ADAPTIVE NOISE CANCELLING

APPLIED TO

MACHINE CONDITION MONITORING

by PAUL GRAHAM BREMER

Submitted to the University of Cape Town in partial fulfilment of the requirements for the degree of Master of Science in Engineering April 1990.

ACKNOWLEDGEMENTS

First of all I would like to thank Adrian Jongens, my supervisor. His encouragement and ideas were very helpful.

A word of thanks must go to Keren Bracey, for acting as my editor. The smooth flow of the text is due to her critical eye.

I would like to thank Professor de Jager for ordering the TMS 320C25 boards, without which there would have been no thesis.

Thanks are also due to Rae Perl and Graham Jack for lending their computers to me for conducting the experiments on the rolling road and on the bearing-monitoring rig.

Finally, thanks must go to Lynn Smallwood and Mark Hayworth of Shell South Africa for allowing me to use their computers to produce the graphics and text.

TERMS OF REFERENCE

In the first week of January 1988, the University of Cape Town instructed me to produce a thesis concerning machine-condition monitoring. My instructions were:

1) To start the thesis in the second week of January 1988.
2) To write a real-time Adaptive-Noise-Cancelling package using a TMS 320C25 digital signal-processing chip.
3) To evaluate the package in a machine-monitoring environment.

It was required that the thesis be handed in by the 30th of March 1990.

SYNOPSIS

The objective of this thesis is to determine whether Adaptive Noise Cancelling can be used successfully in determining the state of machine elements. In addition, this thesis was used to gain experience in real-time computing. This was done by designing and building a real-time machine monitoring package using an IBM PC and a TMS 320C25 digital signal-processing chip manufactured by Texas Instruments.

To determine which adaptive algorithm should be used in the package, experiments were carried out on a computer with different types of adaptive noise cancelling algorithms, the two main ones being the Least-Mean-Squares (LMS) and Recursive-Least-Squares (RLS) algorithms.

It was concluded from the experiments that although the RLS algorithms provided a greater degree of noise cancelling than the LMS method, they were not suitable for high-speed real-time applications. The RLS algorithms require a large number of machine cycles to perform updates of the filter weights, which limits the input signal bandwidth. They are also unstable due to the finite precision of the computer and so the LMS algorithm was chosen to be incorporated into the machine monitoring package.

A real-time machine-monitoring package was written in Turbo C, with the TMS 320C25 routines being written in Assembler language. The data analysis that occurs in the machine monitoring package consists of kurtosis measurement and recording, the spectrum and cepstrum of the signal also being recorded.

Two analogue input channels are required for adaptive noise-cancelling since there are two sensors used: one is placed at the machine element to be monitored and the other one is placed at the source of interference. However, only one channel was available on the computer, so a two-channel multiplexer was built to overcome this difficulty.

Adaptive Noise Cancelling was tested on the Rolling Road at the University of Cape Town. It was hoped that it would be able to detect car engine knock but this did not occur for the reason that no noise cancelling results if the time taken for the machine element signal to travel from one sensor to the other is greater than the time taken for it to pass through the adaptive filter. A remedy for this situation would be to slow the noise-canceller down so that the signal spends longer in the filter.

Noise Cancelling was also tested on a rig consisting of a motor coupled to three bearings on one shaft. It was the noise canceller's task to determine whether on the the bearings was worn and if so, which one was worn. The state of the bearing was determined by observing the kurtosis of the signal produced. With no noise cancelling operating, all three bearings produced signals with high kurtosis values. However, with noise cancelling in operation only the middle bearing produced a high kurtosis value; the other two gave low readings.

Therefore it was determined, first of all, that the middle bearing was worn and, secondly, that Adaptive Noise Cancelling does work when applied to machine monitoring.

LIST OF ILLUSTRATIONS

NOMENCLATURE

These symbols are listed in the order in which they appear in this dissertation.

Symbol		Page
$d(t)$	Present sample of the primary signal.	2.2
$u(t)$	The part of the primary signal that is not correlated to the reference signal.	2.2
$W^*_{N,T}$	Optimum filter weights.	2.2
$y(t)$	Present sample of the reference signal.	2.2
N	Length of the adaptive filter.	2.2
T	Discrete time index.	2.2
$W_{N,T}$	Adaptive filter weights.	2.3
$\mu(t)$	Convergence factor used in both the LMS and RLS algorithms.	2.3
$Y_N(T)$	Samples of the reference signal in the adaptive filter.	2.3
$O(N)$	Of the order of N.	2.4
$\Phi_N(T)$	Sum of the squared errors taken over all time.	2.4
δ	Forgetting factor used in the RLS algorithm.	2.4
y_T	Vector of M values of the reference signal. $M >> T$. $[y(t)\ y(t-1)\ \ldots\ y(0)\ 0\ldots0]'$	2.5
d_T	Vector of M values of the primary signal. $M >> T$. $[d(t)\ d(t-1)\ \ldots\ d(0)\ 0\ldots0]$	2.5
$Y_{N,T}$	An M x N matrix of reference signal values. $[\ Y_T,\ Y_{T-1},\ \cdots,\ Y_{T-N+1}\ ]$	2.5
$\epsilon_{N,T}$	Primary signal added to the filtered reference signal. The reference signal is taken over all time. $d(t) + Y_{N,T}W_{N,T}$	2.5
R^M	M-dimensional vector space.	2.5
$R^{-1}_{N,T}$	Autocorrelation matrix. $(Y'_{N,T}Y_{N,T})^{-1}$	2.6

Symbol		Page
$P_{Y(N,T)}$	Projection matrix. $Y_{N,T}(Y'_{N,T}Y_{N,T})Y'_{N,T}$	2.6
$P_{N,T}$	Projection matrix, same as $P_{Y(N,T)}$.	2.6
$P'_{N,T}$	Inverse projection matrix.	2.6
$K_{N,T}$	A term defined to make the arithmetic simpler. $Y_{N,T}R^{-1}{}_{N,T}$	2.6
σ	Pinning vector.	2.7
$A_{N,T}$	Forward prediction vector.	2.7
$B_{N,T}$	Backward prediction vector.	2.7
$C_{N,T}$	Kalman gain vector.	2.7
$e_N(T)$	Filtered residual: forward prediction error.	2.7
$r_N(T)$	Filtered residual: backward prediction error.	2.7
$\tau_N(T)$	Filtered residual: pinning vector prediction error.	2.7
$\epsilon_N(T)$	The filtered error in predicting the primary signal.	2.7
$\alpha_N(T)$	Minimised squared length of the forward-prediction error vector.	2.8
$\beta_N(T)$	Minimised squared length of the backward-prediction error vector.	2.8
$e^P{}_N(T)$	Prediction residual: the error in forward prediction.	2.8
$r^P{}_N(T)$	Prediction residual: the error in backward prediction.	2.8
$\epsilon^P{}_N(T)$	Prediction residual: the error in predicting the primary signal.	2.8
$p_k(T)$	A term used in Table 2.1 to make the notation simpler, which is eqivalent to: $Y'_T - kY_T$	2.14
β	a) The current gain of a transistor.	5.2
	b) A multiplication factor in determining resonant frequencies of a beam.	6.11

Table of Contents

CHAPTER 1

INTRODUCTION

Machine condition monitoring is performed mostly by recording the signal produced by a machine and then processing it much later. However this is not the optimal arrangement, since it is time-consuming and often it would be preferable to obtain an immediate estimate of the machine's condition. Therefore it was decided to design a real-time signal processing package for the purpose of obtaining on-line information.

A special feature of this package is its adaptive noise cancelling algorithm. The reason for the noise cancelling is described in section 1.2. It is intended that the package designed in this thesis will form the front end of an expert system which will be able to determine the status of the element being monitored by automatically analysing the various statistical and frequency results.

1.1 WHY ADAPTIVE NOISE CANCELLING?

In industry almost every machine with moving parts needs some kind of monitoring in order to determine its state of repair. This is normally done by placing an electronic sensor, such as an accelerometer or a sound-level meter at or near the machine element to be monitored. After analysis of the signal it can be determined with varying degrees of certainty what condition the machine element is in.

However, a problem often occurring with recording signals in this way is that the signal from the monitored element is often swamped by the noise from the rest of the machine. Evidently some kind of noise cancelling is needed if a reliable result is to be obtained.

If the element being observed produces a periodic signal then a variety of conventional noise-cancelling methods can be employed, such as averaging. An example of an element producing a periodic waveform would be a gear, where the signal repeats itself once every revolution. But if the signal being produced is not periodic, then averaging cannot be used. When, for example, a ball bearing is rotating, each ball produces its own signal.

One of the noise cancelling methods suited for this kind of signal is Adaptive Noise Cancelling. Adaptive Noise Cancelling is used with signals where nothing about either the noise or the signal is known, and it is this versatility that makes it so attractive for a software package.

Another noise-cancelling technique used is the Maximum Entropy method. However, this is used only for spectral estimates and cannot be used for cleaning the signal prior to statistical analysis, e.g. kurtosis and RMS measurement.

1.2 THE OBJECTIVES OF THIS THESIS

There were four objectives in mind:

1) To determine whether the Fast Recursive-Least-Squares Adaptive-Filtering algorithm is suitable for real-time signal processing with today's available Signal-Processing chip speeds,
2) To test whether Adaptive Noise-Cancelling would work in the area of Machine Condition Monitoring,
3) To determine if the kurtosis of a signal is a sufficient criterion in estimating machine element conditions, and
4) To develop a real-time machine monitoring package that could test these two theories.

1.3 CONSISTUENTS OF THIS THESIS

The thesis is divided into two parts. The first is a dissertation containing an explanation of the latest types of adaptive-filtering algorithms along with the results obtained. Since 1983 there have been rapid developments in the area of so-called Fast Recursive Least Squares (RLS) algorithms. Since they are fairly complicated in their concept it was decided that an explanation was necessary. One of the motives for including this algorithm in this thesis was to determine whether the latest signal processors were fast enough to handle their somewhat lengthy calculations.

The rest of the dissertation is a description of the package developed for machine condition-monitoring. Also, experiments were carried out on various machines to test the concept of noise cancelling applied to machine monitoring and these are also described, along with the results obtained.

The other part of the thesis consists of the package itself, which is made up of a program and a multiplexer.

The program consists of a software package written for the TMS 320C25 digital signal processor, a fast microchip which operates at a speed of 10 million instructions per second. The TMS 320C25 in this thesis is incorporated into a board manufactured by Dalanco-Spry, which has been inserted into an IBM personal computer. The software consists principally of an Adaptive Noise Canceller. In addition it contains extra signal analysis routines, namely a Kurtosis meter and displays of the spectrum and cepstrum of the recorded signal.

The program for the TMS 320C25 was written in assembly-language and the overall managing program was written in TURBO C. To ease the tedium of converting an assembly-

language program into a hexadecimal string that can be incorporated into a TURBO C program, a special utility was written, also in TURBO C, to perform this function automatically.

1.4 STRUCTURE OF THIS DISSERTATION

Chapter 2 is an explanation of the basic RLS algorithm, along with the latest developments in the field. This is followed in chapter 3 by documentation of the results obtained from the RLS and earlier algorithms. Chapter 4 consists of a description of the programs written for the monitoring package. These programs include, besides the actual monitoring program, a utility program to ease the path of future programmers of the TMS 320C25.

In chapter 5 the multiplexer designed for use with the Dalanco-Spry board is described, while in chapter 6 experiments with the machine monitoring package are evaluated. Finally in chapter 7 conclusions are made and possible improvements to the machine monitoring package are suggested.

A floppy disc is included at the back of this dissertation, which contains all the programs used in the thesis. To obtain a list of these programs, insert the disc in your computer and enter "README". A list is also given in the table of contents.

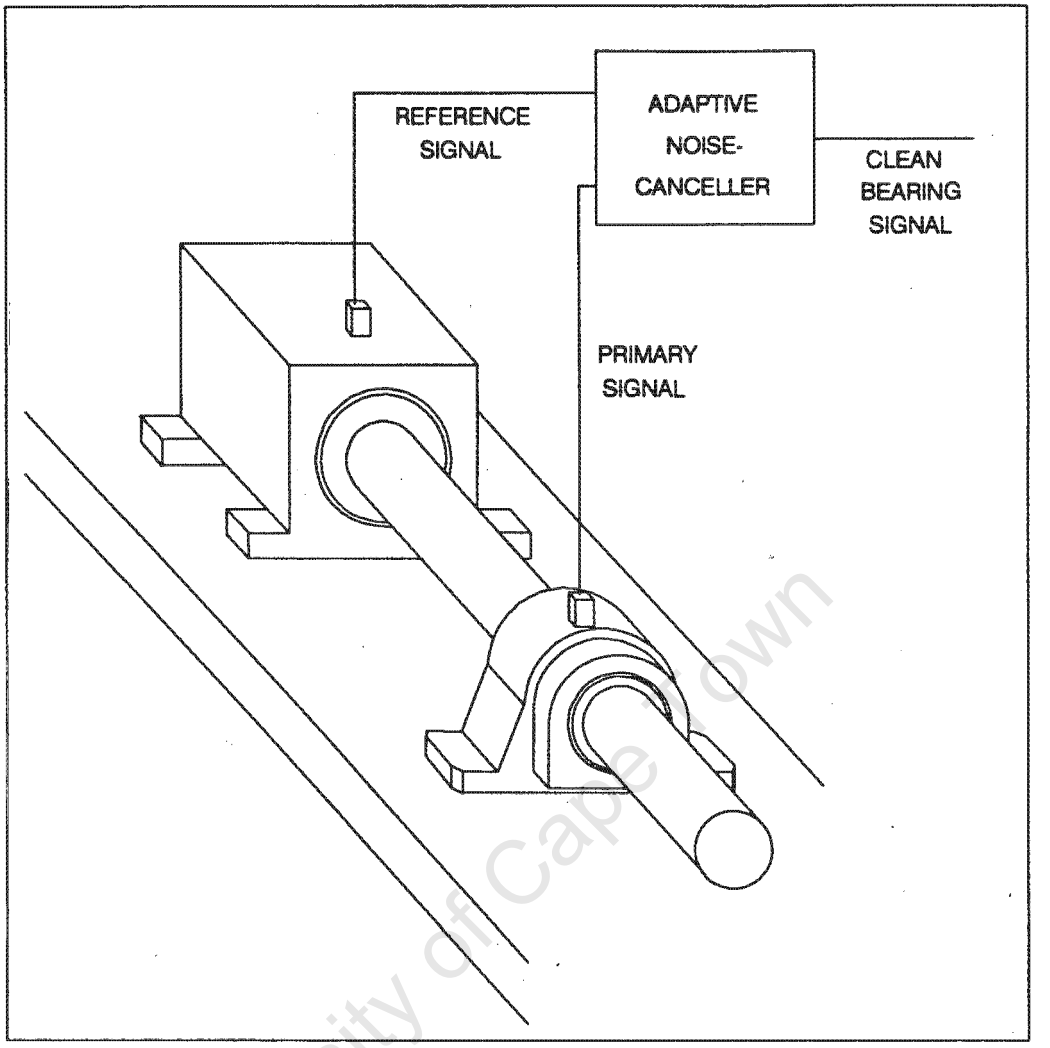


Figure 2.1 Finding a Bearing Signal by Adaptive Noise Cancelling

Chapter 2

ADAPTIVE FILTERING AND THE RECURSIVE-LEAST-SQUARES ALGORITHM

2.1 ADAPTIVE NOISE CANCELLING APPLIED TO MACHINE MONITORING

As stated in the introduction the signal from an element under test is often swamped by noise coming from the machine on which the element is located and it is necessary for this noise to be removed before analysis can be performed.

