson, "Transient signal classification using wavelet packet bases." B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1995, (vi+120) pp.

[Brod95] E. Brodsky, *Programming the Sound Blaster 16 DSP*. (Available through anonymous ftp at x2ftp.oulu.fi/pub/msdos/programming/mxlibs/xsb16io1.zip), 8 pp, 1995.

[Chui92] C. K. Chui, *Introduction to Wavelets*. San Diego, CA: Academic Press, 576 pp, 1992.

[Couc93] L. Couch II, *Digital and Analog Communication Systems, 4th ed.* New York, NY: Maxwell Macmillan, pp. 72-74, 1993.

[Daub88] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Comm. Pure App. Math.*, v. XLI, 1988, pp. 909-996.

[Daub94] I. Daubechies, *Ten Lectures on Wavelets*. Philadelphia, PA: SIAM, 357 pp, 1992.

[Davi91] L. Davis, *Handbook of Genetic Algorithms*, New York, NY: Van Nostrand Reinhold, 1991.

[Diet94] James Dietrich, "A wavelet analysis of transients in phase-locked loops." B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1994, (v+89) pp.

[Hayk94] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Englewood CA: Macmillan Publishing Co, 696 pp, 1994.

[Khan95] Imran Khan, "Transient analysis in frequency synthesizers." B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1995, (vii+93) pp.

[Kins87] W. Kinsner, "Microprocessor and microcomputer interfacing for real-time systems", Lecture Notes, Dept. Electrical and Computer Eng., University of Manitoba, 146 pp, 1987.

[Kins94] W. Kinsner, "Fractal Dimensions: Morphological, entropy, spectrum, and variance classes ", Technical Report, DEL94-5, University of Manitoba, 146 pp, May 1994.

[Kwok95] Raymond Kwok, "High-speed capture of turn-on transients in transmitters." B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1995, (x+173) pp.

[Mall89] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Inform. Theory*, vol. 38, no. 2, pp. 617-643, March 1992.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*: The Art of Scientific Computing, 2nd Ed. New York, NY: Cambridge University Press, 994 pp, 1992.

[Ruda94] Tena Rudachek, "Phase-locked loops and their transients," B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1994, (vii+57) pp.

[Shaw94] Don Shaw, "Identification of transients in PLL using neural networks," B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1994, (ix+179) pp.

[SiPG92] A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts*, 3rd Ed. New York, NY: Addison-Wesley Publishing Company, 695 pp, 1992.

[Toon95] Jason Toonstra, "Wavelet analysis and genetic modelling of transients in radio transmitters." B.Sc. Thesis. Dept. Electrical & Computer Engineering, University of Manitoba, Winnipeg, Manitoba, Canada, March 1995, (vi+147) pp.

[Youn93] R. K. Young, *Wavelet Theory and Its Applications*. Norwell, MA: Kluwer Academic Publishers, 1993.