In figure 2.1 a bearing is producing a signal by virtue of its rotation and a sensor has been placed on the bearing to monitor this signal. However, a short distance away a motor is producing a large amount of noise and this finds its way onto the sensor placed on the bearing. How is a clean signal obtained from this noisy one?

The answer is to place a second sensor close to the noise source. This sensor picks up the noise and, ideally, no bearing signal; it provides a reference by which the noise in the bearing sensor may be determined. It is for this reason that this sensor is called the Reference sensor and the noise it detects is called the Reference Signal. The sensor located at the bearing is known as the Primary sensor and its signal is called the Primary Signal.

By the time the noise has reached the Primary Sensor it has changed because the path through which it has travelled acts like a filter. Therefore no noise-cancelling can be achieved simply by subtracting the waveform at the Reference Sensor from that at the Primary Sensor. The Reference signal is first passed through the adaptive filter, which adapts to become a very good representation of the path filter. This

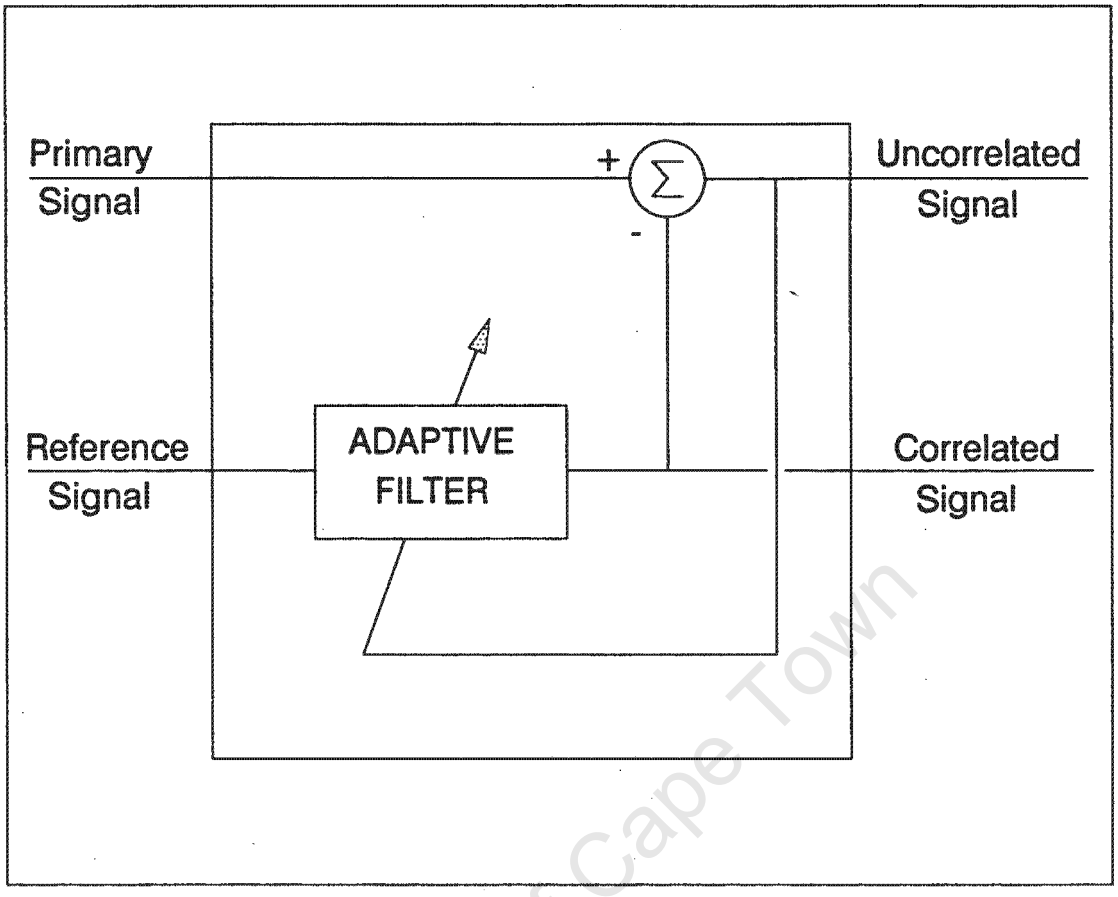


Figure 2.2 Diagram of the Adaptive Noise Cancelling System

filtered signal can then be subtracted from the Primary signal for maximum noise cancellation.

If some of the bearing signal is detected by the Reference sensor the estimate of the clean signal will be degraded depending on how much of the signal is detected. This is because of the time taken for the bearing signal to reach the Reference sensor. Because of this delay it is not correlated with either the Primary signal or the noise. It is therefore regarded by the adaptive filter as being part of the signal and it is thus added to the Primary signal, degrading the estimate of the clean signal.

2.2 THE ADAPTIVE FILTERING CONCEPT

Adaptive filtering is performed digitally and so all the filters mentioned in this explanation are digital, not analogue. The purpose of the adaptive filter is to rearrange the Primary and Reference signals into two parts: the part that is common to both signals and the part that is not. It is the uncorrelated part that is assumed by the adaptive filter to be the signal produced by the element in question, since the common signal should be the background interference cause by the machine.

Separating the signals is performed in the following way. The adaptive filter is a set of multipliers, called weights, that act on the N most recent samples of the Reference signal, where N is the number of weights in the filter. The filter tries to predict the primary signal as accurately as possible, by adjusting its weights to produce a linear filter that changes each of the frequencies in the Reference signal with regard to amplitude and phase, so that the Reference signal best matches the Primary signal. The filtering system is shown in figure 2.2.

Expressed in mathematical terms the Primary signal, d(T), is:

$$d(T) = u(T) - W^*_{N,T}\,[y(T)\ y(T-1)\ \ldots\ y(T-N+1)]^T$$

The optimum filter weights are denoted by the 1 x N vector $W^*_{N,T}$, where T is the discrete time index and N is the number of weights in the filter. The Primary signal is assumed to be made up of two parts: the uncorrelated signal u(T), which is produced by the component in question, and a linear combination of the samples of the Reference signal, y(T), that are found in the filter. It is the task of the adaptive algorithm to transform its filter's weights into $W^*_{N,T}$, the path filter formed by the shaft and bench found in figure 2.1.

The weights are updated using the uncorrelated signal, u(T), according to the following formula:

$$W_{N,T+1} = W_{N,T} + \mu(T)u(T)Y_N(T)$$

where $\mu(T)$ is a factor determining how much each weight is updated. It can be fixed, as in the case of the Least-Mean-Squares algorithm (LMS) [1], [2], or it can be variable as in the Recursive-Least-Squares algorithm (RLS) which makes the filtering fairly complicated.

2.3 REASONS FOR USING THE RLS ALGORITHM

The reasons for using the RLS technique instead of the LMS method are twofold:

1) The RLS filter minimises an error based on the data itself whereas the LMS filter uses merely a statistical error.
2) The RLS filter minimises the error at every point in time as opposed to the LMS filter, which only minimises the error after infinite time.

The result of these differences is that the noise rejection is improved in some cases by up to 10 dB.

2.4 BACKGROUND TO THE RLS ALGORITHM

Originally the RLS method was limited to non-real-time applications because of the large number of calculations needed. This was in the order of N^3, where N is the length of the filter. "In the order of N^3" will subsequently be written $O(N^3)$. Since N can be as large as 256 points in some applications, obviously real-time use of RLS filters was impractical. The first people to bring the number of calculations down to O(8N) were Ljung and Falconer in October 1978 [3]. They were followed by Carayannis et al. in December 1983 [4], who reduced the number of calculations to O(5N) for linear prediction. Both these techniques were derived using only algebraic methods. But in April 1984, Cioffi and Kailath published a derivation using geometrical concepts [5]. This made the RLS algorithm simpler to understand and it is this method that will be used in its explanation.

The rest of this chapter is based on two articles, one by [6] and the other by [5], using one article to explain the other.

2.5 THE RLS ALGORITHM

2.5.1 The Error Vector and Projection Matrix

The RLS algorithm must obtain a set of filter weights, $W_{N,T}$ that approximate $W^*_{N,T}$ very closely. This is done by minimising the sum of the squared errors $\Phi_N(T)$, where

$$\Phi_N(T) = \sum_{t=0}^{T} \delta^{T-1} \{d(T)+W_{N,T}\,[y(t)\ y(t-1)\ \ldots\ y(t-N+1)]^T\}^2$$

δ is a forgetting factor to enable the filter to track

variations in $W^*_{N,T}$. It does this by attaching greater importance to recent samples than to earlier ones. For this explanation δ is set to 1, which means that it has no effect and in this case, the system is known as "prewindowed". If δ were less than 1 then the system would be known as "exponentially windowed". The sum of the squared errors, $\Phi_N(T)$, in the RLS filter is viewed as the squared length of an error vector. The length of this vector is minimised by the algorithm, thus giving the solution the the RLS problem.

To make this process efficient, a vector called a "pinning vector" is used and it will be explained later in this chapter. The calculations take place in an M-dimensional vector space of real numbers. Therefore the input and primary signals are M x 1 column vectors given respectively by:

$$y_T = [y(t)\ y(t-1)\ \ldots\ y(0)\ 0\ldots0]^T \qquad 2.2$$

$$d_T = [d(t)\ d(t-1)\ \ldots\ d(0)\ 0\ldots0]^T \qquad 2.3$$

where M >> T, the time index, N is the filter length and 'T' stands for 'transpose'. The primary signal is also referred to hereafter as "the desired response". The M x N matrix $Y_{N,T}$ is defined by:

$$Y_{N,T} = [\ y_T,\ y_{T-1},\ \ldots,\ y_{T-N+1}\ ] \qquad 2.4$$

and the error vector is defined as:

$$\epsilon_{N,T} = d(t) + Y_{N,T}W_{N,T} \qquad 2.5$$

So the squared length of the error vector $\Phi_N(T)$ can be written as:

$$\Phi_N(T) = |\epsilon_{N,T}|^2 = \epsilon_{N,T}\epsilon_{N,T} \qquad 2.6$$

The solution that minimises this error is the set of weights $W_{N,T}$ that acts on y_T, y_{T-1},..., y_{T-N+1} to form a vector that

is the minimum distance from the desired response d(T). This prediction is constrained to lie in the M-dimensional vector space R^M since it is only made up of existing inputs, which all lie in that vector space. However, d(T) is not constrained in this way, so it can lie outside R^M. From figure 2.3 it is easy to see that $\Phi_N(T)$ is minimised when the error $\epsilon_{N,T}$ is orthogonal to R^M. In figure 2.3 R^M is 2-dimensional for clarity but generally the number of dimensions is far greater. This means that $\epsilon_{N,T}$ is orthogonal to the column space of $Y_{N,T}$ and therefore:

$$Y^T_{N,T}\epsilon_{N,T} = Y^T_{N,T}\ (d_T + Y_{N,T}W^T_{N,T}) = 0 \qquad 2.7$$

The solution for $W_{N,T}$ is therefore:

$$W_{N,T} = -d_T Y_{N,T}\ (Y^T_{N,T}Y_{N,T})^{-1} \qquad 2.8$$

if $(Y^T_{N,T}Y_{N,T})^{-1}$ exists. $(Y^T_{N,T}Y_{N,T})^{-1}$ is called the autocorrelation matrix and its usual symbol is $R^{-1}_{N,T}$. If $R^{-1}_{N,T}$ does not exist then a pseudo-inverse matrix, called a "Moore-Penrose" inverse, is used. This particular inverse ensures that $W_{N,T}$ has a minimum norm [7]. Using 2.7, $\epsilon_{N,T}$ can be expressed as:

$$\epsilon_{N,T} = [I - Y_{N,T}R^{-1}_{N,T}Y^T_{N,T}]\ d_T \qquad 2.9$$

$Y_{N,T}(Y^T_{N,T}Y_{N,T})^{-1}Y^T_{N,T}$ is the projection matrix $P_{Y(N,T)}$ (Which will subsequently written as $P_{N,T}$). $P_{N,T}$ transforms any vector into its projection onto a subspace defined by $Y_{N,T}$. If $P_{N,T}$ is subtracted from the identity matrix I, then what remains is a matrix that finds the component of a vector orthogonal to a subspace, also defined by $Y_{N,T}$. The matrix is written as:

$$P'_{N,T} = I - P_{N,T} \qquad 2.10$$

To make arithmetic simpler, $K_{N,T}$ is defined as:

$$K_{N,T} = Y_{N,T}R^{-1}{}_{N,T} = Y_{N,T}(Y^{T}{}_{N,T}Y_{N,T})^{-1}$$

Using these definitions we can write:

$$W_{N,T} = -d^{T}{}_{T}K_{N,T} \qquad 2.11$$

$$\epsilon_{N,T} = P'_{N,T}d_{T} \qquad 2.12$$

This means that the error $\epsilon_{N,T}$ is the component of d_T orthogonal to the subspace that spans the N most recent input vectors y_T, y_{T-1} etc. Also, $W_{N,T}$ is the negative projection of d_T onto $Y_{N,T}$: i.e. it is a predictor for d_T.

2.5.2 The Fast Algorithm

Equations 2.8 and 2.9 could be used to solve the RLS problem directly but this would require $O(N^3)$ multiplications per iteration. Instead, a different method is used, using the pinning vector σ, mentioned earlier in this chapter, defined as:

$$\sigma = [1,\ 0...0]^{T} \qquad 2.13$$

This vector has the property of eliminating all components from the vectors it is multiplied by, except the most recent one. Three other vectors are introduced to solve the RLS problem:

$$A_{N,T} = [\ 1\ \ -y^{T}{}_{T}K_{N,T-1}\ ] \qquad 2.14$$

$$B_{N,T} = [\ -y^{T}{}_{T-N}K_{N,T}\ \ 1\ ] \qquad 2.15$$

$$C_{N,T} = -\sigma^{T}K_{N,T} \qquad 2.16$$

All these vectors are similar in form to $W_{N,T}$ (see equation. 2.11). The vector $A_{N,T}$ uses previous input values to predict the current input value. $B_{N,T}$ is a backward prediction vector and uses the input values y(T), ..., y(T-N+1) to find the value of y(T-N). $C_{N,T}$ is a negative predictor for the

between the subspaces that span { y_T ... y_{T-N+1} } and { y_{T-1} ... y_{T-N} } or it can be interpreted as the squared length of the error vector between σ and the column space of $Y_{N,T}$. Three prediction residuals are now also defined. These quantities are the results of old filters being excited by new data:

$$e^P_N(T) = A_{N,T-1}[\ y(T) \ \ldots\ y(T-N)\ ] \qquad 2.25$$

$$r^P_N(T) = B_{N,T-1}[\ y(T) \ \ldots\ y(T-N)\ ] \qquad 2.26$$

and

$$\epsilon^P_N(T) = d(T) + W_{N,T-1}[\ y(T) \ \ldots\ y(T+N-1)\ ] \qquad 2.27$$

Having defined all the variables in the algorithm, they need to be updated recursively.

2.5.3 Recursive Updating of Variables

The projection matrix P_Y projects any vector onto the subspace defined by the column space Y, defined by the signal samples y(t) already processed. P_Y takes the form:

$$P_Y = Y(Y^TY)^{-1}Y^T$$

If another sample becomes available, the column subspace changes. However, there is no difference between the old and the new subspace, except that the new subspace contains one new component, the new sample. This makes updating the projection matrix fairly straightforward. Assume that Y is a single-dimensional space (figure 2.4). Then the prediction of a vector 'a' would be $P_Y a$. When a new basis vector, Z, becomes available the new data subspace is now {Y,Z}. However, Z is normally not orthogonal to Y and so one must be created by the projection of Z orthogonal to Y. This vector is W (see figure 2.5) and is given by:

$$W = P'_Y Z$$

P'_Y is the matrix that finds the component of a vector orthogonal to the subspace Y. Since Y and W are orthogonal, the projection matrix onto {Y,Z} can be written as $P_{Y,Z} = P_Y + P_W$ (see figure 2.6). The inverse-projection matrix is written: $P'_Y = I - P_Y$.

So the projection-operator update is:

$$P_{Y,Z} = P_Y + P'_Y Z \, (Z^T P'_Y Z)^{-1} Z^T P'_Y \qquad 2.28$$

In this case $Y = Y_{N,T}$. $P_{Y,Z}$ is the projection operator for the space spanned by Y and Z. Multiplying 2.28 by U^T and V we get:

$$U^T P'_{Y,Z} V = U^T P'_Y - (U^T P'_Y Z)(Z^T P'_Y Z)^{-1}(Z^T P'_Y V) \qquad 2.29$$

Two alternative forms of 2.28 can be derived:

$$U^T K_{Y,Z} = [U^T K_Y \quad 0] + (U^T P'_Y Z)(Z^T P'_Y Z)^{-1}[-Z K_Y \quad 1] \qquad 2.30$$

and

$$U^T K_{Y,Z} = [0 \quad U^T K_Y] + (U^T P'_Y Z)(Z^T P'_Y Z]^{-1}[I \quad -Z^T K_Y] \qquad 2.31$$

U, V and Z are any M x 1 vectors and Y is a matrix of M x 1 vectors. Equations 2.28 to 2.31 are derived in [5] and are used for updating each variable in the algorithm. Substituting $U = y_T$, $Z = \sigma$ and $Y = Y_{N,T-1}$ into 2.30 we obtain:

$$y^T{}_T K_{N,T} = [y^T{}_T K_{N,T-1} \quad 0] + (y^T{}_T P'_{N,T-1}\sigma)(\sigma^T P'_{N,T-1}\sigma)^{-1}[-\sigma^T K_{N,T-1} \quad 1]$$

Substituting 2.14 and 2.16 into Table 2.1 We obtain:

$$A_{N,T-1} = A_{N,T} - e_N(T)/\tau_N(T-1) \; [0 \quad C_{N,T-1}] \qquad 2.32$$

Substituting $U = y_{T-N}$, $Z = \sigma$ and $Y = Y_{N,T}$ into 2.31 and

using the same procedure as in the previous equation it is found that:

$$B_{N,T-1} = B_{N,T} - r_N(T)/\tau_N(T)\ [C_{N,T}\ \ 0] \qquad 2.33$$

With $U = \sigma$, $Z = y_T$ and $Y = Y_{N,T}$ in eqn. 2.31:

$$C_{N+1,T} = [0\ \ C_{N,T-1}] - e_N(T)/\alpha_N(T)\ A_{N,T} \qquad 2.34$$

With $U = \sigma$, $Z = y_{T-N}$ and $Y = Y_{N,T}$ in 2.30:

$$C_{N+1,T} = [C_{N,T}\ \ 0] + C^N{}_{N+1,T}\ B_{N,T} \qquad 2.35$$

where $C^N{}_{N+1,T}$ is the 'N+1'th term of the vector $C_{N+1,T}$. Using 2.29 it can be found that, with $U = y_T$, $Z = \sigma$ and $Y = Y_{N,T-1}$:

$$\alpha_N(T) = \alpha_N(T-1) + e_N(T)\tau_N^{-1}(T-1)e_N(T) \qquad 2.36$$

and, substituting $U = y_{T-N}$, $Z = \sigma$ and $Y = Y_{N,T}$ into equation 2.29:

$$\beta_N(T) = \beta_N(T-1) + r_N(T)\tau_N^{-1}(T)r_N(T) \qquad 2.37$$

By postmultiplying 2.32 and 2.33 by $Y_{N+1}(T)$ we obtain:

$$e^P{}_N(T) = e_N(T)/\tau_N(T-1)$$

and

$$r^P{}_N(T) = r_N(T)/\tau_N(T) \qquad 2.38$$

Using $U = V = \sigma$ and $Z = y_T$ in equation 2.29 we find:

$$\tau_{N+1}(T) = \tau_N(T-1)\ e_N(T)\alpha_N^{-1}(T)e_N(T) \qquad 2.39$$

From 2.36 we obtain:

$$e_N^2(T) = [\alpha_N(T) - \alpha_N(t-1)]\tau_N(T)$$

Substituting into 2.39 we find that:

$$\tau_{N+1}(T) = \tau_N(T)\ \alpha_N(T-1)/\alpha_N(T) \qquad 2.40$$

and

$$\tau_N(T) = \tau_{N+1}(T)\ \ \beta_N(T)/\beta_N(T-1) \qquad 2.41$$

Simultaneously solving 2.33 and 2.35 we see that:

$$[C_{N,T}\ \ 0] = \{C_{N+1,T} - C^N_{N+1,T}B_{N,T+1}\}\ [1 + r^P_N(T)C^N_{N+1,T}] \qquad 2.42$$

However, in [5], the authors obtain this equation, which I suspect is a slip:

$$[C_{N,T}\ \ 0] = \{C_{N+1,T} + r^P_N(T)B_{N,T+1}\}\ [1 + r^P_N(T)C^N_{N+1,T}]$$

The filter weights can now finally be updated. They can be found by using $U = d_T$, $Y = Y_{N,T}$ and $Z = \sigma$ in equation 2.30:

$$W_{N,T} = W_{N,T-1} + \epsilon^P_N(T)C_{N,T} \qquad 2.43$$

A summary of the parameter updates can be found in Table 2.2.

2.6 REDUCING INSTABILITY IN TIME-UPDATE RLS ALGORITHM

Although the RLS algorithm has fast convergent properties, it also becomes unstable if the forgetting factor is less than unity. This is because of the time-updating procedure [8] and also because of finite precision errors which are present since the algorithm is always implemented digitally.

Ever since the discovery of the Fast RLS algorithm there have been various efforts to eliminate this instability. However, so far there have been no O(N) RLS methods that have been proved to be perfectly stable but significant improvements have been made.

In [9] the authors present a Fast RLS algorithm which is claimed to remain stable for longer than half-a-million points using 32-bit floating-point precision and a forgetting factor of 0.96: a significant improvement on the original algorithm, which becomes unstable after only 100 points under the same circumstances.

In [9] it is stated that the most significant cause of instability in the Fast RLS method is the inaccurate computation of $r^P_N(T)$ due to round-off errors. Fortunately there are two independent ways of calculating $r^P_N(T)$:

$$r^P_N(T) = -\delta\ \beta_N(T-1)\ C^{N+1}_{N+1,T} \qquad 2.44$$

and

$$r^P_N(T) = x(T-N) - B^T_{N,T-1}\ X_{N,T} \qquad 2.45$$

If both these methods are used to calculate $r^P_N(T)$ then the difference between the two results can be used as a measure of the accumulated round-off errors. This measure is then used to correct the backward prediction vector $B_{N,T}$, the forward prediction residual $e_N{}^P(T)$, the backward prediction residual $r_N{}^P(t)$ and the Kalman Gain vector $C_{N,T}$.

For further reading on reducing the instability of RLS algorithms refer to section 3.6 of this thesis.

This completes the chapter on the RLS algorithm. In chapter 3 the results of a series of tests performed on the RLS and LMS algorithms are shown.

Input definitions:

$Y_N(T) = [y(T)\ y(T-1),\ldots,y(T-N+1)]^T$

$p_k(t) = y^T{}_{T-k}y_T$

Variable	Definition	TF Computation
filtered errors		
$e_N(T)$	$y^T{}_T P'_{N,T-1}\sigma$	$A_{N,T}Y_{N+1}(T)$
$r_N(T)$	$y^T{}_{T-N}P'_{N,T}\sigma$	$B_{N,T}Y_{N+1}(T)$
$\tau_N(T)$	$\sigma^T P'_{N,T}\sigma$	$1+C_{N,T}Y_N(T)$
$\epsilon_N(T)$	$d^T{}_T P'_{N,T}\sigma$	$d(T)+W_{N,T}Y_N(T)$
predicted errors		
$e^P{}_N(T)$	-	$A_{N,T-1}Y_{N+1}(T)$
$r^P{}_N(T)$	-	$B_{N,T-1}Y_{N+1}(T)$
$\epsilon^P{}_N(T)$	-	$d(T)+W_{N,T-1}Y_N(T)$
residual powers		
$\alpha_N(T)$	$y^T{}_T P'_{N,T-1}y_T$	$A_{N,T}[p_0(T),\ldots,p_N(T)]^T$
$\beta_N(T)$	$y^T{}_{T-N}P'_{N,T}y_{T-N}$	$B_{N,T}[p_N(T),\ldots,p_0(T-N)]^T$

Table 2.1: Transversal Filter Computation of RLS Variables

Computation	Complexity
$e^{P}_{N}(T) = A_{N,T-1}Y_{N+1}(T)$	N
$e_{N}(T) = e^{P}_{N}(T)\tau_{N}(T-1)$	1
$\alpha_{N}(T) = \alpha_{N}(T-1) + e^{P}_{N}(T)e_{N}(T)$	2
$\tau_{N+1}(T) = \alpha_{N}(T-1)\tau_{N}(T-1)/\alpha_{N}(T)$	3
$C_{N+1,T} = [0 \quad C_{N,T-1}] - e^{P}_{N}(T)\alpha_{N}^{-1}(T-1)A_{N,T-1}$	N+1
$A_{N,T} = A_{N,T-1} + e_{N}(T)[0 \quad C_{N,T-1}]$	N
$r^{P}_{N}(T) = -\beta_{N}(T-1)C^{N}_{N+1,T}$	2
$\tau_{N}(T) = [1 + r^{P}_{N}(T)\tau_{N+1}(T)C^{N}_{N+1,T}]^{-1}\tau_{N+1}(T)$	2
$r_{N}(T) = r^{P}_{N}(T)\tau_{N}(T)$	1
$\beta_{N}(T) = \beta_{N}(T-1) + r^{P}_{N}(T)r_{N}(T)$	1
$[C_{N,T} \quad 0] = C_{N+1,T} - C^{N}_{N+1,T}B_{N,T-1}$	N
$B_{N,T} = B_{N,T-1} + r_{N}(T)[C_{N,T} \quad 0]$	N
Weight update	
$\epsilon^{P}_{N}(T) = d(T) + W_{N,T-1}Y_{N}(T)$	N
$\epsilon_{N}(T) = \epsilon^{P}_{N}(T)\tau_{N}(T)$	1
$W_{N,T} = W_{N,T-1} + \epsilon_{N}(T)C_{N,T}$	N
	Total 7N+14

Table 2.2: RLS algorithm found in [Cioffi] table III

pinning vector and is known as the Kalman gain vector. The four filters defined so far ($A_{N,T}$, $B_{N,T}$, $C_{N,T}$ and $W_{N,T}$) do not in general predict their respective parameters exactly. So four scalar residuals, of errors, must now be defined:

$$e_N(T) = A_{N,T}[y(T), \ldots, y(T-N)]^T = y^T{}_T P'_{N,T-1}\sigma \qquad 2.17$$

$$r_N(T) = B_{N,T}[y(T), \ldots, y(T-N)]^T = y^T{}_{T-N} P'_{N,T}\sigma \qquad 2.18$$

$$\tau_N(T) = 1 + C_{N,T}[y(T), \ldots, y(T-N+1)]^T = \sigma^T P'_{N,T}\sigma \qquad 2.19$$

and

$$\epsilon_N(T) = d(T) + W_{N,T}[y(T), \ldots, y(T-N+1)]^T = d^T{}_T P'_{N,T}\sigma \qquad 2.20$$

In these four equations the symbol ^{T} stands for "transpose". To make these equations clearer, it can be shown that, taking equation 2.17 as an example, by substituting equation 2.14 into 2.17. 2.17 can be written as:

$$\begin{aligned} & [1 \quad -y^T{}_T K_{N,T-1}] y^T{}_T \\ = \; & [y^T{}_T \quad -y^T{}_T Y_{N,T}(Y^T{}_{N,T} Y_{N,T}) y^T{}_T] \\ = \; & y_T P'_{N,T}\sigma \end{aligned}$$

The same sort of substitutions can be made for the other three equations.

These scalar residuals are simply the most recent terms in their respective prediction error vectors. The minimized squared lengths of these vectors are given by:

$$\alpha_N(T) = \min|Y_{N+1,T} A^T{}_{N,T}|^2 = y^T{}_T P'_{N,T} y_T \qquad 2.21$$

$$\beta_N(T) = \min|Y_{N+1,T} B^T{}_{N,T}|^2 = y^T{}_{T-N} P'_{N,T} y_{T-N} \qquad 2.22$$

$$\tau_N(T) = \min|\sigma + Y_{N,T} C^T{}_{N,T}|^2 = \sigma^T P'_{N,T}\sigma \qquad 2.23$$

and

$$\Phi_N(T) = \min|d_T + Y_{N,T} W^T{}_{N,T}|^2 = d^T{}_T P'_{N,T} d_T \qquad 2.24$$

These equations are true because $P'_{N,T}$ is idempotent i.e. $P'_{N,T} = P'_{N,T} P'_{N,T}$.