APPENDIX A

SOURCE CODE

```
/*      Based on Ethan Brodsky 1995 code.  All rights reserved       */
/*      Transient capture developed by J. Toonstra and W. Kinsner, 1995 */

/* █ SBIO.C ████████████████████████████████████████

/* █ Interface ██████████████████████████████████████

#define TRUE  1
#define FALSE 0

  typedef enum {input, output} mode;

 /* Interface procedures and functions */
  int init_sb
        (
         int  baseio,
         char irq,
         char dma16,
         mode io,
         unsigned int rate
        );
  void shutdown_sb(void);

  void startio(unsigned long length);
  void sethandler(void far *proc);

// void getbuffer(int far **bufptr, unsigned int length);
// void freebuffer(int far **bufptr);

 /* Interface variables that can be changed in the background */
  volatile long intcount;
  volatile int  done;
  volatile char curblock;
  volatile long samplesremaining;

/* ██ Implementation ████████████████████████████████


#include <alloc.h>
#include <conio.h>
#include <dos.h>
#include <mem.h>
#include <stdlib.h>
```

```
#include <stdio.h>

#define lo(value) (unsigned char)((value) & 0x00FF)
#define hi(value) (unsigned char)((value) >> 8)

  int  resetport;
  int  readport;
  int  writeport;
  int  pollport;
  int  poll16port;

  int  pic_rotateport;
  int  pic_maskport;

  int  dma_maskport;
  int  dma_clrptrport;
  int  dma_modeport;
  int  dma_baseaddrport;
  int  dma_countport;
  int  dma_pageport;

  char irq_startmask;
  char irq_stopmask;
  char irq_intvector;
  char int_controller;

  char dma_startmask;
  char dma_stopmask;
  char dma_mode;

  void interrupt (*oldintvector)() = NULL;
  int  handlerinstalled;

//  void far *memarea = NULL; /* Twice the size of the output buffer */
//  int  memareasize;

  void far *MemoryArea = NULL;
  int MemoryAreaSize;

  unsigned char far *Buffer;

  unsigned char far *BufferPointer;
```

```
 unsigned long buf_addr;   /* 16-bit addressing */
 unsigned char buf_page;
 unsigned int  buf_ofs;

 int buf_length;          /* In words */
 int block_length;         /* In words */

 unsigned int samplingrate;

 unsigned long SampleIndex;
 unsigned long SampleLimit;

 unsigned char TransientBuffer[32768];

 int Transient;

 unsigned int NoiseLocation;

 mode iomode;
 void far (*handler)(void) = NULL;

/* == Low level sound card I/O ============================================ */
 void write_dsp(unsigned char value)
        {
                while (inp(writeport) & 0x80);  /* Wait for bit 7 to be cleared */
                outp(writeport, value);
        }

 unsigned char read_dsp(void)
        {
                unsigned int value;

                while (!(inp(pollport) & 0x80)); /* Wait for bit 7 to be set */
                value = inp(readport);
                return value;
        }

        int reset_dsp(void)
         {
                int i;

                outp(resetport, 1);
                outp(resetport, 0);
                i = 100;

                while ((read_dsp() != 0xAA) && i--);
                return i;
            }

/* == Initialization and shutdown ============================================ */
  void installhandler(void);  /* Prototypes for private functions */
  void uninstallhandler(void);
  void sb_exitproc(void);

  int  init_sb(int baseio, char irq, char dma16, mode io, unsigned int rate)
        {
        /* Sound card IO ports */
              resetport  = baseio + 0x006;
              readport   = baseio + 0x00A;
              writeport  = baseio + 0x00C;
              pollport   = baseio + 0x00E;
              poll16port = baseio + 0x00F;

        /* Reset DSP */
              if (!reset_dsp()) return FALSE;

        /* Compute interrupt ports and parameters */
              if (irq < 8)
                {
                      int_controller = 1;
                      pic_rotateport = 0x20;
                      pic_maskport   = 0x21;
                      irq_intvector  = 0x08 + irq;
                }
              else
                {
                      int_controller = 2;
                      pic_rotateport = 0xA0;
                      pic_maskport   = 0x21;
                      irq_intvector  = 0x70 + irq-8;
                }
              irq_stopmask  = 1 << (irq % 8);
              irq_startmask = ~irq_stopmask;

        /* Compute DMA ports and parameters */
              dma_maskport    = 0xD4;
            dma_clrptrport   = 0xD8;
            dma_modeport     = 0xD6;
            dma_baseaddrport = 0xC0 + 4*(dma16-4);
            dma_countport    = 0xC2 + 4*(dma16-4);

            switch(dma16)
             {
                    case 5:  dma_pageport = 0x8B; break;
                    case 6:  dma_pageport = 0x89; break;
                    case 7:  dma_pageport = 0x8A; break;
             }

            dma_stopmask  = dma16-4 + 0x04;   /* 000001xx */
            dma_startmask = dma16-4 + 0x00;   /* 000000xx */

        /* Other initialization */
            samplingrate = rate;
            iomode = io;
            switch (iomode)
             {
                    case input:  dma_mode = dma16-4 + 0x54; break;  /* 010101xx */
                    case output: dma_mode = dma16-4 + 0x58; break;  /* 010110xx */
             }

            installhandler();    /* Install interrupt handler */

            return TRUE;
        }

/* ————————————————————————————————————————————————————————————————

 void shutdown_sb(void)
        {
            if (handlerinstalled) uninstallhandler();
            reset_dsp();
        }

/* ================================================================

 void startio(unsigned long length)
        {
            done = FALSE;
            samplesremaining = length;
        curblock = 0;

        Transient = 0;

        SampleLimit = block_length;

        SampleIndex = 0;

        samplesremaining -= (block_length/2);

/* Program DMA controller */

        printf("\n");
        printf("Programming the dma controller. \n");

        outp(dma_maskport,    dma_stopmask);
        printf("1.  Reseting the dma mask for programming. \n");

        outp(dma_clrptrport,  0x00);
        printf("2.  Clearing the dma flip-flop. \n");

        outp(dma_modeport,    dma_mode);
        printf("3.  Setting the dma mode. \n");

        outp(dma_baseaddrport, lo(buf_ofs));                /* Low byte of offset
*/
        outp(dma_baseaddrport, hi(buf_ofs));        /* High word of offset */
        printf("4.  Setting the buffer offset. \n");

        outp(dma_countport,   lo(buf_length-1));    /* Low byte of count   */
        outp(dma_countport,   hi(buf_length-1));    /* High byte of count  */
        printf("5.  Setting the buffer length in the counter. \n");

        outp(dma_pageport,    buf_page);
        printf("6.  Setting the buffer page in the dma controller. \n");

        outp(dma_maskport,    dma_startmask);
        printf("7.  Reseting the dma mask port to complete programming. \n");

/* Program sound card */

        printf("\n");
```

```
                printf("Programming the soundblaster. \n");

                iomode = input;

                switch (iomode)
                 {
                        case input:  write_dsp(0x42); break;  /* Set input sampling rate  */
                        case output: write_dsp(0x41); break;  /* Set output sampling rate
*/
                 }
                write_dsp(hi(samplingrate));          /* High byte of sampling rate */
                write_dsp(lo(samplingrate));          /* Low byte of sampling rate  */
                printf("1.  Setting the input sampling rate. \n");

                switch (iomode)
                 {
                        case output: write_dsp(0xB6); break;  /* 16-bit D->A, A/I, FIFO
*/
                        case input:  write_dsp(0xBE); break;  /* 16-bit A->D, A/I, FIFO
*/
                 }
                write_dsp(0x20);                  /* DMA Mode:  16-bit unsigned stereo */
                printf("2.  Setting the 16 bit transfer. \n");

                printf("3.  Writting the block length the soundblaster. \n");
                printf("\n");
                printf("Transfer has begun ... \n \n");
                write_dsp(lo(block_length-1));      /* Low byte of block length     */
                write_dsp(hi(block_length-1));      /* High byte of block length    */
        }

/* == Interrupt handling ============================================== */

 void sethandler(void far *proc)
        {
                handler = proc;
        }

/* -----------------------------------------------------------------

 void interrupt inthandler()
        {                       /* CurBlock -> Block that just finished */
                intcount++;
```

```
if(curblock == 0)
{
        BufferPointer = Buffer;
}
else
{
        BufferPointer = Buffer + (2*block_length);
}

if(((*(BufferPointer+(2*255)) >= 128) &&
        ((*(BufferPointer+(2*255)+1)) != 128))
        && (!Transient)) /* If the right channel sample (squelch) is not
equal
                                                    to unsigned zero,
                                                    finish transfer to
                                                    array. */
{
        Transient = !Transient;
}

while(SampleIndex < SampleLimit)
{
        TransientBuffer[((0x7FFF)&SampleIndex)] = *BufferPointer;  /*
        Copy low byte of discriminator sample. */

        SampleIndex++;
        BufferPointer++;

        TransientBuffer[((0x7FFF)&SampleIndex)] = *BufferPointer; /*
        Copy high byte of discriminator sample. */

        SampleIndex++;
        BufferPointer++;

        BufferPointer++;  /* Increment past low byte of sqelch sample. */
        BufferPointer++;  /* Increment past high byte of squelch sample. */
}

NoiseLocation=((0x7FFF)&SampleIndex);

SampleLimit += block_length;
```

```
            if(Transient)
            {
                    samplesremaining -= (block_length/2);
            }

            curblock = !curblock;       /* Toggle current block */
            if (samplesremaining < 0)
             {
                    done = TRUE;
                    write_dsp(0xD9);
             }

            inp(poll16port);
            outp(0x20, 0x20);
            outp(0xA0, 0x20);
        }                       /* CurBlock -> Block that just started */


/* ------------------------------------------------------------------------

  void installhandler(void)
        {
            disable();                          /* Disable interrupts  */
            outp(pic_maskport, (inp(pic_maskport)|irq_stopmask)); /* Mask IRQ    */

            oldintvector = getvect(irq_intvector);      /* Save old vector    */
            setvect(irq_intvector, inthandler);        /* Install new handler */

            outp(pic_maskport, (inp(pic_maskport)&irq_startmask)); /* Unmask IRQ
*/
            enable();                           /* Reenable interupts  */

            handlerinstalled = TRUE;

        }
/* ------------------------------------------------------------------------

  void uninstallhandler(void)
        {
            disable();                          /* Disable interrupts  */
            outp(pic_maskport, (inp(pic_maskport)|irq_stopmask)); /* Mask IRQ    */
            setvect(irq_intvector, oldintvector);       /* Restore old vector  */

            enable();                              /* Enable interrupts   */

            handlerinstalled = FALSE;
        }

/* == Memory management ==================================================

  unsigned long getlinearaddr(unsigned char far *p)
        {
            unsigned long addr;

            addr = (unsigned long)FP_SEG(p)*16 + (unsigned long)FP_OFF(p);
            return(addr);
        }

/* ----------------------------------------------------------------------

  void getbuffer(unsigned int length)
        {
        /* Find a block of memory that does not cross a page boundary */
            MemoryAreaSize = 16 * length;
            if ((MemoryArea = malloc(MemoryAreaSize)) == NULL) /* Can't allocate
            mem?  */
              exit(EXIT_FAILURE);                  /*  error            */
            Buffer = (unsigned char far *)MemoryArea;            /* Pick first half
*/

            if (((getlinearaddr(Buffer) >> 1) % 65536) + length*8 > 65536)
              Buffer += 8*length; /* Pick second half to avoid crossing boundary */

        /* DMA parameters */
            buf_addr = getlinearaddr(Buffer);
            buf_page = buf_addr >> 16;
            buf_ofs  = (buf_addr >> 1) % 65536;