The quantity $\tau_N(T)$ can have two physical interpretations: it can either be seen as the squared cosine of the angle

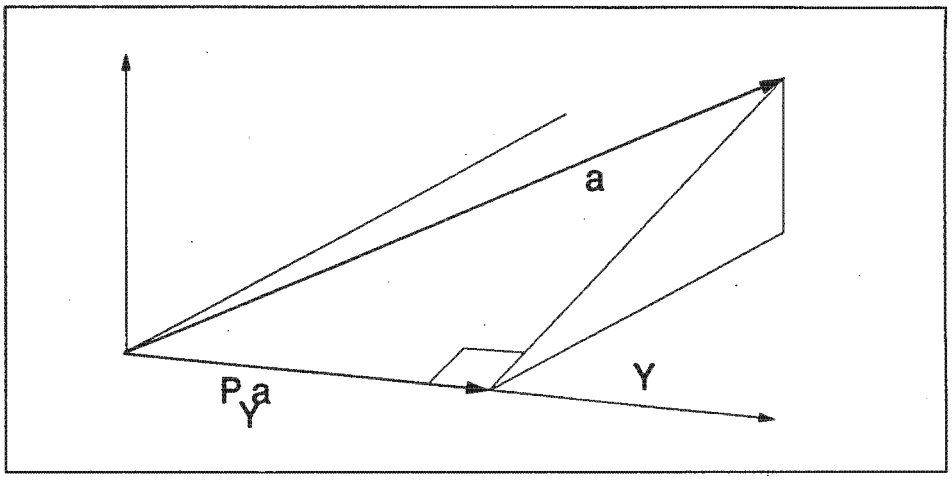


Figure 2.4 First-order Prediction of the Vector 'a'.

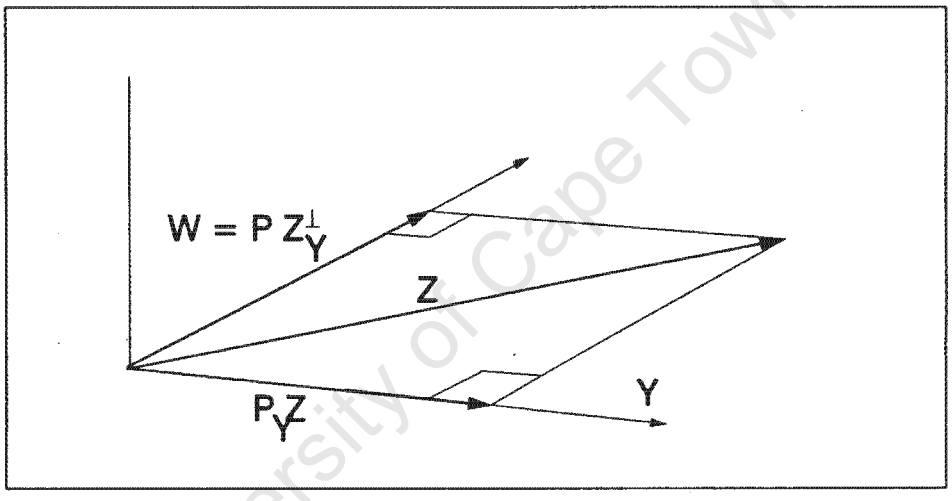


Figure 2.5 Another Basis Vector 'Z' Becomes Available

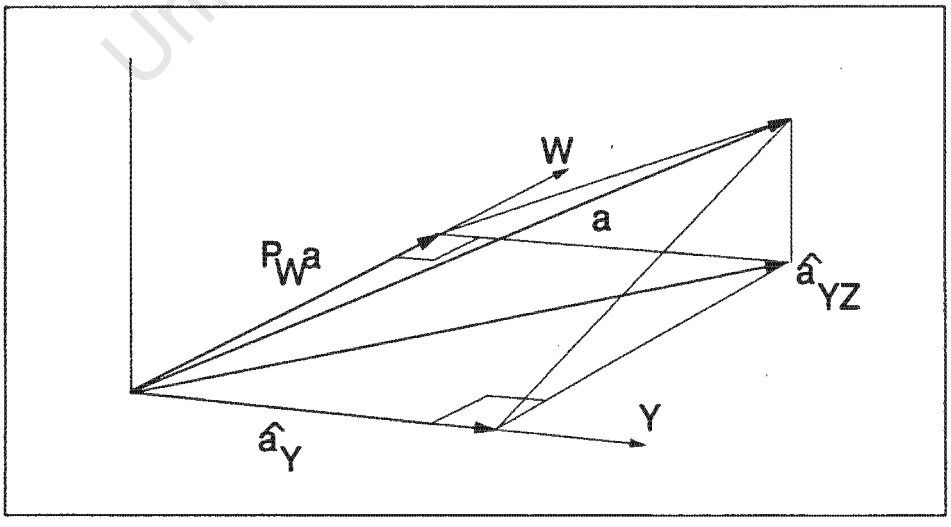


Figure 2.6 Second-order Estimate of 'a' Using The New Vector 'Z'.

CHAPTER 3

TESTING THE VARIOUS ADAPTIVE ALGORITHMS

3.1 INTRODUCTION

In the previous chapter the RLS adaptive filtering method was explained. In this chapter variations of both this and the simpler Least-Mean-Squares (LMS) algorithm are evaluated.

3.2 THE ALGORITHMS TESTED

There were six algorithms tested on a Hewlett-Packard HP9836, programmed in HP BASIC 5.0, and five tested on an IBM PS2/30, programmed in Turbo C. The algorithms tested on the HP9836 were:

1) The original Fast-RLS algorithm [4].
2) A slow $O(N^2)$ RLS algorithm obtained from a sequential-regression method found in [2].
3) A robust O(8N) RLS algorithm found in Table VII of [5].
4) The original LMS algorithm [2].
5) The SHARF method (Simplified Hyperstable Adaptive Recursive Filter), an IIR algorithm found in [10].
6) A smoothed LMS algorithm of my own derivation which is a combination of the SHARF and LMS methods, obtained simply by smoothing the system error that is used to update the adaptive-filter weights.

The algorithms tested on the IBM PS2/30 were:

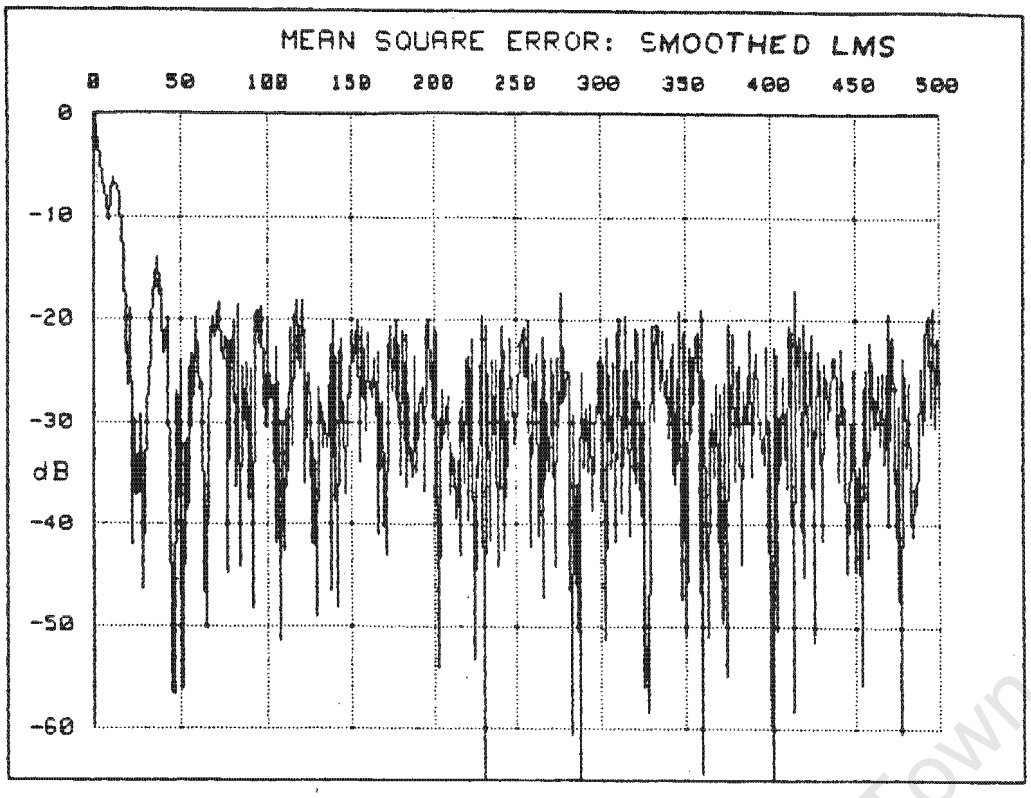


Figure 3.1 MSE Trace: Smooth-LMS. Noise Is 20% Of Signal Ampl.

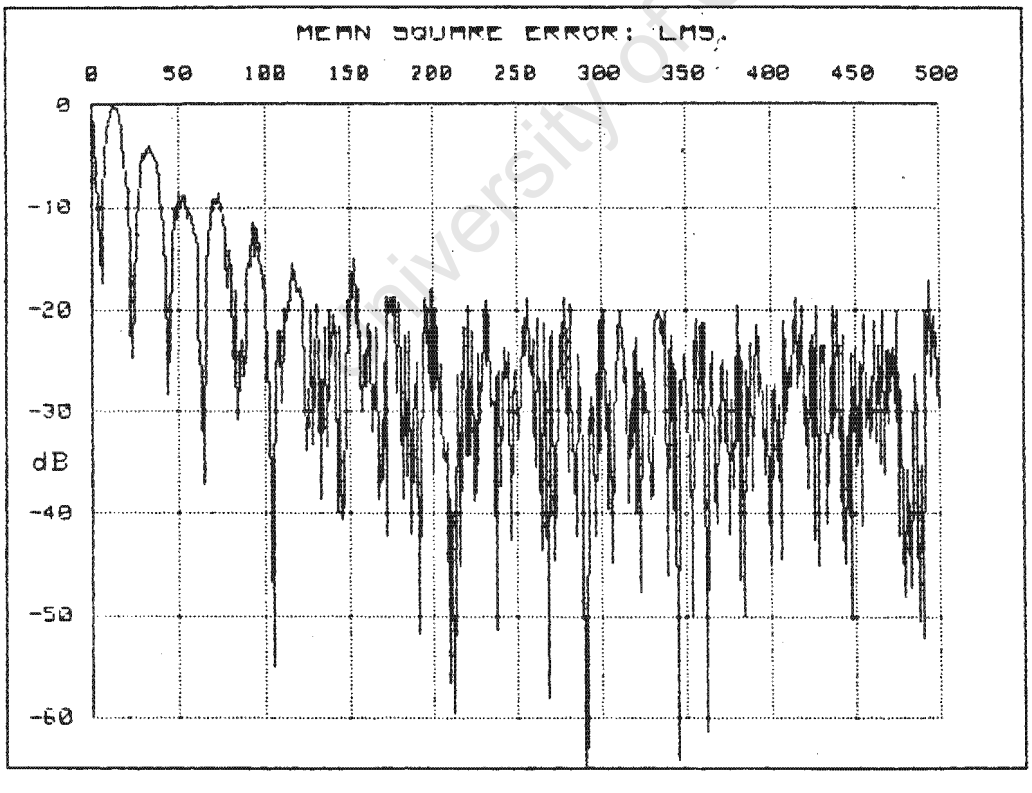


Figure 3.2 MSE Trace: LMS. Noise Is 20% Of Signal

1) The robust O(8N) RLS algorithm found in Table VII of [5],
2) The original LMS algorithm,
3) An IIR RLS method proposed in [11],
4) An even more stable Fast RLS algorithm explained in [9],
5) An order-recursive RLS algorithm derived in [8].

Each of these algorithms is presented in Appendix A and program listings of each on can be found in Appendix B.

The test-signal used on all the algorithms, both on the HP9836 and the IBM PS2/30 was comprised of the following:

1) The Reference signal was made up of a low-frequency sinusoid and random noise of half the sinusoid's amplitude. On the HP9836 the noise had a square distribution curve, while on the IBM PS2/30 the noise was Gaussian.

2) The Reference signal was passed through a filter that consisted of two forward weights that varied linearly with time, and one feedback weight that remained constant. The output of this filter was added to a high-frequency sinusoid and the resultant signal became the Desired, or Primary, signal.

The result of passing these signals through the Adaptive Filters was expected to be just the high-frequency sinusoid, since this is the only part of the Primary signal that does not correlate with the Reference signal. The results of all the tests are summarised below.

3.3 TEST RESULTS

3.3.1 Description of Figures 3.1 - 3.17

Figures 3.1 - 3.13 were derived in each case by squaring the difference between the filter error signal and the clean

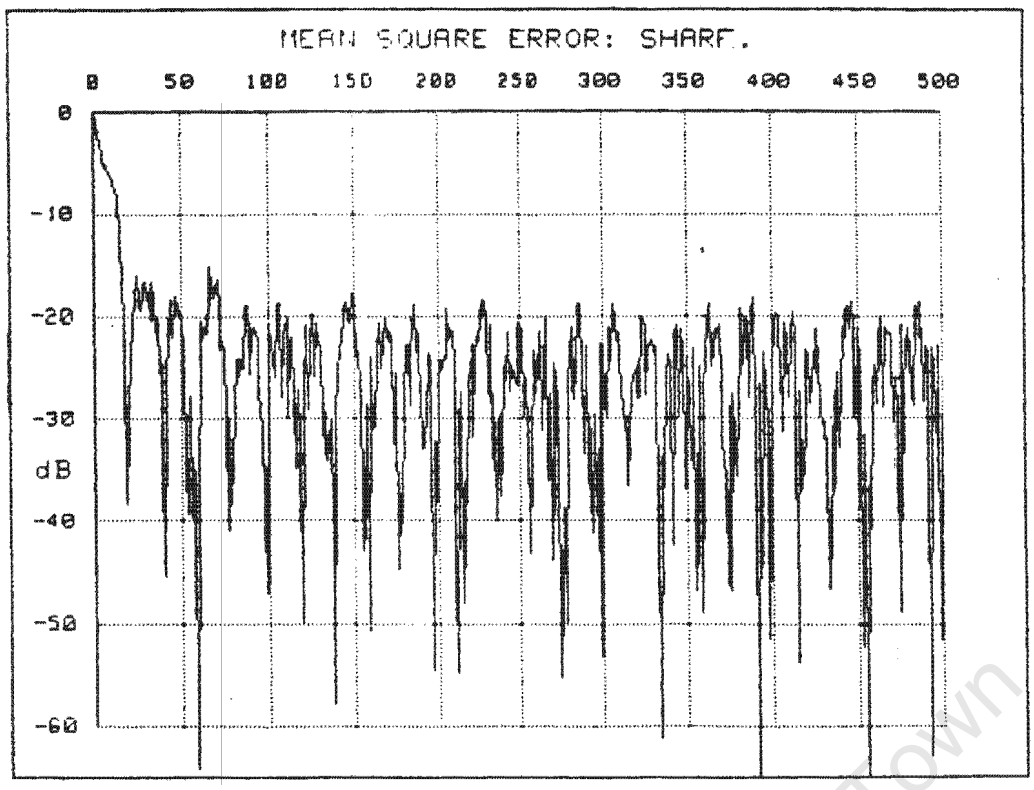


Figure 3.3 MSE Trace: SHARF. Noise Is 20% Of Signal Ampl.

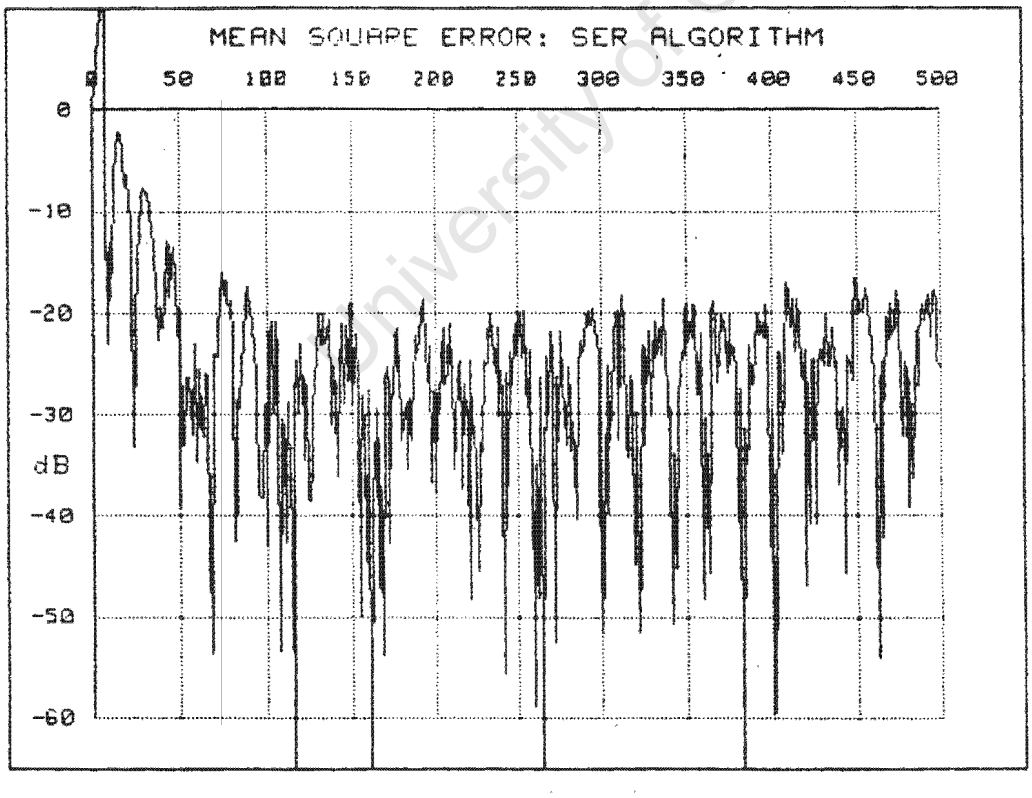


Figure 3.4 MSE Trace: $O(N^2)$ RLS. Forgetting Factor = 1.0.

signal (the high-frequency sinusoid). See figure 2.2. This was found to give a good indication of how well the adaptive filter was performing. A more detailed description now follows.

Figures 3.1 - 3.6 were created with the filter-length set to 4 and the forgetting factor, in the case of figures 3.4 - 3.6, was set to 1. The maximum amplitude of the random noise component was 20% of the clean signal's.

Figures 3.9 - 3.11 were recorded with the filter length also set at 4. However, the forgetting factor was lowered to 0.97 and the random-noise level was raised to 100% of the clean signal.

Figures 3.12 - 3.15 are time-traces of all the signals present in the adaptive-filter systems of 4 different algorithms. Figure 3.12 represents the robust RLS algorithm of [5] while figure 3.13 shows the same algorithm but with an exact initialization procedure, also found in [5]. The filter length was 4, as before, and the random noise level was 100% of the clean signal.

The same RLS algorithm was used in figures 3.16 - 3.17. These graphs were recorded to demonstrate the effect of exact initialization on stability. The filter length was set to 3 and there was only periodic interference present. The forgetting factor was set to 0.95.

3.3.2 Speed and Accuracy of Convergence

In the case where the random-noise component was low (figures 3.1 - 3.6), all the algorithms performed more or less identically. All except the LMS algorithm converged within 100 points and the final value for their mean-squared-error (MSE) graphs was -20 dB. The slight increase in the MSE in the RLS algorithms (figures 3.4 - 3.6) is due to the forgetting factor being set to unity, which means

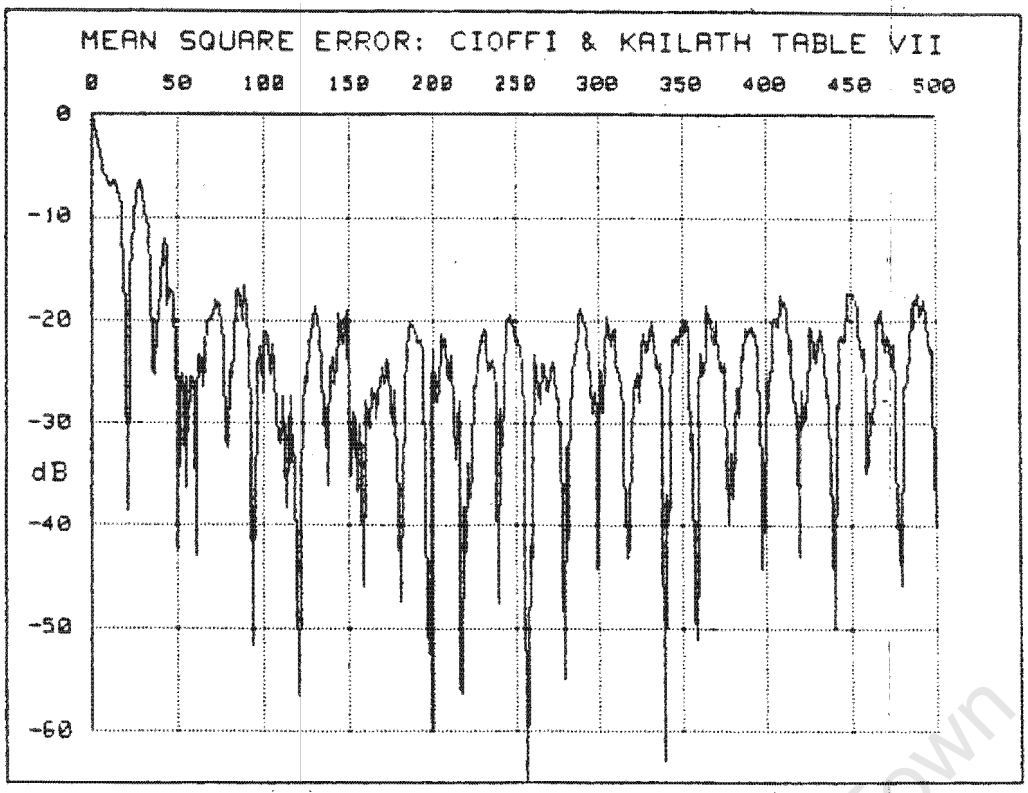


Figure 3.5 MSE Trace: Robust RLS. Forgetting Factor = 1.0.

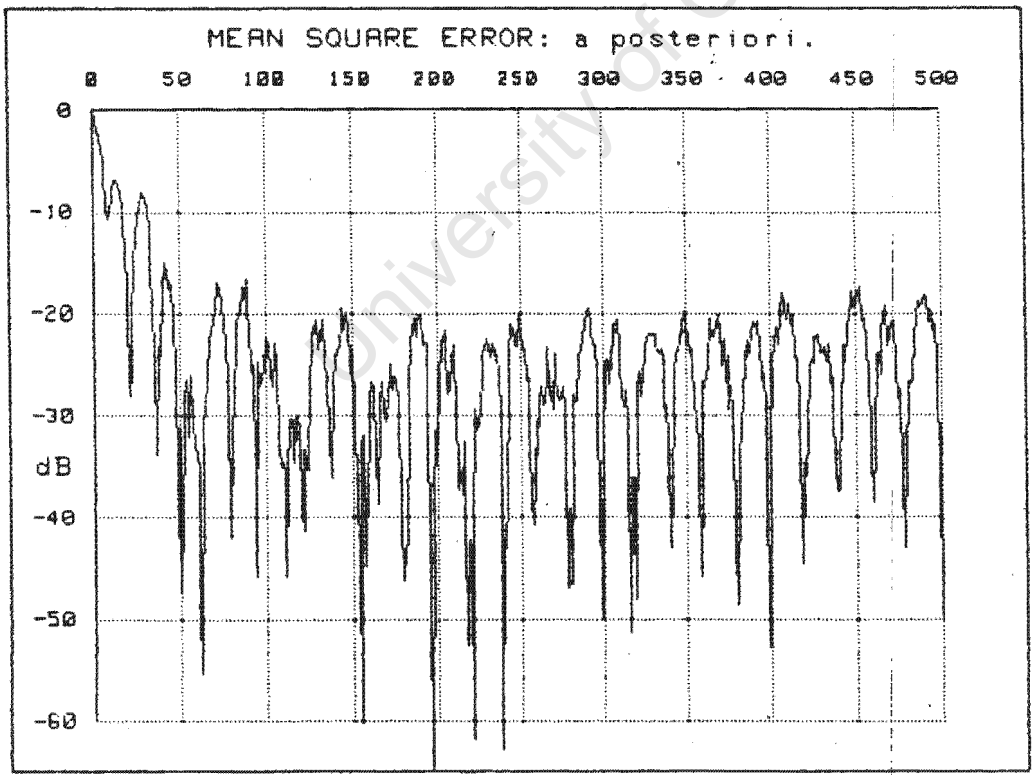


Figure 3.6 MSE Trace: Original O(N) RLS. Forgetting Factor = 1.0.

that their ability to adapt decreases as time increases. It can also be seen that the RLS graphs are smoother than those of the LMS-type algorithms'. This is because the random-noise rejection property of the RLS algorithms is superior to the LMS methods.

With the random noise component raised and the forgetting-factor lowered to 0.97 (figures 3.7 - 3.15) some significant differences can be seen compared with figures 3.1 - 3.6. All the algorithms converged within 100 points, including the LMS method. The steady-state mean-squared-error for the $O(N^2)$ RLS algorithm (figure 3.7) has increased by 2 dB, while that of the LMS method has increased by about 5 dB (figure 3.8). The signals in the LMS system can be seen in figure 3.14.

The original fast-RLS algorithm of [4] in figure 3.9 shows a best mean-squared-error of about -27 dB but after 200 points the filter becomes unstable and has to be restarted, hence the discontinuity in the graph at points 200 and 370. To detect instability, a rescue factor was devised, based on the performance of certain of the filter parameters just prior to the filter becoming unstable. This factor was derived empirically because there is no single rescue factor which functions for all RLS algorithms: the filter parameters for each algorithm behave slightly differently. The rescue factor is explained in section 3.4.

The robust algorithm of [5] in figure 3.10 has deteriorated by 12 dB compared with figure 3.5. Figure 3.12 shows the inaccuracy of the error-signal estimation: the filter error contains a fair amount of low-frequency periodic interference. In comparison figures 3.11 and 3.13, which use exactly the same algorithm but implemented with an exact initialization procedure found in [5], show a marked improvement. The mean-square-error in this case is better than -20 dB and the error signal in figure 3.13 is a very good estimate of the clean signal.

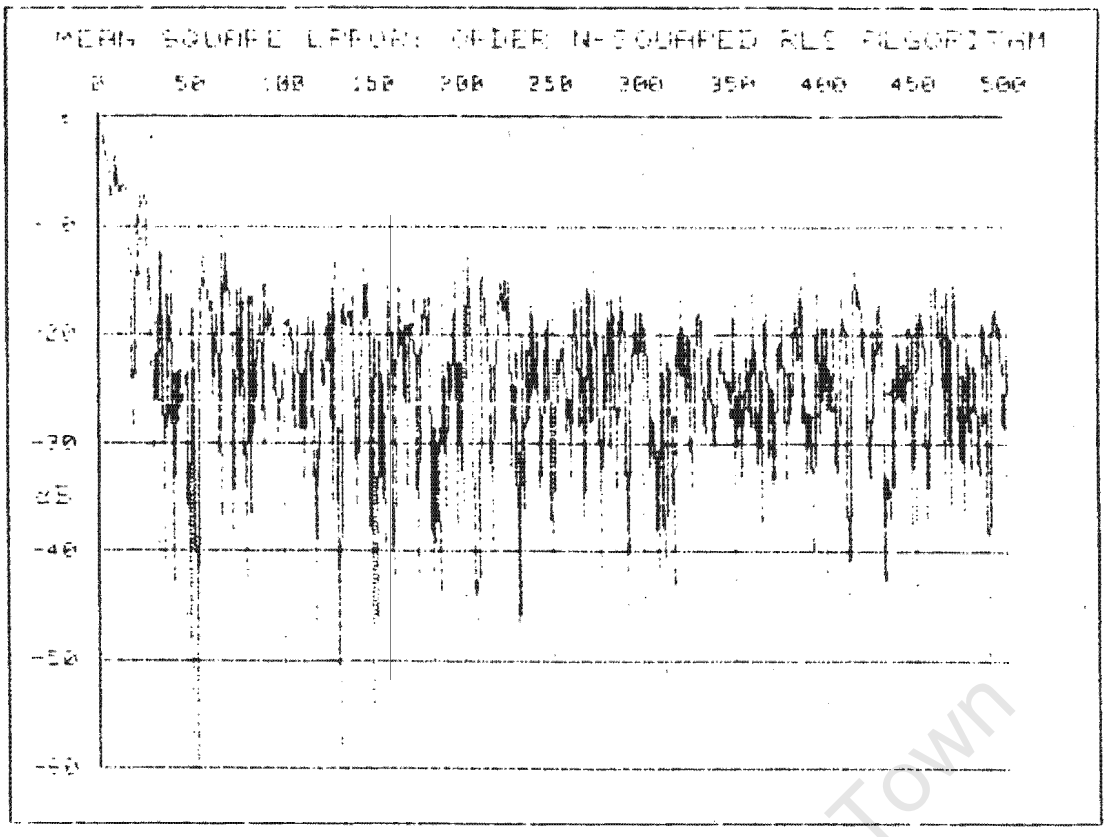


Figure 3.7 MSE Trace: $O(N^2)$ RLS. Forgetting Factor = 0.97.