            buf_length = length*2;  block_length = length;          /* In samples */
        }

/* ----------------------------------------------------------------------

  void freebuffer(void)
        {
            Buffer = NULL;
            free((void *)MemoryArea);
        }

/* == Exit shutdown ================================================
  void sb_exitproc(void)
        {
            outp(0x20, 0x20);  outp(0xA0, 0x20);  /* Acknowledge any hanging ints */
            write_dsp(0xD0);                /* Pause digitized sound output */
            outp(dma_maskport, dma_stopmask);   /* Mask DMA channel          */
            if (handlerinstalled) uninstallhandler(); /* Uninstall int handler   */
            reset_dsp();                   /* Reset SB DSP             */
        }


#define SizeOfBlock 256

        int main(int argc,char *argv[])
        {
            FILE *FilePointer;

            unsigned int FileIndex=0;

            unsigned int X;

            unsigned char FileBuffer[17408];

            if(argc != 2)
            {
                printf("No file name \n");
                exit(1);
            }

            printf("Opening file %s ...",argv[1]);
            FilePointer = fopen(argv[1],"w");
            if(FilePointer != NULL)
            {
                printf(" opened. \n \n");
            }
            else
            {
                printf(" file cannot be opened. \n");
                exit(1);
            }
        printf("Getting buffer ...");
        getbuffer(SizeOfBlock);
        printf(" done. \n \n");

        printf("Initializing the soundblaster ...");
        init_sb(0x220,7,5,input,44100); /* Modify this for changing irq */
/*        Address,irq,dma           */
        printf(" done. \n \n");

        printf("Starting procedure for the soundblaster transfer: \n");
        startio(8192);

        while(!(done||kbhit()));

        if(done)
        {
                printf("Writting to file ...");

                printf("Noise location:  %u. \n",NoiseLocation);
                getc(stdin);

                FileIndex=0;
                while(FileIndex <(2*(16384U)))
                {
                        X =
                        TransientBuffer[((0x7FFF)&(NoiseLocation+FileIndex))];
                        FileIndex++;
                        X = X +
                        (TransientBuffer[((0x7FFF)&(NoiseLocation+FileIndex))]*
                        255);
                        FileIndex++;
                        fprintf(FilePointer,"%u \n",X);
                }

                FileIndex = 0;

                while(FileIndex < (2*8704))
                {
                        FileBuffer[FileIndex] =
                        TransientBuffer[((SampleIndex+15360)&0x7FFF)];

                        SampleIndex++;
                        FileIndex++;
                }

                printf(" done. \n \n");

                printf("Closing file ...");
                fclose(FilePointer);
                printf(" done. \n \n");
        }

        if(kbhit())
        {
                printf("Termination of search. \n");
                getch();
        }

        printf("Shutting down the soundblaster ...");
        shutdown_sb();
        printf(" done. \n");

//      sethandler(NULL);

        printf("Free buffer memory ...");
        freebuffer();
        printf(" done. \n");

        return(EXIT_SUCCESS);
}
```

```
/* variance dimension calculator */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <values.h>

#define FileSize 16384U

#define NumberOfSuperSamples 4096
#define SuperSampleSize 4

#define maxValue 65536L

#define maximumDelay 10

/* values for variance dimension calculations */
#define b 2.0
#define c 30

int dyadic=1;


float VarianceDimensionXX(float *data,unsigned windowSize)
{
        int kHi;
        int kLow;

        int kMax;
        int kMin;

        int kIndex;
        int windowIndex;

        int delta;

        float *variance;

        float sumSquare;
        float squareSum;

        float sumX;
        float sumY;
        float sumXY;
        float squareSumX;
        float sumSquareX;

        float slope;

        kHi=((int) (log(((float) windowSize))/log(2.0)))-5;
        kLow=1;

        kMax=pow(2,kHi);
        kMin=kLow;

/*      printf("k High:  %d. \n",kHi);
        printf("k Low:  %d. \n",kLow);
        printf("k Maximum:  %d. \n",kMax);
        printf("k Minimum:  %d. \n",kMin); */

        variance=((float *) malloc(sizeof(float)*(kMax+1)));

        if(!variance)
        {
                printf("Cannot allocate memory for variances. \n");

                printf("Exitting to dos... \n");
                exit(1);
        }

        for(kIndex=kMax;kIndex>=kMin;kIndex--)
        {
                sumSquare=0.0;
                squareSum=0.0;

                windowIndex=0;

                delta=kIndex;

/*      printf("k index:  %d. \n",kIndex); */

                while((windowIndex+delta)<windowSize)
                {
                        sumSquare+=(((data[windowIndex+delta])-
                                                (data[windowIndex]))*
                                                ((data[windowIndex+delta])-
                                                (data[windowIndex])));

                                squareSum+=((data[windowIndex+delta])-
                                                (data[windowIndex]));

                                windowIndex++;
                        }

                        squareSum=((1.0/((float) windowIndex))*(squareSum*squareSum));

                        variance[kIndex]=((1.0/((float)
                        (windowIndex-1)))*(sumSquare-squareSum));
                }

                sumX=0.0;
                sumY=0.0;
                sumXY=0.0;
                sumSquareX=0.0;
                for(kIndex=kMin;kIndex<=kMax;kIndex++)
                {
                        sumXY+=log(variance[kIndex])*log(((float) kIndex));
                        sumX+=log(((float) kIndex));
                        sumY+=log(variance[kIndex]);
                        sumSquareX+=log(((float) kIndex))*log(((float) kIndex));
                }
                squareSumX=sumX*sumX;

                slope=(((((float) kMax)*sumXY)-(sumX*sumY))/((((float)
                kMax)*sumSquareX)-squareSumX));

                free(variance);

/*              printf("Dimension:  %f. \n",2-(0.5*slope)); */

                return(2-(0.5*slope));
}

void main(int argc,char *argv[])
{
                unsigned uData;

                unsigned index;
                unsigned superSampleIndex;
unsigned windowIndex;
unsigned filterIndex;

unsigned minimumDirivativeLocation;
unsigned maximumDirivativeLocation;

unsigned minimumDimensionLocation;

unsigned filterSize;
unsigned filterSplit;

unsigned transientLocation;

unsigned noiseLocation;

unsigned searchStart;

int windowSize;
int windowIncrement;

float *data;

float *window;

float *dimension;
float *filteredDimension;
float *tempDimension;

float *switchPointer;

float minimumDirivative;
float maximumDirivative;

float minimumDimension;

float superSample;

float *dirivative;

FILE *filePointer;

if(argc!=9)
{
		printf("Incorrect number of arguments. \n");
		printf("Exitting to dos... \n");

		exit(1);
	}

	filePointer=fopen(argv[1],"r");
	if(!filePointer)
	{
		printf("File does not exist. \n");
		printf("Exitting to dos... \n");

		exit(1);
	}

	windowSize=atoi(argv[4]);

	windowIncrement=atoi(argv[5]);

	data=((float *) malloc(sizeof(float)*NumberOfSuperSamples));
	if(!data)
	{
		printf("Data memory cannot be allocated. \n");
		printf("Exitting to dos... \n");

		exit(1);
	}

	window=((float *) malloc(sizeof(float)*windowSize));
	if(!window)
	{
		printf("Window memory cannot be allocated. \n");
		printf("Exitting to dos... \n");

		exit(1);
	}

	dimension=((float *) malloc(sizeof(float)*((unsigned) (((float)
	NumberOfSuperSamples)/((float) windowIncrement)))));
	if(!dimension)
	{
		printf("Dimension memory cannot be allocated. \n");
		printf("Exitting to dos... \n");
```

```
        exit(1);
}

filteredDimension=((float *) malloc(sizeof(float)*((unsigned) (((float)
NumberOfSuperSamples)/((float) windowIncrement)))));
if(!filteredDimension)
{
        printf("Dimension filter memory cannot be allocated. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

superSampleIndex=1;
superSample=0.0;
for(index=0;index<=FileSize;index++)
{
        fscanf(filePointer,"%u\n",&uData);
        if(index<(SuperSampleSize*superSampleIndex))
        {
                superSample+=((float) uData);
        }
        else
        {
/*              printf("supersample: %f. \n",superSample); */
                data[superSampleIndex-1]=((superSample/((float)
                SuperSampleSize)));

                superSampleIndex++;
                superSample=((float) uData);
        }
}

fclose(filePointer);

for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
windowIncrement)));index++)
{
        for(windowIndex=0;windowIndex<windowSize;windowIndex++)
        {
                if((windowIndex+(windowIncrement*(index)))>=NumberOfSuperS
                amples)
                {
                        window[windowIndex]=data[(windowIndex+(windowIncre
                        ment*(index)))-NumberOfSuperSamples];
                }
                else
                {
                        window[windowIndex]=data[windowIndex+(windowIncre
                        ment*(index))];
                }
        }

/*              printf("%u -- ",windowIncrement*index); */
                dimension[index]=VarianceDimensionXX(window,windowSize);
        }

        free(window);

        transientLocation=0;

        filterSize=101;

        filterSplit=((unsigned) floor(filterSize/2.0));
        printf("Filter size:  %u. \n",filterSize);
        printf("Filter split:  %u. \n",filterSplit);

        for(index=filterSplit;index<(((unsigned) (((float) NumberOfSuperSamples)/((float)
        windowIncrement)))-filterSplit);index++)
        {
                filteredDimension[index]=dimension[index];
                for(filterIndex=1;filterIndex<=filterSplit;filterIndex++)
                {
                        filteredDimension[index]+=
                                (dimension[index+filterIndex]+dimension[index-filte
                                rIndex]);
                }
                filteredDimension[index]=filteredDimension[index]/filterSize;
        }

        for(index=0;index<filterSplit;index++)
        {
                filteredDimension[index]=dimension[index];
                for(filterIndex=1;filterIndex<=filterSplit;filterIndex++)
                {
                        filteredDimension[index]+=dimension[index+filterIndex];
        if(index<filterIndex)
        {
                filteredDimension[index]+=
                        dimension[((unsigned) (((float)
                        NumberOfSuperSamples)/((float)
                        windowIncrement)))+(index-filterIndex)];
        }
        else
        {
                filteredDimension[index]+=dimension[index-filterIndex];
        }
}
filteredDimension[index]=filteredDimension[index]/filterSize;

filteredDimension[((unsigned) (((float) NumberOfSuperSamples)/((float)
windowIncrement)))-1-index]=
        dimension[((unsigned) (((float) NumberOfSuperSamples)/((float)
        windowIncrement)))-1-index];
for(filterIndex=1;filterIndex<=filterSplit;filterIndex++)
{
        filteredDimension[((unsigned) (((float)
        NumberOfSuperSamples)/((float) windowIncrement)))-1-index]+=
                dimension[((unsigned) (((float)
                NumberOfSuperSamples)/((float)
                windowIncrement)))-1-(index+filterIndex)];
        if(index<filterIndex)
        {
                filteredDimension[((unsigned) (((float)
                NumberOfSuperSamples)/
                ((float) windowIncrement)))-1-index]+=
                dimension[(filterIndex-index)-1];
        }
        else
        {
                filteredDimension[((unsigned) (((float)
                NumberOfSuperSamples)/
                ((float) windowIncrement)))-1-index]+=
                dimension[(((unsigned) (((float) NumberOfSuperSamples)/
                ((float) windowIncrement)))-(index+1))+filterIndex];
        }
}
filteredDimension[((unsigned) (((float) NumberOfSuperSamples)/((float)
windowIncrement)))-1-index]=
			filteredDimension[((unsigned) (((float) NumberOfSuperSamples)/((float)
			windowIncrement)))-1-index]/filterSize;
		}

/*		free(dimension); */