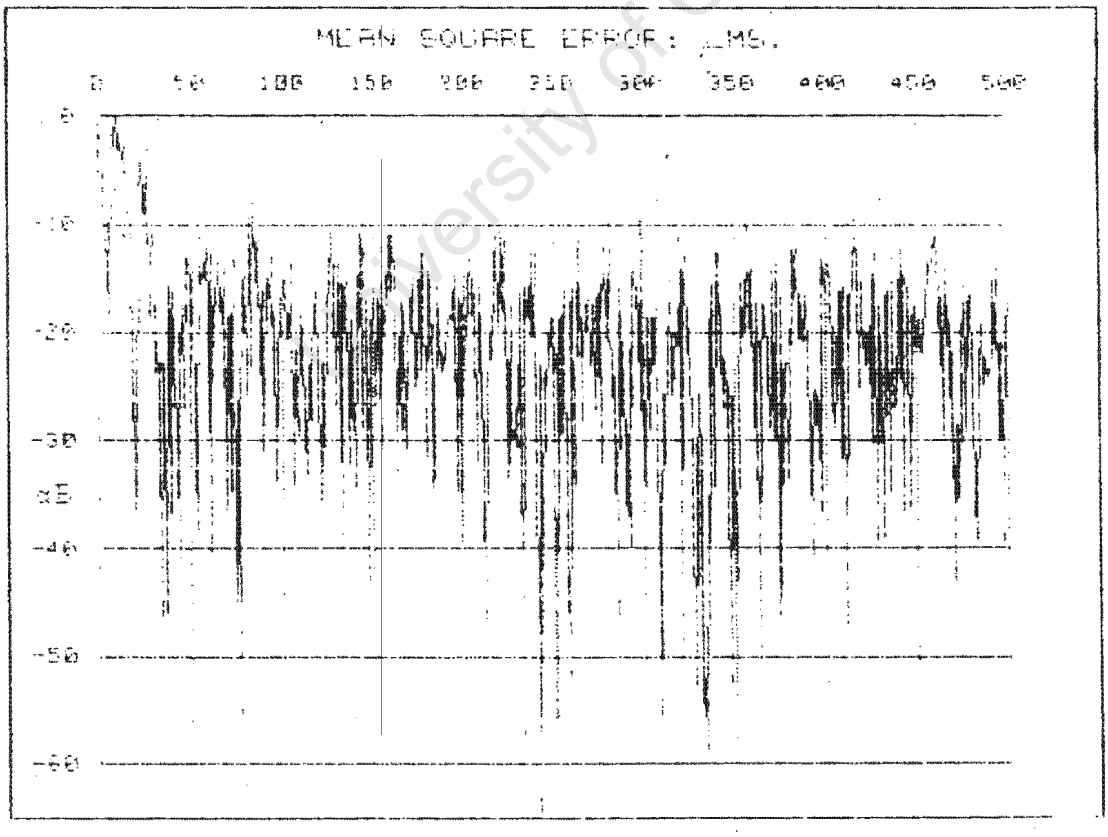


Figure 3.8 MSE Trace: LMS. Noise Is 100% Of Signal Ampl.

Figures 3.1 and 3.2 show the smooth-LMS and LMS methods' mean-square-error graphs respectively. It can be seen that, with low interference, the smooth-LMS method converges much faster than the LMS algorithm. In the case of high interference (figures 3.14 and 3.15) the smooth-LMS algorithm is less accurate than the LMS method due to the smoothing of the error signal prior to updating the filter weights. It appears that, from figure 3.15, that this smoothing causes the adaptive filter to act as a low-pass filter. Therefore, when the filter output is subtracted from the Primary signal, a significant amount of high-frequency interference will remain.

3.3.3 Stability

Theoretically, none of the algorithms tested is perfectly stable. The LMS-type methods can be made to be unstable by choosing the convergence factor, μ, to be too large and the RLS methods will be unstable if the forgetting factor is not equal to unity. The instability problem of the LMS methods can be solved by careful choice of μ. However, we can only improve the stability of the RLS algorithms to a finite degree if it is desired that the filter must be able to remain adaptable. This improvement can be achieved by using the exact initialization procedure found in [5].

A comparison of figures 3.16 and 3.17 shows the great improvement that can be made in the stability of the robust RLS method [5]. In a separate test on the HP9836 with the forgetting factor set to 0.98 the algorithm was still stable after 5000 points. However, when the same test was repeated on an IBM PS2/30 the filter became unstable after only 1000 points. This is because the HP9836 is a 32-bit machine whereas the IBM uses only 16-bit arithmetic.

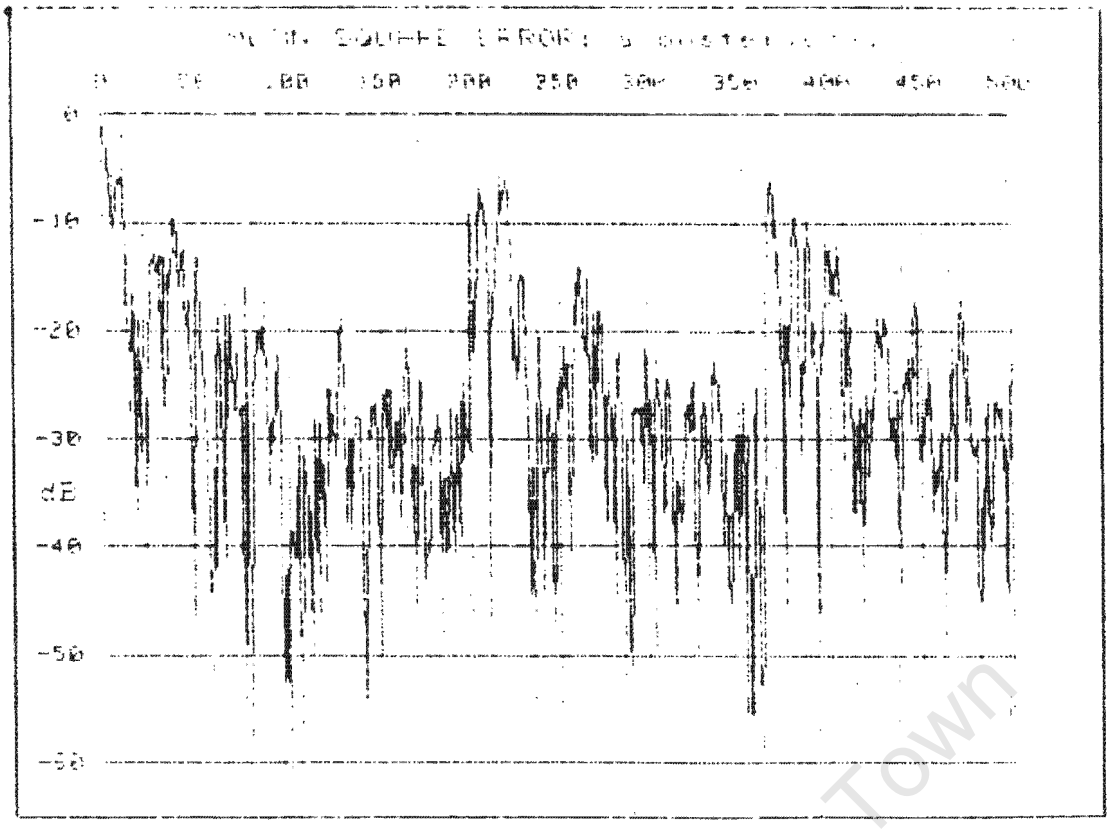


Figure 3.9 MSE Trace: Original O(N) RLS. Forgetting Factor = 0.97.

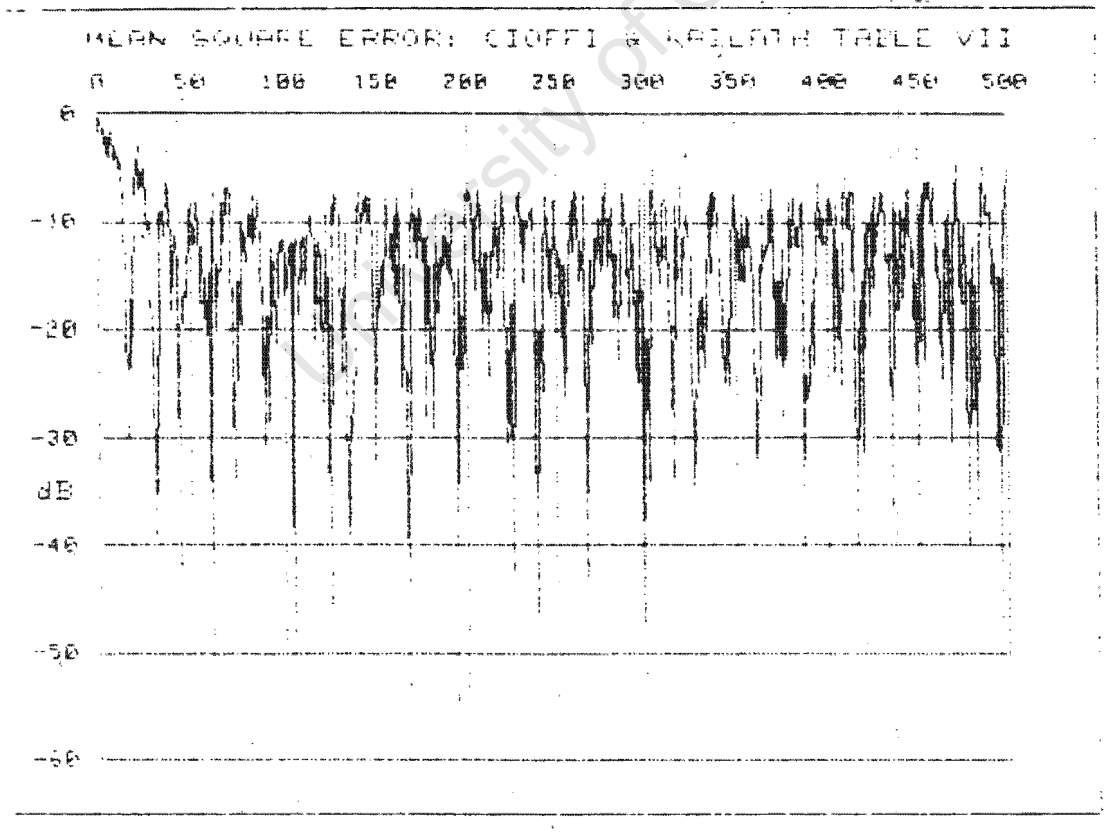


Figure 3.10 MSE Trace: Robust RLS. Forgetting Factor = 0.97.

3.4 THE RESCUE FACTOR

The algorithm of [5] (figure 3.10) is stable for 500 points with the forgetting factor equal to 0.98, albeit with bad adaptation. When this is compared with figure 3.16, the only component which differs is a lowered forgetting factor (0.95). This being the case, a large difference between the two figures should not be observed. However, such a difference can, in fact, be seen. The reason for this is that the rescue factor was different in each case. This factor was designed to become negative immediately before the system became unstable and thus to restart the adaptive filter. For figure 3.10 the rescue factor was:

$$1 - [c^{N}_{N+1,T} * e^{P}_{N}(T)]^{2} \qquad 3.1$$

whereas for figure 3.16 the factor was:

$$\beta_{N}(T) \;/\; \alpha_{N}(T) - [e^{P}_{N}(T) - e^{P}_{N}(T-1)] \qquad 3.2$$

the program being restarted if either factor becomes negative.

Each of these factors was chosen after careful analysis of all the variables present in the system just before instability occurred. It was found that the rescue factor 3.1 worked well when used in conjunction with filters that used the exact initialisation program [5]. However, when used with filters that lacked this facility it only triggered a restart if there was a significant jump in the forward á priori error ($e^{P}_{N}(T)$) - hence the inaction of the rescue factor in figure 3.10. A rescue factor works well if it triggers a restart just before the system becomes unstable. It does not work if it triggers a restart unnecessarily or if it never triggers at all.

The rescue factor provided with the robust RLS algorithm [5] never triggered a restart and was discarded for this reason.