		tempDimension=((float *) malloc(sizeof(float)*((unsigned) (((float)
		NumberOfSuperSamples)/((float) windowIncrement)))));
		if(!tempDimension)
		{
			printf("Dirivative memory cannot be allocated. \n");
			printf("Exitting to dos... \n");

			exit(1);
		}

		noiseLocation=0;
		for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
		windowIncrement)));index++)
		{
			if((!noiseLocation)&&(filteredDimension[index]>1.950))
			{
				noiseLocation=index;
			}
		}

		for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
		windowIncrement)));index++)
		{
			if((index+noiseLocation)<((unsigned) (((float)
			NumberOfSuperSamples)/((float) windowIncrement))))
			{
				tempDimension[index]=filteredDimension[noiseLocation+index];
			}
			else
			{
				tempDimension[index]=
				filteredDimension[(noiseLocation+index)-((unsigned) (((float)
				NumberOfSuperSamples)/((float) windowIncrement)))];
			}
		}

		switchPointer=filteredDimension;
```

```
 filteredDimension=tempDimension;
 tempDimension=switchPointer;


 for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
 windowIncrement)));index++)
 {
         if((index+noiseLocation)<((unsigned) (((float)
         NumberOfSuperSamples)/((float) windowIncrement))))
         {
                 tempDimension[index]=dimension[noiseLocation+index];
         }
         else
         {
                 tempDimension[index]=
                 dimension[(noiseLocation+index)-((unsigned) (((float)
                 NumberOfSuperSamples)/((float) windowIncrement)))];
         }
 }


dimension=tempDimension;

filePointer=fopen(argv[2],"w");
if(!filePointer)
{
        printf("Cannot write file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
windowIncrement)));index++)
{
        fprintf(filePointer,"%u %f\n",index,filteredDimension[index]);
}

fclose(filePointer);

filePointer=fopen(argv[7],"w");
if(!filePointer)
{
        printf("Cannot write file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

searchStart=0;
for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
windowIncrement)));index++)
{
        if((!searchStart)&&(filteredDimension[index]<1.800))
        {
                searchStart=index;
        }
}

minimumDimension=MAXFLOAT;
for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/((float)
 windowIncrement)));index++)
{
        if(minimumDimension>dimension[index])
        {
                minimumDimension=dimension[index];
                minimumDimensionLocation=index;
        }
}

dirivative=((float *) malloc(sizeof(float)*((unsigned) (((float)
NumberOfSuperSamples)/((float) windowIncrement)))));
if(!dirivative)
{
        printf("Dirivative memory cannot be allocated. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

for(index=searchStart;(index<searchStart+532);index++)
{
        dirivative[index]=pow(50.0,(2.0-filteredDimension[index]))*
        (filteredDimension[index+20]-filteredDimension[index]);
        fprintf(filePointer,"%u
        %f\n",index,(pow(50.0,(2.0-filteredDimension[index]))*
        (filteredDimension[index+20]-
        filteredDimension[index])));
}
fclose(filePointer);

minimumDirivative=0.0;
for(index=searchStart;index<(searchStart+532);index++)
{
        if(dirivative[index]<minimumDirivative)
        {
                minimumDirivative=dirivative[index];
                minimumDirivativeLocation=index;
        }
}

maximumDirivative=0.0;
for(index=searchStart;index<(searchStart+532);index++)
{
        if(dirivative[index]>maximumDirivative)
        {
                maximumDirivative=dirivative[index];
                maximumDirivativeLocation=index;
        }
}

printf("Minimum dirivative:  %u. \n",minimumDirivativeLocation);
printf("Maximum dirivative:  %u. \n",maximumDirivativeLocation);

if(maximumDirivativeLocation<minimumDirivativeLocation)
{
        printf("Doing a second search... \n");

        minimumDirivative=0.0;
        for(index=searchStart;index<maximumDirivativeLocation;index++)
        {
                if(dirivative[index]<minimumDirivative)
                {
                        minimumDirivative=dirivative[index];
                        minimumDirivativeLocation=index;
                }
        }
}
```

```
filePointer=fopen(argv[8],"w");
if(!filePointer)
{
        printf("Cannot write file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

for(index=searchStart;index<(searchStart+512);index++)
{
        dirivative[index]=(dirivative[index+20]-dirivative[index]);
        fprintf(filePointer,"%u %f\n",index,dirivative[index]);
}
fclose(filePointer);

filePointer=fopen(argv[6],"w");
if(!filePointer)
{
        printf("Cannot write file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

printf("Minimum dimension:  %f >> %u.
\n",minimumDimension,minimumDimensionLocation);
printf("Minimum dirivative:  %f >> %u.
\n",minimumDirivative,minimumDirivativeLocation);
printf("Noise location:  %u. \n",noiseLocation);

if(minimumDimensionLocation>(maximumDirivativeLocation+64))
{
        transientLocation=minimumDirivativeLocation;

        printf("Took the minimum dirivative as transient indicator. \n");
}
else
{
        transientLocation=minimumDimensionLocation;

        printf("Took the minimum dimension as transient indicator. \n");
}
```

```
if((noiseLocation+transientLocation)<((unsigned) (((float)
NumberOfSuperSamples)/((float) windowIncrement))))
{
        transientLocation=transientLocation+noiseLocation;
}
else
{
        transientLocation=(transientLocation+noiseLocation)-((unsigned) (((float)
        NumberOfSuperSamples)/((float) windowIncrement)));
}

printf("Transient location:  %u. \n",transientLocation);

transientLocation=transientLocation*windowIncrement;

for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/2.0));index++)
{
        if(index+transientLocation>=NumberOfSuperSamples)
        {
                fprintf(filePointer,"%u\n",((unsigned)
                data[index+transientLocation-NumberOfSuperSamples]));
        }
        else
        {
                fprintf(filePointer,"%u\n",((unsigned)
                data[index+transientLocation]));
        }
}

fclose(filePointer);

filePointer=fopen(argv[3],"w");
if(!filePointer)
{
        printf("Cannot write file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

transientLocation=transientLocation*windowIncrement;

superSampleIndex=0;
```

```
        superSample=0.0;

        for(index=0;index<((unsigned) (((float) NumberOfSuperSamples)/2.0));index++)
        {
                if(index+transientLocation>=NumberOfSuperSamples)
                {
                        if(superSampleIndex<32)
                        {
                                superSample+=data[index+transientLocation-
                                NumberOfSuperSamples];
                                superSampleIndex++;
/*                              printf("Super sample:  %u  %f.
                                        \n",superSampleIndex,superSample); */
                        }
                        else
                        {
                                fprintf(filePointer,"%u\n",((unsigned) (superSample/32.0)));

                                superSampleIndex=1;
                                superSample=data[index+
                                transientLocation-NumberOfSuperSamples];
                        }
                }
                else
                {
                        if(superSampleIndex<32)
                        {
                                superSample+=data[index+transientLocation];
                                superSampleIndex++;

        /*                      printf("Super sample:  %u  %f.
                                 \n",superSampleIndex,superSample); */

                        }
                        else
                        {
                                fprintf(filePointer,"%u\n",((unsigned) (superSample/32.0)));

                                superSampleIndex=1;
                                superSample=data[index+transientLocation];
                        }
                }
        }
```

```
	fclose(filePointer);

	free(dirivative);
}
```

```
/* Genetic algorithm source code */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define C0 0.4829629131445341
#define C1 0.8365163037378079
#define C2 0.2241438680420134
#define C3 -0.1294095225512604

#define ForwardTransform 1
#define InverseTransform -1

#define NR_END 1
#define FREE_ARG char*

#define NumberOfGenes 64
#define NumberOfChromosomes 20

#define NumberOfGenerations 100

#define InitialRandomization 24

#define LocationOfLowerCoefficients 8
#define LocationOfUpperCoefficients 32

#define NumberOfLowerCoefficients 24
#define NumberOfUpperCoefficients 32

#define MaximumCrossOvers 20

#define MaximumMutations 8

void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
	fprintf(stderr,"Numerical Recipes run-time error...\n");
	fprintf(stderr,"%s\n",error_text);
	fprintf(stderr,"...now exiting to system...\n");
	exit(1);
}
```

```
float *vector(long nl, long nh)
/* allocate a float vector with subscript range v[nl..nh] */
{
        float *v;

        v=(float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
        if (!v) nrerror("allocation failure in vector()");
        return v-nl+NR_END;
}

void free_vector(float *v, long nl, long nh)
/* free a float vector allocated with vector() */
{
        free((FREE_ARG) (v+nl-NR_END));
        nh = nh + 0;  // Just included to quiet warning reports
}

float **matrix(long nrl, long nrh, long ncl, long nch)
/* allocate a float matrix with subscript rage m[mrl..nrh][ncl..nch] */
{
        long i, nrow=nrh-nrl+1, ncol=nch-ncl+1;
        float **m;

        /* allocate pointers to rows */
        m=(float **) malloc((size_t)((nrow+NR_END)*sizeof(float*)));
        if(!m) nrerror("allocation failure 1 in matrix()");
        m += NR_END;
        m -= nrl;

        /* allocate rows and set pointers to them */
        m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
        if(!m[nrl]) nrerror("allocation failure 2 in matrix()");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

        /* return pointer to array of pointers to rows */
        return m;
}

void free_matrix(float **m, long nrl, long nrh, long ncl, long nch)
/* free a float matrix allocated by matrix() */
```

```
{
        free((FREE_ARG) (m[nrl]+ncl-NR_END));
        free((FREE_ARG) (m+nrl-NR_END));

        nrh += 0;
        nch += 0;
}

void daub4(float a[], unsigned long n, int isign)
{
        float *wksp;
        unsigned long nh,nh1,i,j;

        if (n < 4) return;
        wksp=vector(1,n);
        nh1=(nh=n >> 1)+1;
        if (isign >= 0) {
                for (i=1,j=1;j<=n-3;j+=2,i++) {
                        wksp[i]=C0*a[j]+C1*a[j+1]+C2*a[j+2]+C3*a[j+3];
                        wksp[i+nh] = C3*a[j]-C2*a[j+1]+C1*a[j+2]-C0*a[j+3];
                }
                wksp[i]=C0*a[n-1]+C1*a[n]+C2*a[1]+C3*a[2];
                wksp[i+nh] = C3*a[n-1]-C2*a[n]+C1*a[1]-C0*a[2];
        } else {
                wksp[1]=C2*a[nh]+C1*a[n]+C0*a[1]+C3*a[nh1];
                wksp[2] = C3*a[nh]-C0*a[n]+C1*a[1]-C2*a[nh1];
                for (i=1,j=3;i<nh;i++) {
                        wksp[j++]=C2*a[i]+C1*a[i+nh]+C0*a[i+1]+C3*a[i+nh1];
                        wksp[j++] = C3*a[i]-C0*a[i+nh]+C1*a[i+1]-C2*a[i+nh1];
                }
        }
        for (i=1;i<=n;i++) a[i]=wksp[i];
        free_vector(wksp,1,n);
}
/* (C) Copr. 1986-92 Numerical Recipes Software v)2$0Y`4. */

void wt1(float a[], unsigned long n, int isign,
        void (*wtstep)(float [], unsigned long, int))
{
        unsigned long nn;

        if (n < 4) return;
        if (isign >= 0) {
            for (nn=n;nn>=4;nn>>=1) (*wtstep)(a,nn,isign);
        } else {
            for (nn=4;nn<=n;nn<<=1) (*wtstep)(a,nn,isign);
        }
}
/* (C) Copr. 1986-92 Numerical Recipes Software v)2$0Y`4. */

float *GeneticAlgorithm(float **geneticPool, unsigned chromosomes, unsigned genes)
{
        unsigned cIndex;
        unsigned gIndex;
        unsigned scaleIndex;
        unsigned generationIndex;
        unsigned crossOverIndex;
        unsigned selectionIndex;
        unsigned mutationIndex;
        unsigned bestSolutionIndex;

        unsigned parentA;
        unsigned parentB;

        unsigned numberOfMutations;

        unsigned mutationGeneA;
        unsigned mutationGeneB;

        unsigned totalCoefficients;

        float **temporaryGeneticPool;

        float *child;

        float *selectionVector;

        float *bestSolution;

        float scaleNormalizationTotal;
        float scaleCoefficientTotal;