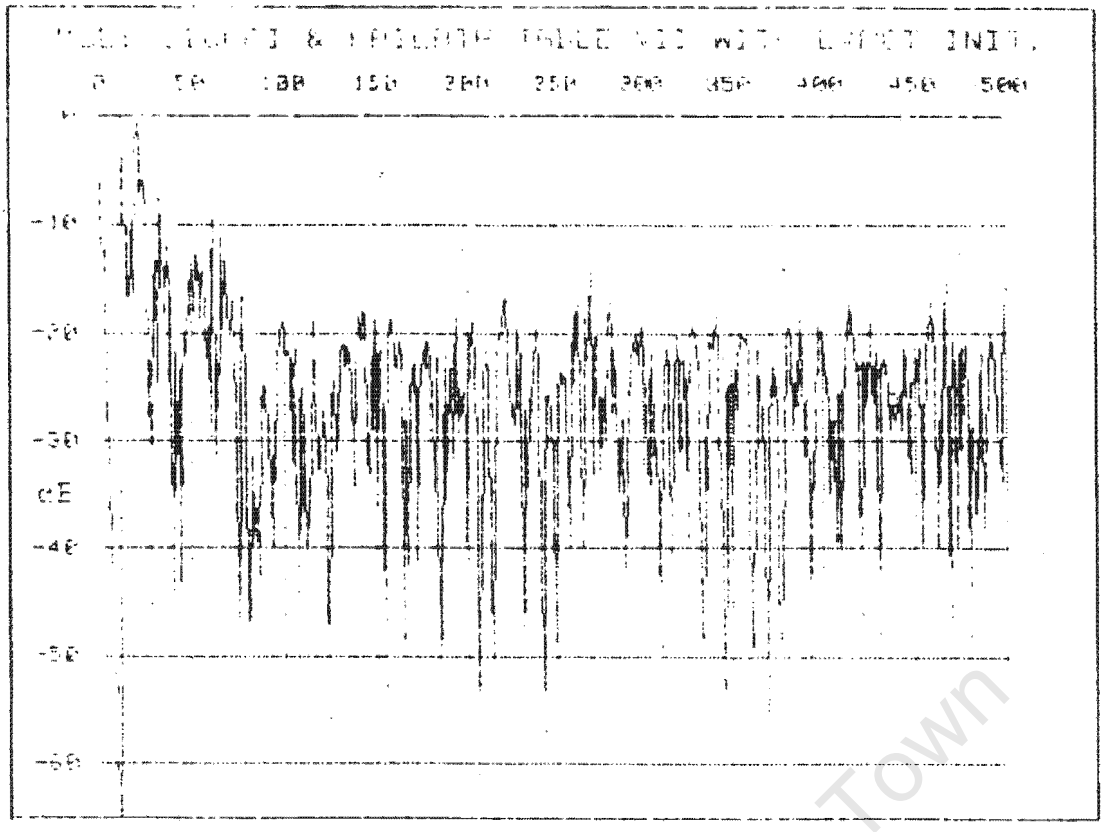


Figure 3.11 MSE Trace: Robust RLS with Exact Init. F-Factor = 0.97

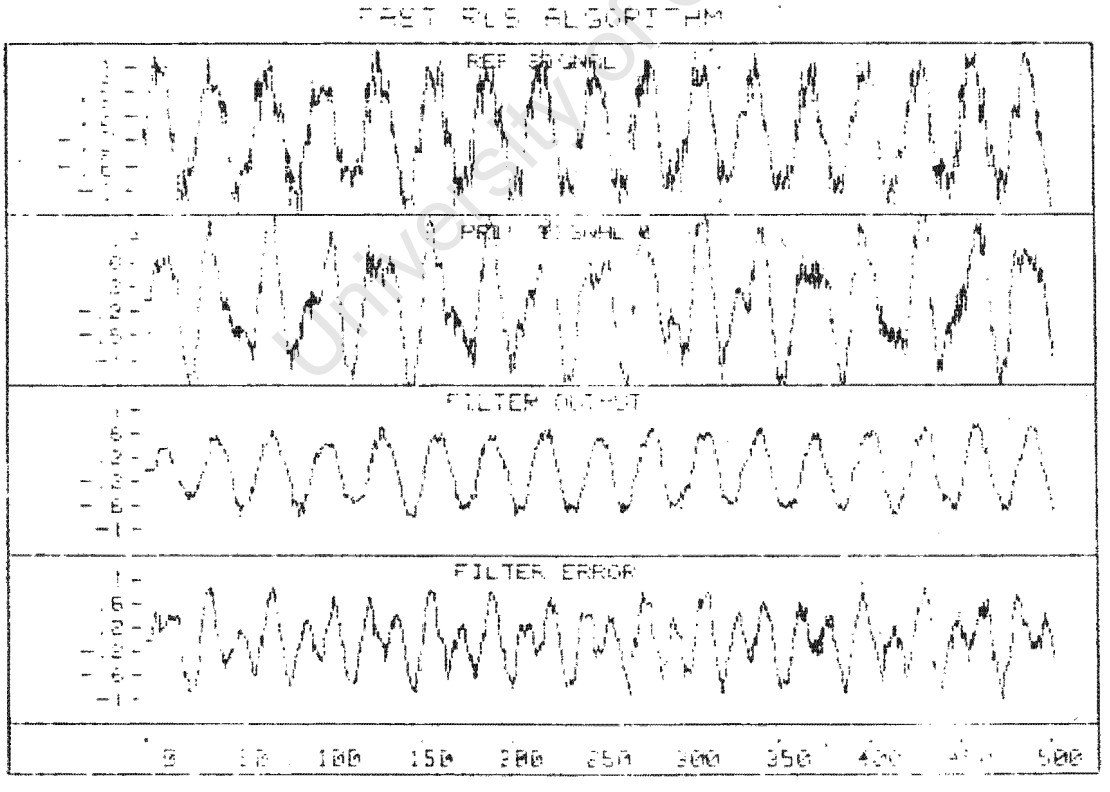


Figure 3.12 Time Trace: Robust RLS. Forgetting Factor = 0.97.

Also, [4] did not provide a rescue factor with their algorithm, presumably because it was the first paper on the subject, so it became necessary to derive the ones above. The rescue factor used in figure 3.9 was the expression 3.1.

Rescue factor 3.2 worked if there was only low-frequency interference present. However, when high-frequency noise was present it triggered the algorithms every 5 to 10 points and was therefore deemed unsuitable. Figure 3.16 was generated with this rescue factor and no high-frequency interference.

3.5 ALGORITHMS TESTED ON THE IBM PS/2

The tests performed on the PS/2 were identical to those on the HP9836. However the results obtained differed slightly because of the lower precision of the IBM machine: 16 bits compared to 32 bits in the HP9836. No graphs are shown because the printer attached to the PS/2 could not print high-resolution graphics.

In both the LMS and Robust RLS algorithms the mean-square error was 2 dB greater than on the HP9836 because of the lower precision of the PS/2. The robust RLS algorithm became unstable after 1000 points with the forgetting factor set to 0.98, compared to the same test on the HP9836 where the system was still stable after 5000 points. The rescue factor used was equation 3.1, which worked satisfactorily.

The tests with the IIR RLS algorithm [11] showed no improvement over the robust RLS method regarding convergence speed or accuracy, the number of filter weights being set to 3 forward and 2 backward. This algorithm was proposed for system modelling where poles in the model's transfer function allow for a more compact model than an all-zero one. The drawbacks of this algorithm are its complexity and the difficulty of writing a suitable exact-initialisation procedure, which result in reduced speed of execution and instability, respectively.

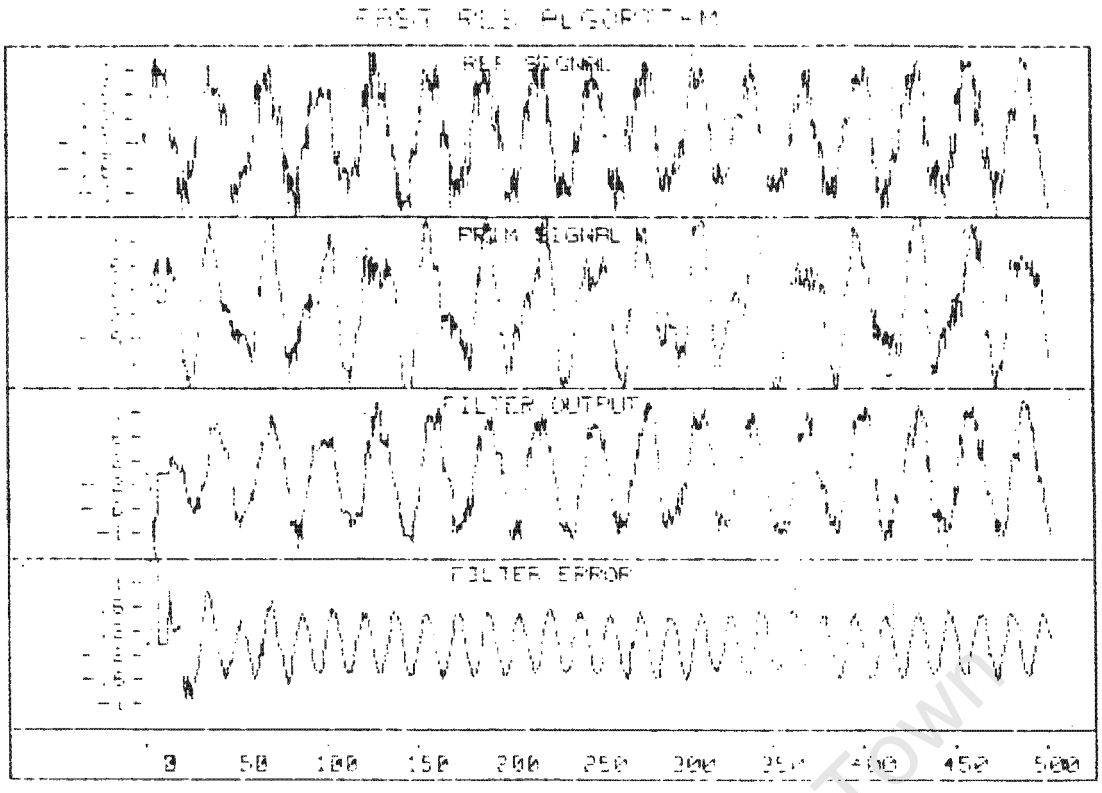


Figure 3.13 Time Trace: Robust RLS With Exact Init. F-Factor = 0.97.

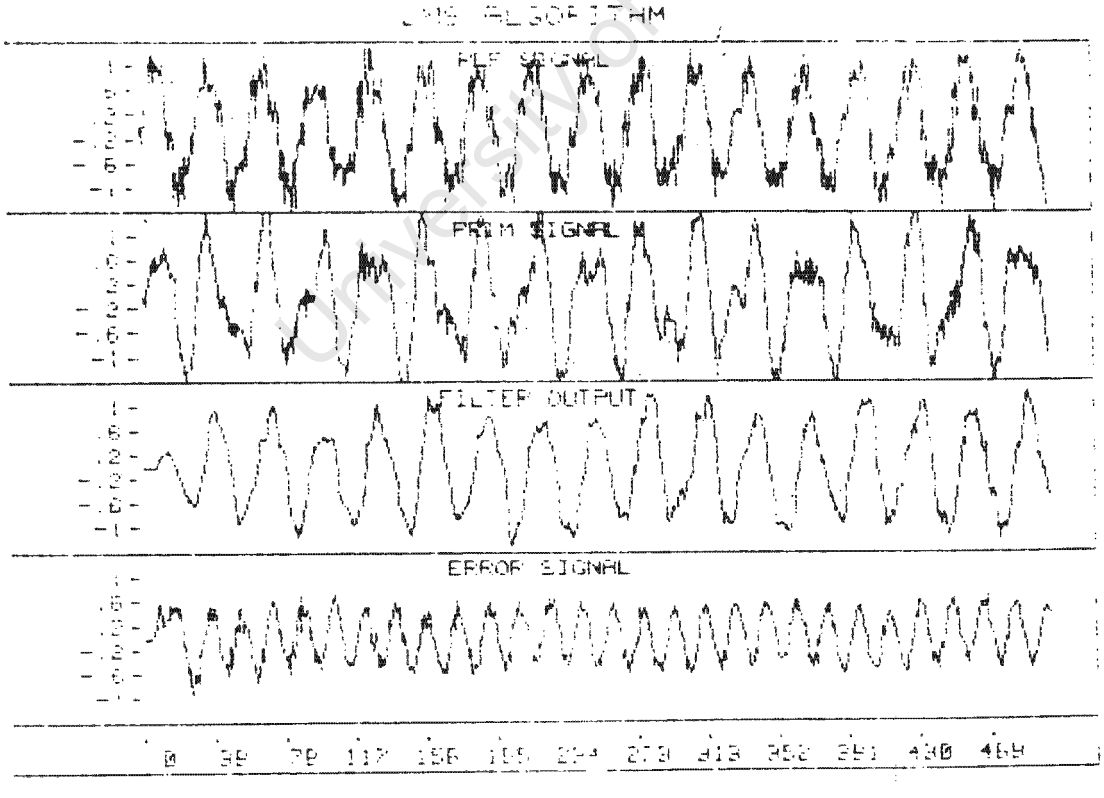


Figure 3.14 Time Trace: LMS. Noise Is 100% Of Signal Ampl.

3.6 THE LATEST DEVELOPMENT IN FAST RLS ALGORITHMS

It has been found by [8] that the instability of the RLS algorithms is attributable both to the time-updating procedure and the finite precision errors, which are present since the algorithm is always implemented digitally. Ever since the discovery of the Fast RLS algorithm there have been various efforts to eliminate this instability. So far there have been no O(N) RLS methods that have been proved to be perfectly stable but significant improvements have been made.

In [9] the authors present a Fast RLS algorithm, based on that found in [5] in Table III, which is claimed to remain stable for longer than half-a-million points using 32-bit floating-point precision and a forgetting factor of 0.96. This is a significant improvement on the original algorithm, which becomes unstable after only 100 points under the same circumstances.

In tests performed by the author, it was found that the algorithm of [9] was indeed much more stable than the ordinary Fast RLS methods. However, since only 16-bit precision was used, the algorithm became unstable after approximately 500 points with a forgetting factor of 0.96.

Other work to eliminate the instability of the Fast RLS algorithm has been done by [8]. Since it is the time-update that causes the long-term error propagation it was decided to eliminate this update entirely. This has been done by using an order-recursive filter; that is, updating the order of the filter at every time-step instead of just updating the values of the filter weights. In this way the number of cycles over which the errors can accumulate is restricted to N, the number of weights in the filter.

This kind of algorithm is known as a Mixed Transversal and Ladder Filter (MITAL). It is essentially an $O(N^2)$ method but

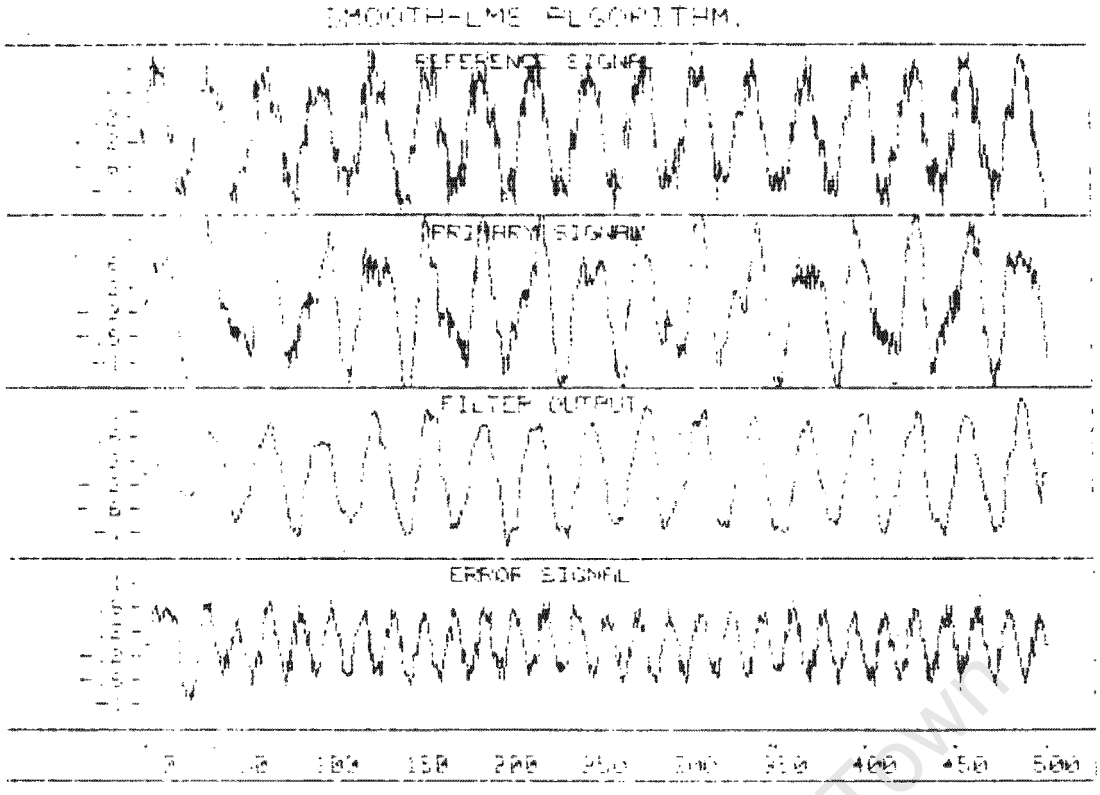


Figure 3.15 Time Trace: Smooth-LMS. Noise Is 100% Of Signal Ampl.