        float selectionNormalization;

        float selectionCriterion;
float totalSelectionVector;

float chromosomeFitness;

float selection;

float bestSolutionFitness;

/* get the selection of the fittest criterion */
selectionCriterion=0.0;
for(scaleIndex=3;scaleIndex<=5;scaleIndex++)
{
        scaleCoefficientTotal=0.0;
        for(gIndex=(pow(2,scaleIndex)+1);gIndex<=pow(2,(scaleIndex+1));gInde
        x++)
        {
                scaleCoefficientTotal+=fabs(geneticPool[chromosomes+1][gIndex]
                );
        }

        if(scaleIndex==3)
        {
                selectionCriterion+=scaleCoefficientTotal;
        }
        else
        {
                if(scaleIndex==4)
                {
                        selectionCriterion+=0.90*scaleCoefficientTotal;
                }
                else
                {
                        selectionCriterion+=0.50*scaleCoefficientTotal;
                }
        }
}


/* start the genetic algorithm */

/* get memory for child of each generation */
child=vector(1,genes);
```

```
/* get the memory for the selection vector */
selectionVector=vector(1,chromosomes);

/* get the memory for the temporary genetic pool */
temporaryGeneticPool=matrix(1,chromosomes,1,genes);

/* get the memory for the best solution */
bestSolution=vector(1,genes);

for(generationIndex=1;generationIndex<=NumberOfGenerations;generationIndex
++)
{
        /* crossover */
        for(crossOverIndex=1;crossOverIndex<=MaximumCrossOvers;crossOverI
        ndex++)
        {
                /* get parents of child for next generation */
                parentA=(rand()%chromosomes)+1;
                parentB=(rand()%chromosomes)+1;

                /* generate child */
                for(gIndex=1;gIndex<=LocationOfLowerCoefficients;gIndex++)
                {
                        child[gIndex]=geneticPool[parentA][gIndex];
                }
                for(gIndex=(LocationOfLowerCoefficients+1);gIndex<=genes;gInd
                ex++)
                {
                        child[gIndex]=geneticPool[parentB][gIndex];
                }

                /* enter child in genetic pool */
                if(rand()%2)
                {
                        for(gIndex=1;gIndex<=genes;gIndex++)
                        {
                                geneticPool[parentA][gIndex]=child[gIndex];
                        }
                }
                else
                {
                        for(gIndex=1;gIndex<=genes;gIndex++)
                        {
```

```
                        geneticPool[parentB][gIndex]=child[gIndex];
                    }
            }
    }

    /* selection of the fittest */
    for(cIndex=1;cIndex<=chromosomes;cIndex++)
    {
            chromosomeFitness=0.0;
            for(scaleIndex=3;scaleIndex<=5;scaleIndex++)
            {
                    scaleCoefficientTotal=0.0;
                    for(gIndex=(pow(2,scaleIndex)+1);gIndex<=pow(2,(scaleIn
                    dex+1));gIndex++)
                    {
                            if(geneticPool[cIndex][gIndex])
                            {
                                    scaleCoefficientTotal+=fabs(geneticPool[chr
                                    omosomes+1][gIndex]);
                            }
                    }

                    if(scaleIndex==3)
                    {
                            chromosomeFitness+=scaleCoefficientTotal;
                    }
                    else
                    {
                            if(scaleIndex==4)
                            {
                                    chromosomeFitness+=0.90*scaleCoefficient
                                    Total;
                            }
                            else
                            {
                                    chromosomeFitness+=0.50*scaleCoefficient
                                    Total;
                            }
                    }
            }
            selectionVector[cIndex]=chromosomeFitness/selectionCriterion;
    }
```

```
        bestSolutionFitness=0.0;
        for(cIndex=1;cIndex<=chromosomes;cIndex++)
        {
                if(selectionVector[cIndex]>bestSolutionFitness)
                {
                        bestSolutionFitness=selectionVector[cIndex];
                        bestSolutionIndex=cIndex;
                }
        }

        for(gIndex=1;gIndex<=genes;gIndex++)
        {
                bestSolution[gIndex]=geneticPool[bestSolutionIndex][gIndex];
        }

        selectionNormalization=0.0;
        for(cIndex=1;cIndex<=chromosomes;cIndex++)
        {
                selectionNormalization+=selectionVector[cIndex];
        }

        totalSelectionVector=0.0;
        for(cIndex=1;cIndex<=chromosomes;cIndex++)
        {
/*              printf("Sv: %f ",selectionVector[cIndex]); */
                totalSelectionVector+=selectionVector[cIndex]/selectionNormaliza
                tion;
                selectionVector[cIndex]=totalSelectionVector;
/*              printf("%f. \n",selectionVector[cIndex]); */
        }

        /* move the genetic pool to a temporary location for selection */
        for(cIndex=1;cIndex<=chromosomes;cIndex++)
        {
                for(gIndex=1;gIndex<=genes;gIndex++)
                {
                        temporaryGeneticPool[cIndex][gIndex]=geneticPool[cInde
                        x][gIndex];

        /*              printf("%u",((unsigned)
                        temporaryGeneticPool[cIndex][gIndex])); */
                }
/*              printf("\n"); */
```

```
        }

        /* perform selection */
        for(cIndex=1;cIndex<=chromosomes;cIndex++)
        {
                selection=((float) rand())/((float) RAND_MAX);
                selectionIndex=1;
                while(selection>selectionVector[selectionIndex])
                {
                        selectionIndex++;
                }
                if((selectionIndex+1)>chromosomes)
                selectionIndex=chromosomes-1;

/*              printf("Selected chromosome:  %u. \n",selectionIndex+1); */

                for(gIndex=1;gIndex<=genes;gIndex++)
                {
                        geneticPool[cIndex][gIndex]=temporaryGeneticPool[selecti
                        onIndex+1][gIndex];
                }
        }

        /* mutation */
        numberOfMutations=rand()%MaximumMutations;
                for(mutationIndex=0;mutationIndex<=numberOfMutations;mutatio
                nIndex++)
        {
                /* select chromosome for mutation */
                cIndex=(rand()%chromosomes)+1;

                if(rand()%2)
                {
                        /* select genes for mutation, lower half */
                        mutationGeneA=(rand()%(NumberOfLowerCoefficients))+
                        LocationOfLowerCoefficients+1;
                        mutationGeneB=(rand()%(NumberOfLowerCoefficients))+
                        LocationOfLowerCoefficients+1;
                }
                else
                {
                        /* select gene for mutation, upper half */
                        mutationGeneA=(rand()%(NumberOfUpperCoefficients))+
                    LocationOfUpperCoefficients+1;
                    mutationGeneB=(rand()%(NumberOfUpperCoefficients))+
                    LocationOfUpperCoefficients+1;
            }

            if(mutationGeneA)
            {
                    if(!mutationGeneB)
                    {
                            geneticPool[cIndex][mutationGeneA]=0.0;
                            geneticPool[cIndex][mutationGeneB]=1.0;
                    }
            }
            else
            {
                    if(mutationGeneB)
                    {
                            geneticPool[cIndex][mutationGeneA]=1.0;
                            geneticPool[cIndex][mutationGeneB]=0.0;
                    }
            }
    }

    for(cIndex=1;cIndex<=chromosomes;cIndex++)
    {
            totalCoefficients=0;
            for(gIndex=1;gIndex<=genes;gIndex++)
            {
                    totalCoefficients+=((unsigned)
                    geneticPool[cIndex][gIndex]);
            }

            if(totalCoefficients!=32)
            {
                    printf("Error, more or less than 32 coefficients >> %u.
                    \n",totalCoefficients);

                    getc(stdin);
            }
    }
}

/* transform the best solution back to the wavelet coefficients */
```

```
		for(gIndex=1;gIndex<=genes;gIndex++)
		{
			printf("%u",((unsigned) bestSolution[gIndex]));
			if(bestSolution[gIndex])
			{
				bestSolution[gIndex]=geneticPool[chromosomes+1][gIndex];
			}
			else
			{
				bestSolution[gIndex]=0.0;
			}
		}
		printf("\n");

		for(gIndex=1;gIndex<=genes;gIndex++)
		{
			printf("%f %f
			\n",bestSolution[gIndex],geneticPool[chromosomes+1][gIndex]);
		}


/*		wt1(bestSolution,genes,InverseTransform,daub4); */

		return(bestSolution);
}

float **InitializeGeneticAlgorithm(unsigned chromosomes, unsigned genes,
					char *inputSignal)
{
		unsigned cIndex;
		unsigned gIndex;
		unsigned scaleIndex;
		unsigned superSampleIndex;

		unsigned superSampleSize=32;

		unsigned unsignedInput;

		unsigned lowerCoefficientTotal;
		unsigned upperCoefficientTotal;

		unsigned totalCoefficients;
unsigned randomLocation;

float **geneticPool;

float *input;

float superSample;

float scaleEnergy;

float lowerCoefficientEnergy;
float upperCoefficientEnergy;

FILE *filePointer;

/* get the pool's memory */
geneticPool=matrix(1,(chromosomes+2),1,genes);

/* initialize the pool to zeros */
for(cIndex=1;cIndex<=chromosomes;cIndex++)
{
        for(gIndex=1;gIndex<=genes;gIndex++)
        {
                geneticPool[cIndex][gIndex]=0.0;
        }
}

/* get the input signal */
filePointer=fopen(inputSignal,"r");
if(!filePointer)
{
        printf("Exitting to dos... \n");
        printf("Cannot open input file. \n");

        exit(1);
}

/* get memory for input file */
input=vector(1,genes);

for(gIndex=1;gIndex<=genes;gIndex++)
{
        superSample=0.0;
```

```
		for(superSampleIndex=1;superSampleIndex<=superSampleSize;superSamp
		leIndex++)
		{
			fscanf(filePointer,"%u\n",&unsignedInput);
			superSample+=((float) unsignedInput);
		}
		input[gIndex]=(superSample/((float) superSampleSize));
	}

	fclose(filePointer);

	filePointer=fopen("z.dat","w");
	for(gIndex=1;gIndex<=genes;gIndex++)
	{
		fprintf(filePointer,"%f\n",input[gIndex]);
	}

	fclose(filePointer);

	/* perform the wavelet transform */
	wt1(input,genes,ForwardTransform,daub4);

	/* place the input file's wavelet coefficients into the first extra space
	 in the genetic pool */
	for(gIndex=1;gIndex<=genes;gIndex++)
	{
		geneticPool[chromosomes+1][gIndex]=input[gIndex];
		printf("%f. \n",geneticPool[chromosomes+1][gIndex]);
	}

	/* add the first 8 coefficients automatically to each chromosome */
	/* (the dc component of the signal and the first 2 scales) */
	for(cIndex=1;cIndex<=chromosomes;cIndex++)
	{
		for(gIndex=1;gIndex<=8;gIndex++)
		{
			geneticPool[cIndex][gIndex]=1.0;
		}
	}

	/* show the genetic pool */
/*	for(cIndex=1;cIndex<=chromosomes;cIndex++)
	{
        for(gIndex=1;gIndex<=genes;gIndex++)
        {
                printf("%u",((unsigned) geneticPool[cIndex][gIndex]));
        }
        printf("\n");
}
printf("\n"); */

/* calculated the energy for the remaining scales */
/* and put normalized coefficient values in the second extra space */
/* in the genetic pool */
for(scaleIndex=3;scaleIndex<=5;scaleIndex++)
{
        scaleEnergy=0.0;
        for(gIndex=(pow(2,scaleIndex)+1);gIndex<=pow(2,(scaleIndex+1));gInde
        x++)
        {
                scaleEnergy+=fabs(geneticPool[chromosomes+1][gIndex]);
        }

        for(gIndex=(pow(2,scaleIndex)+1);gIndex<=pow(2,(scaleIndex+1));gInde
        x++)
        {
                        geneticPool[chromosomes+2][gIndex]=fabs(geneticPool[ch
                        romosomes+1][gIndex])/scaleEnergy;
        }
}

randomize();

/* randomly initialize the genetic pool */
/* number of genes selected via InitialRandomization */

lowerCoefficientEnergy=0.0;
for(gIndex=(LocationOfLowerCoefficients+1);gIndex<=LocationOfUpperCoeffici
ents;gIndex++)
{
        lowerCoefficientEnergy+=fabs(geneticPool[chromosomes+1][gIndex]);
}

upperCoefficientEnergy=0.0;
for(gIndex=(LocationOfUpperCoefficients+1);gIndex<=genes;gIndex++)
{
        upperCoefficientEnergy+=fabs(geneticPool[chromosomes+1][gIndex]);
    }

    lowerCoefficientTotal=((unsigned)
    ceil((lowerCoefficientEnergy/(lowerCoefficientEnergy+upperCoefficientEnergy))*
    InitialRandomization));
    upperCoefficientTotal=((unsigned)
    floor((upperCoefficientEnergy/(lowerCoefficientEnergy+upperCoefficientEnergy))
    *InitialRandomization));

    printf("Lower coefficients:  %u. \n",lowerCoefficientTotal);
    printf("Upper coefficients:  %u. \n",upperCoefficientTotal);

    for(cIndex=1;cIndex<=chromosomes;cIndex++)
    {
        for(gIndex=1;gIndex<=lowerCoefficientTotal;gIndex++)
        {
            randomLocation=(rand()%(NumberOfLowerCoefficients))+Locatio
            nOfLowerCoefficients+1;
            while(geneticPool[cIndex][randomLocation])
            {
                randomLocation++;
                if(randomLocation>genes)
                {
                    randomLocation=LocationOfLowerCoefficients;
                }
            }
            geneticPool[cIndex][randomLocation]=1.0;

/*          printf("Random location:  %u. \n",randomLocation); */
        }

        for(gIndex=1;gIndex<=upperCoefficientTotal;gIndex++)
        {
                randomLocation=(rand()%(NumberOfUpperCoefficients))+
                LocationOfUpperCoefficients+1;
            while(geneticPool[cIndex][randomLocation])
            {
                randomLocation++;
                if(randomLocation>genes)
                {
                    randomLocation=LocationOfLowerCoefficients;
                }
				}
				geneticPool[cIndex][randomLocation]=1.0;

/*				printf("Random location:  %u. \n",randomLocation); */
			}

			totalCoefficients=0;
			for(gIndex=1;gIndex<=genes;gIndex++)
			{
				totalCoefficients+=((unsigned) geneticPool[cIndex][gIndex]);
			}

			printf("Total coefficients for chromosome:  %u >> %u.
			\n",cIndex,totalCoefficients);

		}

		/* show the genetic pool */
/*		for(cIndex=1;cIndex<=chromosomes;cIndex++)
		{
			for(gIndex=1;gIndex<=genes;gIndex++)
			{
				printf("%u",((unsigned) geneticPool[cIndex][gIndex]));
			}
			printf("\n");
		}
		printf("\n"); */

		return(geneticPool);
}

void main(int argc, char *argv[])
{
		unsigned index;

		float **geneticPool;

		float *output;
		float *signal;

		FILE *filePointer;

		if(argc!=4)
```

```
{
        printf("Exitting to dos... \n");
        printf("Incorrect number of input arguments. \n");

        exit(1);
}

/* initialize the genetic algorithm */
        geneticPool=InitializeGeneticAlgorithm(NumberOfChromosomes,Number
        OfGenes,argv[1]);

/* start the genetic algorithm */
output=GeneticAlgorithm(geneticPool,NumberOfChromosomes,NumberOfGenes)
;

filePointer=fopen(argv[2],"w");
if(!filePointer)
{
        printf("Exitting to dos... \n");
        printf("Cannot open output file  >> %s. \n",argv[2]);

        exit(1);
}

/* get some memory for the signal */
signal=vector(1,NumberOfGenes);

for(index=1;index<=NumberOfGenes;index++)
{
        signal[index]=output[index];

        printf("Signal  %u %f. \n",index,signal[index]);
}

wt1(signal,NumberOfGenes,InverseTransform,daub4);

for(index=1;index<=NumberOfGenes;index++)
{
        fprintf(filePointer,"%f\n",signal[index]);
}

fclose(filePointer);
```

```
filePointer=fopen(argv[3],"w");
if(!filePointer)
{
        printf("Exitting to dos... \n");
        printf("Cannot open output file  >> %s. \n",argv[3]);

        exit(1);
}

for(index=1;index<=NumberOfGenes;index++)
{
        output[index]=fabs(output[index]);
        geneticPool[NumberOfChromosomes+1][index]=fabs(geneticPool[Number
        OfChromosomes+1][index]);

}

fprintf(filePointer,"%f\n",output[1]/(geneticPool[NumberOfChromosomes+1][1]+
geneticPool[NumberOfChromosomes+1][2]));

fprintf(filePointer,"%f\n",output[2]/(geneticPool[NumberOfChromosomes+1][1]+
geneticPool[NumberOfChromosomes+1][2]));

fprintf(filePointer,"%f\n",output[3]/(geneticPool[NumberOfChromosomes+1][3]+
geneticPool[NumberOfChromosomes+1][4]));

fprintf(filePointer,"%f\n",output[4]/(geneticPool[NumberOfChromosomes+1][3]+
geneticPool[NumberOfChromosomes+1][4]));

for(index=5;index<=8;index++)
{
        fprintf(filePointer,"%f\n",(output[index])/
        (geneticPool[NumberOfChromosomes+1][5]+geneticPool[NumberOfChro
        mosomes+1][6]+geneticPool[NumberOfChromosomes+1][7]+geneticPool[
        NumberOfChromosomes+1][8]));
}

for(index=9;index<=16;index++)
{
        fprintf(filePointer,"%f\n",(output[index])/
        (geneticPool[NumberOfChromosomes+1][9]+geneticPool[NumberOfChro
        mosomes+1][10]+geneticPool[NumberOfChromosomes+1][11]+geneticPo
        ol[NumberOfChromosomes+1][12]+geneticPool[NumberOfChromosomes
```

```
		+1][13]+geneticPool[NumberOfChromosomes+1][14]+geneticPool[Numb
		erOfChromosomes+1][15]+geneticPool[NumberOfChromosomes+1][16]))
		;
	}

	for(index=17;index<=32;index++)
	{
		fprintf(filePointer,"%f\n",(output[index])/
		(geneticPool[NumberOfChromosomes+1][17]+
	geneticPool[NumberOfChromosomes+1][18]+geneticPool[NumberOfChromosom
	es+1][19]+geneticPool[NumberOfChromosomes+1][20]+
	geneticPool[NumberOfChromosomes+1][21]+geneticPool[NumberOfChromosom
	es+1][22]+geneticPool[NumberOfChromosomes+1][23]+geneticPool[NumberOfC
	hromosomes+1][24]+geneticPool[NumberOfChromosomes+1][25]+geneticPool[
	NumberOfChromosomes+1][26]+geneticPool[NumberOfChromosomes+1][27]+g
	eneticPool[NumberOfChromosomes+1][28]+
	geneticPool[NumberOfChromosomes+1][29]+geneticPool[NumberOfChromosom
	es+1][30]+geneticPool[NumberOfChromosomes+1][31]+geneticPool[NumberOfC
	hromosomes+1][32]));
	}

	for(index=33;index<=64;index++)
	{
		fprintf(filePointer,"%f\n",(output[index])/
		(geneticPool[NumberOfChromosomes+1][33]+geneticPool[NumberOfChr
		omosomes+1][34]+geneticPool[NumberOfChromosomes+1][35]+geneticP
		ool[NumberOfChromosomes+1][36]+
		geneticPool[NumberOfChromosomes+1][37]+geneticPool[NumberOfChro
		mosomes+1][38]+geneticPool[NumberOfChromosomes+1][39]+geneticPo
		ol[NumberOfChromosomes+1][40]+
		geneticPool[NumberOfChromosomes+1][41]+geneticPool[NumberOfChro
		mosomes+1][42]+geneticPool[NumberOfChromosomes+1][43]+geneticPo
		ol[NumberOfChromosomes+1][44]+
		geneticPool[NumberOfChromosomes+1][45]+geneticPool[NumberOfChro
		mosomes+1][46]+geneticPool[NumberOfChromosomes+1][47]+geneticPo
		ol[NumberOfChromosomes+1][48]+
		geneticPool[NumberOfChromosomes+1][49]+geneticPool[NumberOfChro
		mosomes+1][50]+geneticPool[NumberOfChromosomes+1][51]+geneticPo
		ol[NumberOfChromosomes+1][52]+
		geneticPool[NumberOfChromosomes+1][53]+geneticPool[NumberOfChro
		mosomes+1][54]+geneticPool[NumberOfChromosomes+1][55]+geneticPo
		ol[NumberOfChromosomes+1][56]+
		geneticPool[NumberOfChromosomes+1][57]+geneticPool[NumberOfChro
```

```
            mosomes+1][58]+geneticPool[NumberOfChromosomes+1][59]+geneticPo
            ol[NumberOfChromosomes+1][60]+
            geneticPool[NumberOfChromosomes+1][61]+geneticPool[NumberOfChro
            mosomes+1][62]+geneticPool[NumberOfChromosomes+1][63]+geneticPo
            ol[NumberOfChromosomes+1][64]);
    }

    fclose(filePointer);


    /* free the gene pool */
    free_matrix(geneticPool,1,NumberOfChromosomes,1,NumberOfGenes);
}
```

```
/* backpropagation neural network. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define StringLength 128

#define LearningRate 0.3

#define momentum 0.80

#define endThreshold 0.3

#define NR_END 1
#define FREE_ARG char*

int momentumControl=1;
int linearOutput=0;

struct neuralNetwork
{
        unsigned inputs;
        unsigned testInputs;
        unsigned selectedInput;

        unsigned inputNodes;
        unsigned hiddenNodes;
        unsigned outputNodes;

        float **inputs2Network;

        float **trainingPatterns;

        float **input2HiddenLayerWeights;
        float **hidden2OutputLayerWeights;

        float **previousInput2HiddenLayerWeights;
        float **previousHidden2OutputLayerWeights;

        float *hiddenLayerThresholds;
        float *outputLayerThresholds;
        float *hiddenLayer;
        float *outputLayer;
};


void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
        fprintf(stderr,"Numerical Recipes run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
}

float *vector(long nl, long nh)
/* allocate a float vector with subscript range v[nl..nh] */
{
        float *v;

        v=(float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
        if (!v) nrerror("allocation failure in vector()");
        return v-nl+NR_END;
}

void free_vector(float *v, long nl, long nh)
/* free a float vector allocated with vector() */
{
        free((FREE_ARG) (v+nl-NR_END));
        nh = nh + 0;  // Just included to quiet warning reports
}

float **matrix(long nrl, long nrh, long ncl, long nch)
/* allocate a float matrix with subscript rage m[nrl..nrh][ncl..nch] */
{
        long i, nrow=nrh-nrl+1, ncol=nch-ncl+1;
        float **m;

        /* allocate pointers to rows */
        m=(float **) malloc((size_t)((nrow+NR_END)*sizeof(float*)));
        if(!m) nrerror("allocation failure 1 in matrix()");
        m += NR_END;
        m -= nrl;
```

```
	/* allocate rows and set pointers to them */
	m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
	if(!m[nrl]) nrerror("allocation failure 2 in matrix()");
	m[nrl] += NR_END;
	m[nrl] -= ncl;

	for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

	/* return pointer to array of pointers to rows */
	return m;
}

void free_matrix(float **m, long nrl, long nrh, long ncl, long nch)
/* free a float matrix allocated by matrix() */
{
	free((FREE_ARG) (m[nrl]+ncl-NR_END));
	free((FREE_ARG) (m+nrl-NR_END));

	nrh += 0;
	nch += 0;
}

char *DeleteNewline(char *input)
{
	char *inputPointer;

	/* delete the newline from the string */
	inputPointer=input;
	while(*inputPointer) inputPointer++;
	*(inputPointer-1)=NULL;

	return(input);
}

void ShowVector(float *vector, unsigned elements)
{
	unsigned i;

	for(i=1;i<=elements;i++)
	{
		printf("Element %u --> %f.\n",i,vector[i]);
	}
}
```

```
void ShowMatrix(float **matrix, unsigned rows, unsigned columns)
{
        unsigned i;
        unsigned j;

        for(i=1;i<=rows;i++)
        {
                for(j=1;j<=columns;j++)
                {
                        printf("Row %u Column %u --> %f.\n",i,j,matrix[i][j]);
                }
        }
}

void RandomizeThresholds(float *thresholdVector,unsigned elements)
{
        unsigned i;

        /* randomize the thresholds */
        for(i=1;i<=elements;i++)
        {
                thresholdVector[i]=0.1*
                        (((float) rand())/((float) RAND_MAX));
        }

        /* show the vector to ensure proper operation */
/*      printf("Threshold vector: \n");
        ShowVector(thresholdVector,elements); */
}

void RandomizeWeights(float **weightMatrix,unsigned inputs, unsigned outputs)
{
        unsigned i;
        unsigned j;

        /* randomize the weights */
        for(i=1;i<=outputs;i++)
        {
                for(j=1;j<=inputs;j++)
                {
                        weightMatrix[i][j]=0.1*
                                (((float) rand())/((float) RAND_MAX));
                }
        }

        /* show the matrix to ensure proper operation */
/*      printf("Weight Matrix: \n");
        ShowMatrix(weightMatrix,outputs,inputs); */
}

void InitializePreviousWeights(float **previousWeightMatrix, unsigned inputs, unsigned
outputs)
{
        unsigned i;
        unsigned j;

        /* initialize all weights to zero */
        for(i=1;i<=outputs;i++)
        {
                for(j=1;j<=inputs;j++)
                {
                        previousWeightMatrix[i][j]=0.0;
                }
        }
}

void propagateInput2Output(struct neuralNetwork network)
{
        unsigned i;
        unsigned j;

        /* propagate input to hidden layer */
        for(i=1;i<=network.hiddenNodes;i++)
        {
                network.hiddenLayer[i]=0.0;
                for(j=1;j<=network.inputNodes;j++)
                {
                        network.hiddenLayer[i]=network.hiddenLayer[i]+
                                (network.input2HiddenLayerWeights[i][j]*
                                network.inputs2Network[network.selectedInput][j]);

/*                      printf("Input %f. Weight %f.
                        \n",network.inputs2Network[network.selectedInput][j],
                        network.input2HiddenLayerWeights[i][j]); */
                }
                network.hiddenLayer[i]=network.hiddenLayer[i]-network.hiddenLayerThr
                esholds[i];
/*              printf("Threshold:  %f. Output %f.
                \n",network.hiddenLayerThresholds[i],network.hiddenLayer[i]); */
                network.hiddenLayer[i]=(1.0/(1.0+exp(-network.hiddenLayer[i])));
/*              printf("Hidden layer output %f. \n",network.hiddenLayer[i]); */
        }

        /* propagate hidden to output layer */
        for(i=1;i<=network.outputNodes;i++)
        {
                network.outputLayer[i]=0.0;
                for(j=1;j<=network.hiddenNodes;j++)
                {
                        network.outputLayer[i]=network.outputLayer[i]+
                                        (network.hidden2OutputLayerWeights[i][j]*networ
                                        k.hiddenLayer[j]);

        /*              printf("Input %f. Weight %f.
                        \n",network.hiddenLayer[j],
                        network.hidden2OutputLayerWeights[i][j]);
                        printf("Acc %f. \n",network.outputLayer[i]); */
                }
                network.outputLayer[i]=network.outputLayer[i]-network.outputLayerThre
                sholds[i];
        /*      printf("Threshold:  %f. Output %f.
                \n",network.outputLayerThresholds[i],network.outputLayer[i]); */
                if(!linearOutput)
                {
                        network.outputLayer[i]=(1.0/(1.0+exp(-network.outputLayer[i])));
                }
/*              printf("Output output %f. \n",network.outputLayer[i]); */
        }
}

void UpdateWeightsAndThresholds(struct neuralNetwork network)
{
        unsigned i;
        unsigned j;
        unsigned k;

        float **temporaryWeights;

        float *deriavative;
        float summation;

        /* get some memory for the temporary weights */
        temporaryWeights=matrix(1,network.outputNodes,1,network.hiddenNodes);

        /* memory for deriavative calculation */
        deriavative=vector(1,network.outputNodes);

        /* copy the hidden2Output weights to a temporary location */
        for(i=1;i<=network.outputNodes;i++)
        {
                for(j=1;j<=network.hiddenNodes;j++)
                {
                        temporaryWeights[i][j]=network.hidden2OutputLayerWeights[i][j];
/*                      printf("%f %f
                        \n",temporaryWeights[i][j],
                        network.hidden2OutputLayerWeights[i][j]); */
                }
        }

        /* update output-hidden weights and thresholds */
        for(i=1;i<=network.outputNodes;i++)
        {
                if(!linearOutput)
                {
                        deriavative[i]=((-(network.trainingPatterns[network.selectedInput][
                        i]-
                                network.outputLayer[i]))*((network.outputLayer[i])*
                                (1.0-network.outputLayer[i])));
                }
                else
                {
                        deriavative[i]=((-(network.trainingPatterns[network.selectedInput][
                        i]-
                                network.outputLayer[i])));
                }

/*              printf("%u Deriavative:  %f. \n",i,deriavative[i]); */
        }

        for(i=1;i<=network.outputNodes;i++)
        {
                for(j=1;j<=network.hiddenNodes;j++)
```

```
		{
			if(momentumControl)
			{
				network.hidden2OutputLayerWeights[i][j]=
					network.hidden2OutputLayerWeights[i][j]+((-Learn
					ingRate)*
					(deriavative[i]*network.hiddenLayer[j]))+
					(momentum*network.previousHidden2OutputLayer
					Weights[i][j]);
			}
			else
			{
				network.hidden2OutputLayerWeights[i][j]=
					network.hidden2OutputLayerWeights[i][j]+((-Learn
					ingRate)*
					(deriavative[i]*network.hiddenLayer[j]));
			}

			network.previousHidden2OutputLayerWeights[i][j]=((-LearningRa
			te)*(deriavative[i]*network.hiddenLayer[j]));

	/*		printf("%u %u. Weight - hidden->output:  %f.
			\n",i,j,network.hidden2OutputLayerWeights[i][j]); */
		}
		network.outputLayerThresholds[i]=network.outputLayerThresholds[i]+
			((-LearningRate)*(deriavative[i]*(-1.0)));
	}

	for(j=1;j<=network.hiddenNodes;j++)
	{
		summation=0.0;
		for(i=1;i<=network.outputNodes;i++)
		{
			summation=summation+deriavative[i]*temporaryWeights[i][j];
		}

		for(k=1;k<=network.inputNodes;k++)
		{
			if(momentumControl)
			{
				network.input2HiddenLayerWeights[j][k]=
					network.input2HiddenLayerWeights[j][k]+
					((-LearningRate)*summation*
                        (network.hiddenLayer[j]*(1.0-network.hiddenLayer[
                        j]))*
                        (network.inputs2Network[network.selectedInput][k
                        ]))+
                            (momentum*network.previousInput2Hidden
                            LayerWeights[j][k]);
                }
                else
                {
                    network.input2HiddenLayerWeights[j][k]=
                        network.input2HiddenLayerWeights[j][k]+
                        ((-LearningRate)*summation*
                            (network.hiddenLayer[j]*(1.0-network.hidde
                            nLayer[j]))*
                        (network.inputs2Network[network.selectedInput][k
                        ]));
                }

                network.previousInput2HiddenLayerWeights[j][k]=
                    ((-LearningRate)*summation*
                    (network.hiddenLayer[j]*(1.0-network.hiddenLayer[j]))*
                    (network.inputs2Network[network.selectedInput][k]));

/*              printf("%u %u. Weight - input->hidden:  %f.
                \n",i,j,network.input2HiddenLayerWeights[i][j]); */

            }

            network.hiddenLayerThresholds[j]=
                network.hiddenLayerThresholds[j]+
                ((-LearningRate)*summation*
                (network.hiddenLayer[j]*(1.0-network.hiddenLayer[j]))*
                (-1.0));
        }

        /* free the deriavative vector */
        free_vector(deriavative,1,network.outputNodes);

        /* free the temporary weight matrix */
        free_matrix(temporaryWeights,1,network.outputNodes,1,network.hiddenNodes);
}

        void RearrangeInputs(struct neuralNetwork network)
```

```
{
        unsigned index;
        unsigned rearrangeIndex;

        unsigned selectedInputA;
        unsigned selectedInputB;

        float temporary;

/*      printf("%u. \n",((unsigned) (network.inputs/2.0))); */

        /* rearrange */
        for(rearrangeIndex=1;rearrangeIndex<=((unsigned)
        (network.inputs/2.0));rearrangeIndex++)
        {
                /* select the two inputs to swap */
                selectedInputA=(rand()%((unsigned) network.inputs))+1;
                selectedInputB=(rand()%((unsigned) network.inputs))+1;

/*              printf("Swap %u to %u and back. \n",selectedInputA,selectedInputB);
                getc(stdin); */

                /* swap the data */
                for(index=1;index<network.inputNodes;index++)
                {
                        temporary=network.inputs2Network[selectedInputA][index];
                        network.inputs2Network[selectedInputA][index]=
                                network.inputs2Network[selectedInputB][index];
                        network.inputs2Network[selectedInputB][index]=temporary;
                }
        }
}

void main(int argc, char *argv[])
{
        unsigned index;
        unsigned inputIndex;
        unsigned outputIndex;
        unsigned fileIndex;
        unsigned testIndex;
        unsigned showIndex=1000;

        unsigned i;
unsigned j;

unsigned mode;
unsigned inputFiles;
unsigned testFiles;
unsigned networkInputs;
unsigned networkOutputs;
unsigned hiddenNodes;

float **input2HiddenLayerWeights;
float **hidden2OutputLayerWeights;

float **previousInput2HiddenLayerWeights;
float **previousHidden2OutputLayerWeights;

float **inputData;

float **trainingPatterns;

float *outputLayer;
float *hiddenLayer;

float *hiddenLayerThresholds;
float *outputLayerThresholds;

float *randomizationCheck;

float fData;

float error;
float testError;

char *input;

FILE *dataFilePointer;
FILE *inputFilePointer;
FILE *errorFilePointer;
FILE *networkFilePointer;
FILE *progressFilePointer;
FILE *checkFilePointer;

struct neuralNetwork network;
```

```
/* check input arguments */
if(argc!=4)
{
        printf("Number of input arguments is incorrect. \n");
        printf("Require a input data file. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

/*open the data file */
dataFilePointer=fopen(argv[1],"r");
if(!dataFilePointer)
{
        printf("Data file not found. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

/* get a string of length StringLength for input */
input=((char *) malloc(sizeof(char)*StringLength));
if(!input)
{
        printf("Cannot allocate enough memory. \n");
        printf("Exitting to dos... \n");

        exit(1);
}

/* read the mode type:  1 - train, 2 - classify */
mode=atoi(fgets(input,StringLength,dataFilePointer));
printf("Network Mode:  ");
switch(mode)
{
        case 1: printf("Training mode. \n"); break;
        case 2: printf("Classify mode. \n"); break;

        default:  printf("No mode selected. \n");
                                printf("Exitting to dos... \n");

                                exit(1);
}
```

```
/* get the number of input files */
inputFiles=atoi(fgets(input,StringLength,dataFilePointer));

/* get the number of testing files */
testFiles=atoi(fgets(input,StringLength,dataFilePointer));

/* get the number of network inputs */
networkInputs=atoi(fgets(input,StringLength,dataFilePointer));

/* get the number of hidden layer nodes */
hiddenNodes=atoi(fgets(input,StringLength,dataFilePointer));

/* get the number of network outputs */
networkOutputs=atoi(fgets(input,StringLength,dataFilePointer));

/* report new inputs to screen */
printf("Number of input files: %u. \n",inputFiles);
printf("Number of test files: %u. \n",testFiles);
printf("Number of network inputs: %u. \n",networkInputs);
printf("Number of hidden layer nodes: %u. \n",hiddenNodes);
printf("Number of network outputs: %u. \n",networkOutputs);

/* get necessary memory for the network */
/* memory for each layer */
hiddenLayer=vector(1,hiddenNodes);
outputLayer=vector(1,networkOutputs);

/* memory for thresholds */
hiddenLayerThresholds=vector(1,hiddenNodes);
outputLayerThresholds=vector(1,networkOutputs);

/* memory for input data */
inputData=matrix(1,inputFiles+testFiles,1,networkInputs);

/* memory for weights */
input2HiddenLayerWeights=matrix(1,hiddenNodes,1,networkInputs);
hidden2OutputLayerWeights=matrix(1,networkOutputs,1,hiddenNodes);

/* memory for the previous weights for momentum calculation */
previousInput2HiddenLayerWeights=matrix(1,hiddenNodes,1,networkInputs);
previousHidden2OutputLayerWeights=matrix(1,networkOutputs,1,hiddenNodes);

/* memory for training patterns if necessary */
```

```
if(mode==1)
{
        trainingPatterns=matrix(1,inputFiles+testFiles,1,networkOutputs);
}

/* load the neural network structure */
/* parameters */
network.inputs=inputFiles;
network.testInputs=testFiles;
network.selectedInput=1;
network.inputNodes=networkInputs;
network.hiddenNodes=hiddenNodes;
network.outputNodes=networkOutputs;

/* input data */
network.inputs2Network=inputData;

/* traing data */
network.trainingPatterns=trainingPatterns;

/* weights */
network.input2HiddenLayerWeights=input2HiddenLayerWeights;
network.hidden2OutputLayerWeights=hidden2OutputLayerWeights;

/* previous weights for the calculation of momentum */
network.previousInput2HiddenLayerWeights=previousInput2HiddenLayerWeights
;
network.previousHidden2OutputLayerWeights=previousHidden2OutputLayerWei
ghts;

/* thresholds */
network.hiddenLayerThresholds=hiddenLayerThresholds;
network.outputLayerThresholds=outputLayerThresholds;

/* layers */
network.hiddenLayer=hiddenLayer;
network.outputLayer=outputLayer;

/* open progress file if in classification mode */
if(mode==2)
{
        progressFilePointer=fopen(argv[3],"w");
        if(!progressFilePointer)
        {
                printf("Cannot write to file. \n");
                printf("Exitting to dos... \n");

                exit(1);
        }
}

/* get the data from the files and put it in memory */
for(inputIndex=1;inputIndex<=inputFiles+testFiles;inputIndex++)
{
        input=DeleteNewline(fgets(input,StringLength,dataFilePointer));
        inputFilePointer=fopen(input,"r");
        if(!inputFilePointer)
        {
                printf("Could not open file:  %s. \n", input);
                printf("Exitting to dos... \n");

                exit(1);
        }
        else
        {
                printf("Filename:  %s.  File %u of %u.
                \n",input,inputIndex,inputFiles);
        }

        /* read the data from the file */
        for(fileIndex=1;fileIndex<=networkInputs;fileIndex++)
        {
                fscanf(inputFilePointer,"%f\n",&fData);
                inputData[inputIndex][fileIndex]=fData;
/*              printf("%u %u %f. \n",inputIndex,fileIndex,
                                inputData[inputIndex][fileIndex]); */
        }

        /* close the input file */
        fclose(inputFilePointer);

        /* use a trained network */
        if(mode!=1)
        {
                /* open the file */
                networkFilePointer=fopen(argv[2],"r");
```

```
if(!networkFilePointer)
{
        printf("Could not open file:  %s. \n", input);
        printf("Exitting to dos... \n");

        exit(1);
}

for(i=1;i<=hiddenNodes;i++)
{
        for(j=1;j<=networkInputs;j++)
        {
                fscanf(networkFilePointer,"%f\n",&fData);
                input2HiddenLayerWeights[i][j]=fData;
        }
}

/* hidden to output weights */
for(i=1;i<=networkOutputs;i++)
{
        for(j=1;j<=hiddenNodes;j++)
        {
                fscanf(networkFilePointer,"%f\n",&fData);
                hidden2OutputLayerWeights[i][j]=fData;
        }
}

/* hidden thresholds */
for(i=1;i<=hiddenNodes;i++)
{
        fscanf(networkFilePointer,"%f\n",&fData);
        hiddenLayerThresholds[i]=fData;
}

/* output thresholds */
for(i=1;i<=networkOutputs;i++)
{
        fscanf(networkFilePointer,"%f\n",&fData);
        outputLayerThresholds[i]=fData;
}

/* close the file */
fclose(networkFilePointer);
```

```
                    /* select the input */
                    network.selectedInput=inputIndex;

                    /* propagate the input */
                    propagateInput2Output(network);

                    /* show the output */
                    fprintf(progressFilePointer,"%s  %u/%u
                    ",input,inputIndex,inputFiles);
                    for(i=1;i<=networkOutputs;i++)
                    {
                            printf("I%d - %f ",i,outputLayer[i]);

                            fprintf(progressFilePointer,"%f  ",outputLayer[i]);
                    }
                    printf("\n");

                    fprintf(progressFilePointer,"\n");

                    /* increment the current input counter */
/*                  network.selectedInput=InputIndex; */
            }
            else
            {
                    /* read the training pattern */
                    for(outputIndex=1;outputIndex<=networkOutputs;outputIndex++)
                    {
                            fscanf(dataFilePointer,"%f\n",&fData);
                            trainingPatterns[inputIndex][outputIndex]=fData;
                    }
            }
    }

    /* if not training, the program is done, so terminate it */
    if(mode==2)
    {
            fclose(progressFilePointer);

            exit(1);
    }


    /* display for debugging */
/*      printf("Input data matrix:  \n");
        ShowMatrix(inputData,inputFiles,networkInputs); */

/*      if(mode==1)
        {
                printf("Training data matrix:  \n");
                ShowMatrix(trainingPatterns,inputFiles,networkOutputs);
        } */

        /* start the training if necessary */
        if(mode==1)
        {
                /* initialize the weights and thresholds */
                /* set the random number generator once */
                randomize();

                /* randomize the weights */
                RandomizeWeights(input2HiddenLayerWeights,networkInputs,hiddenNod
                es);
                RandomizeWeights(hidden2OutputLayerWeights,hiddenNodes,networkOu
                tputs);

                /* initialize the previous weights for momentum calculation */
                InitializePreviousWeights(previousInput2HiddenLayerWeights,networkInp
                uts,hiddenNodes);
                InitializePreviousWeights(previousHidden2OutputLayerWeights,hiddenNo
                des,networkOutputs);

                /* randomize the thresholds */
                RandomizeThresholds(hiddenLayerThresholds,hiddenNodes);
                RandomizeThresholds(outputLayerThresholds,networkOutputs);

                /* open a file for reporting the errors */
                errorFilePointer=fopen(argv[3],"w");
                if(!errorFilePointer)
                {
                        printf("Cannot open file. \n");
                        printf("Exitting to dos... \n");

                        exit(1);
                }

                /* allocate and initialize the randomization check */
                randomizationCheck=vector(1,inputFiles);

                for(index=1;index<=inputFiles;index++)
                {
                        randomizationCheck[index]=0.0;
                }

                /* train the network */
                for(index=1;index<=25000;index++)
                {
                        error=0.0;
                        for(inputIndex=1;inputIndex<=inputFiles;inputIndex++)
                        {
                                /* rearrange inputs */
/*                              RearrangeInputs(network); */

                                /* set the input file selector */
                                network.selectedInput=inputIndex;
                                /* ((rand()%inputFiles)+1); */
                                /* increment the randomization check */
                                randomizationCheck[network.selectedInput]++;

                                /* propagate the input to the output */
                                propagateInput2Output(network);

                                /* update the weights and thresholds */
                                UpdateWeightsAndThresholds(network);

                                /* show the calculated output and etc. */
/*                              printf("Epoch index:  %u, Input index:  %u.
                                \n",index,inputIndex); */
                                for(outputIndex=1;outputIndex<=networkOutputs;outputIn
                                dex++)
                                {
                                        if(index==showIndex)
                                        {
        /*                                      printf("Calculated output:  %f
                                                ",outputLayer[outputIndex]);
                                                printf("target output:  %f for output:  %u.
                                                \n",trainingPatterns[inputIndex]
                                                [outputIndex],
                                                outputIndex); */
                                        }
                                        error+=pow((trainingPatterns[inputIndex][outputInd
                                        ex]-outputLayer[outputIndex]),2);
                                }
                                /* show and write the errors associated with this network */
/*                              printf("Error this epoch: %u--> %f. \n",index,error); */
/*                              printf("Inputs: %f %f. \n \n",inputData[inputIndex][1],
                                inputData[inputIndex][2]); */
/*                              getc(stdin); */
                        }
                        if(index==showIndex)
                        {
                                /* test the data */
                                printf("Testing the data... ");
                                testError=0.0;
                                for(testIndex=(inputFiles+1);testIndex<=(inputFiles+testFile
                                s);testIndex++)
                                {
                                        /* select the file */
                                        network.selectedInput=testIndex;

                                        /* propagate input to the output */
                                        propagateInput2Output(network);

                                        printf("Input Number: %u. \n",testIndex);

                                        /* calculate the error */
                                        for(outputIndex=1;outputIndex<=network.outputN
                                        odes;outputIndex++)
                                        {
                                                printf("Output: %u > Error: %f >> Actual:
                                                %f.
                                                \n",outputIndex,network.outputLayer[output
                                                Index],
                                                        network.trainingPatterns[network.sel
                                                        ectedInput][outputIndex]);

                                                testError+=fabs(network.outputLayer[outpu
                                                tIndex]-
                                                        network.trainingPatterns[network.sel
                                                        ectedInput][outputIndex]);
                                        }
                                        printf("\n");
                                }
                                printf("Error:  %f, average error:  %f >>> %u  >> %f. \n",
                                          testError,testError/((float)
                                         testFiles),index,LearningRate);

                                /* stop if less than threshold */
                                if((testError/((float) testFiles))<(endThreshold))
                                index=100000;

                                fprintf(errorFilePointer,"%f\n",(testError/((float)
                                testFiles)));
                                showIndex+=1000;
                        }
                }
                fclose(errorFilePointer);

                checkFilePointer=fopen("randchk.dat","w");
                if(!checkFilePointer)
                {
                        printf("Could not open file. \n");
                        printf("Exitting to dos... \n");

                        exit(1);
                }

                for(index=1;index<=inputFiles;index++)
                {
                        fprintf(checkFilePointer,"%u %u\n",index,((unsigned)
                        randomizationCheck[index]));
                }

                fclose(checkFilePointer);
        }

        /* save the network */
        /* open the file */
        networkFilePointer=fopen(argv[2],"w");
        if(!networkFilePointer)
        {
                printf("Could not open file:  %s. \n",argv[2]);
                printf("Exitting to dos... \n");

                exit(1);
```

```
}

/* input to hidden weights */
for(i=1;i<=hiddenNodes;i++)
{
        for(j=1;j<=networkInputs;j++)
        {
                fprintf(networkFilePointer,"%f\n",input2HiddenLayerWeights[i][j])
                ;
        }
}
printf("Hidden weights have been saved. \n");


/* hidden to output weights */
for(i=1;i<=networkOutputs;i++)
{
        for(j=1;j<=hiddenNodes;j++)
        {
                fprintf(networkFilePointer,"%f\n",hidden2OutputLayerWeights[i][j]
                );
        }
}
printf("Output weights have been saved. \n");

/* hidden thresholds */
for(i=1;i<=hiddenNodes;i++)
{
        fprintf(networkFilePointer,"%f\n",hiddenLayerThresholds[i]);
}
printf("Hidden thresholds have been saved. \n");

/* output thresholds */
for(i=1;i<=networkOutputs;i++)
{
        fprintf(networkFilePointer,"%f\n",outputLayerThresholds[i]);
}
printf("Output thresholds have been saved. \n");

/* close the file */
fclose(networkFilePointer);

/* free the memory */
```

```
        free_vector(hiddenLayer,1,hiddenNodes);
        free_vector(outputLayer,1,networkOutputs);

        free_vector(hiddenLayerThresholds,1,hiddenNodes);
        free_vector(outputLayerThresholds,1,networkOutputs);

        free_matrix(inputData,1,networkInputs,1,inputFiles+testFiles);

        free_matrix(input2HiddenLayerWeights,1,hiddenNodes,1,networkInputs);
        free_matrix(hidden2OutputLayerWeights,1,networkOutputs,1,hiddenNodes);

        free_matrix(previousInput2HiddenLayerWeights,1,hiddenNodes,1,networkInputs);
        free_matrix(previousHidden2OutputLayerWeights,1,networkOutputs,1,hiddenNod
        es);

        if(mode==1)
        {
                free_matrix(trainingPatterns,1,inputFiles+testFiles,1,networkOutputs);
        }

        /* close the data file */
        fclose(dataFilePointer);
}
```