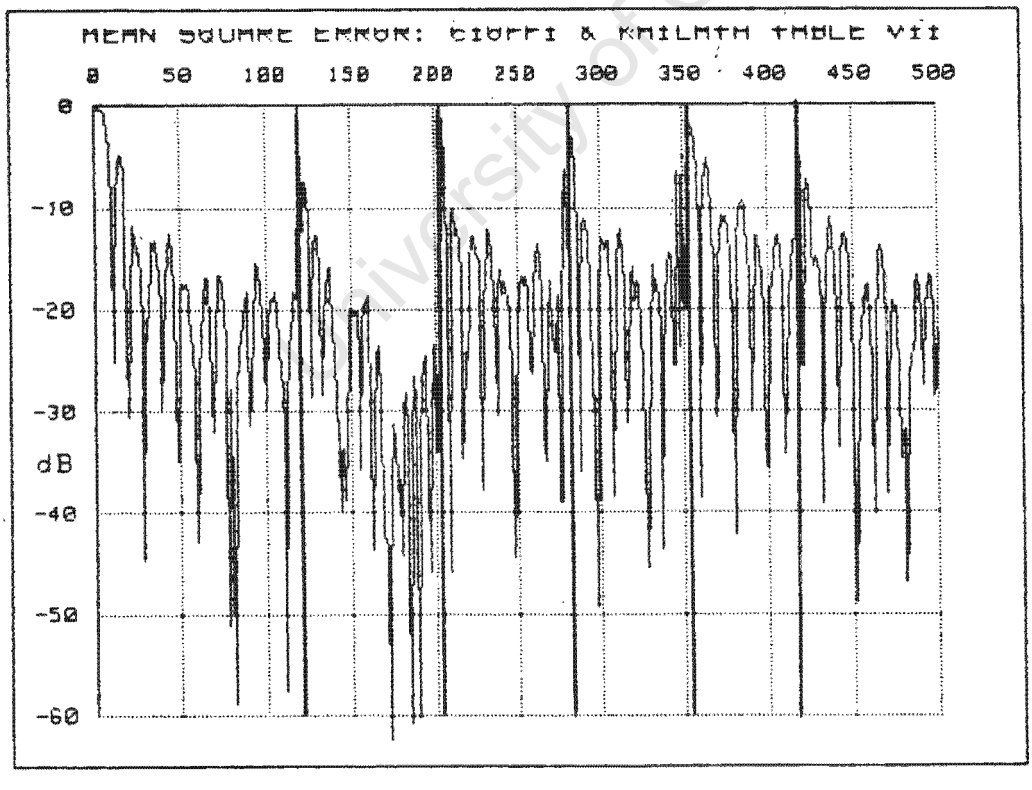


Figure 3.16 MSE Trace: Robust RLS. F-Factor = 0.95. Rescue = eqn 3.1

it can be reduced to approximately O(N) by updating the filter only when desired, rather than at every time step.

The tests performed on the MITAL algorithm show that the algorithm is totally stable in the long term. A rescue factor is not necessary since errors do not propagate for more than N cycles. However, this algorithm is only stable if the forgetting factor is less than unity, as opposed to ordinary Fast RLS algorithms, where the forgetting factor must equal unity for long-term stability. This difference is due to accumulative cross- and auto-correlation calculations in the MITAL algorithm. With the forgetting factor set to unity there would be no bounds on these values, and consequently a floating-point overflow would arise.

This algorithm is attractive for real-time use because, when the filter weights are not updated, the number of calculations needed is the same as the ordinary LMS algorithm. However, the number of weights is restricted due to the time cycle during which the weights are updated. This restriction is imposed by the amount of memory of the signal processor and/or the time needed to update the filter weights.

3.7 USING THE ALGORITHMS WITH THE TMS 320C25 SIGNAL PROCESSOR

Only two types of algorithm were used on the TMS 320C25: the LMS and smoothed-LMS algorithms. The reason for this is that these programs took only 4.5N+84 machine cycles to execute, where N is the length of the filter; whereas if the RLS-type algorithms had been used they would have executed in approximately 18N+200 cycles; about four times longer than the LMS methods.

This longer execution time would have meant that the A/D converter on the TMS 320C25 board would not have been able to operate at its maximum frequency of 109 kHz, but at a

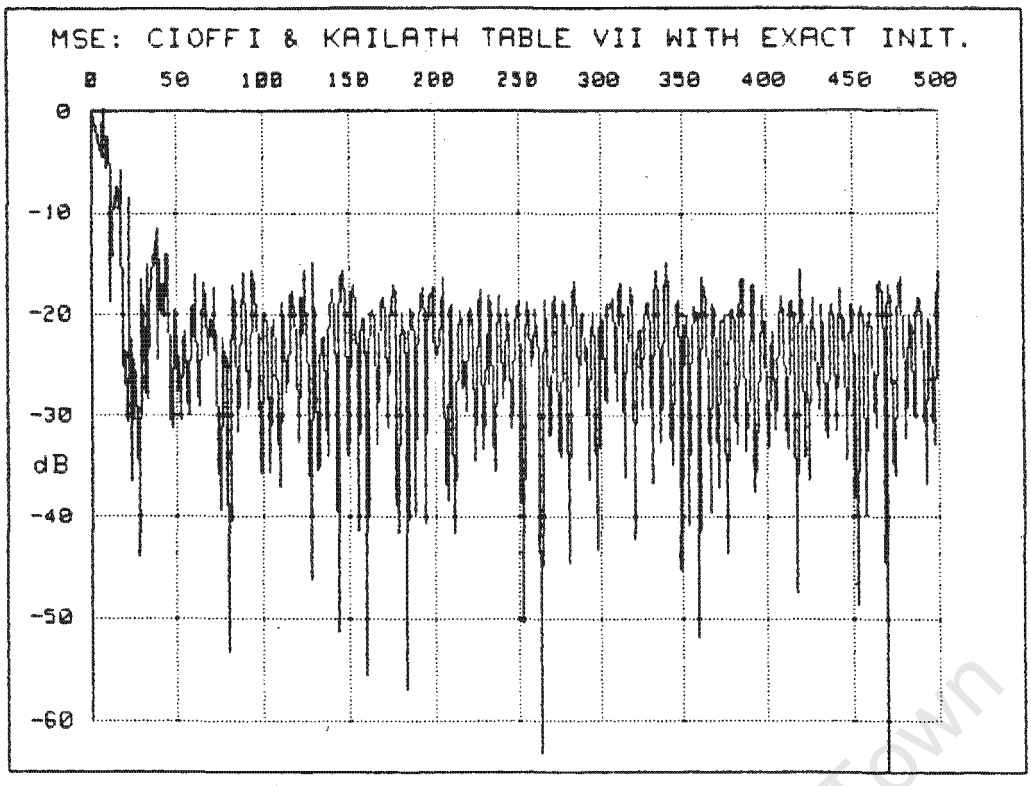


Figure 3.17 MSE Trace: Robust RLS with Exact Init. F-Factor = 0.95.

reduced rate of, at most, 20 kHz. It was for this reason, and also because the RLS algorithm would have been tedious to program and debug, that it was decided not to program the TMS 320C25 with the RLS method.

When run on the TMS 320C25, both the LMS and Smoothed-LMS algorithms showed the same results. When tested with one sinusoid for the primary and a sinusoid of a different frequency for the reference signal, the filter output was about 15 dB lower than the primary and reference signals. This corresponds to a noise-rejection of 15 dB since the ideal filter output would be zero. When the two signals were set at the same frequency and amplitude, however, the filter output was negligibly smaller than the two signals.

All the programs used for the TMS 320C25 are discussed in chapter 4.

3.8 COMPARISON OF RLS AND LMS ALGORITHMS

Table 3.1 shows in a concise manner the fundamental differences between the LMS and RLS algorithms.

LMS	RLS
* Stable.	* Unstable if sensitive to transfer-function changes.
* Fast real-time execution.	* Estimated one fifth of LMS speed when executed on the TMS board, and one half of LMS speed on a computer.
* Slow convergence.	* Rapid convergence when the primary and reference signals are highly correlated.
* Fairly good noise rejection.	* Good noise rejection.
* Simple to program.	* Complicated, especially in TMS 320C25 language.

Table 3.1

3.9 CONCLUSIONS

1) The RLS algorithms become unstable if they are able to track changes in the optimum filter's weights. The only exception to this is the order-recursive algorithm of [8].

2) Despite the RLS algorithm of [8] being superior to the time-recursive RLS algorithms as regards stability, it was decided not to use it in the TMS 320C25 chip because the number of weights is restricted. This is due to the $O(N^2)$ nature of the algorithm.

3) The Smooth-LMS algorithm was rejected in favour of the ordinary LMS method because it is less accurate, despite its faster convergence speed. The SHARF algorithm, which is an IIR version of the smooth-LMS algorithm, was rejected for the same reasons. The smoothing of the error signal to update the filter weights in the Smoothed-LMS

and SHARF algorithms has the effect of smoothing the filter output, which results in a noisier filter error (the estimate of the clean machine-element signal).

4) It is estimated that the LMS algorithm executes five times faster than the RLS algorithm when run on the TMS 320C25 chip and it is this characteristic, besides the fact that it is much simpler to program, that makes the LMS method an attractive algorithm for real-time applications.

The next chapter, Chapter 4, is a description of the condition-monitoring program and all the routines written for the TMS 320C25.

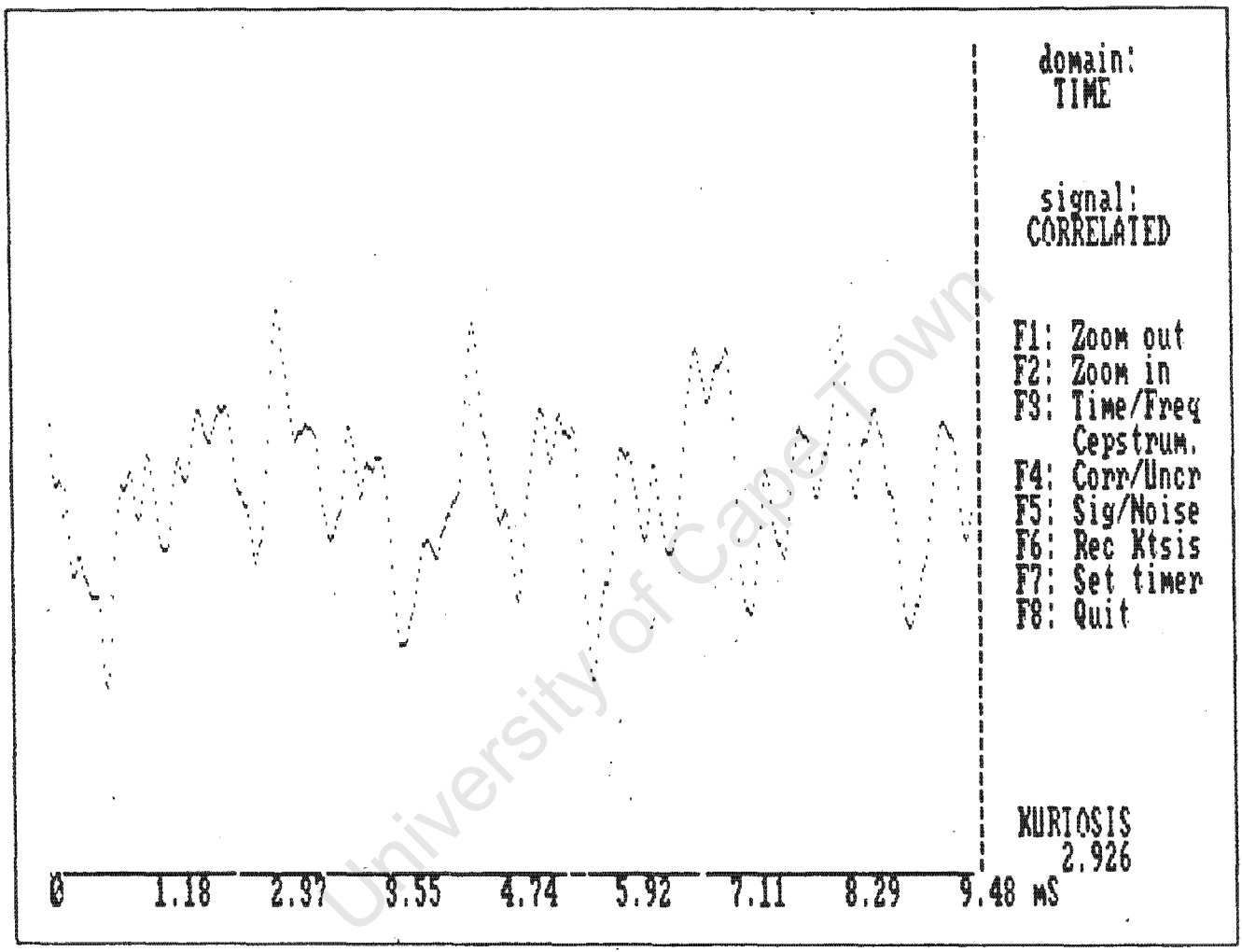


Figure 4.1 A View of the Machine-Monitoring Package Screen.

CHAPTER 4

THE CONDITION-MONITORING SYSTEM FOR THE TMS 320C25 BOARD

4.1 INTRODUCTION

The software for the system consists of two parts: the assembler-language programs and the system manager program. The assembler programs are embedded in the system manager, but they will be dealt with separately. The system hardware, namely the multiplexer referred to in this chapter, is discussed in chapter 5.

4.2 USER'S GUIDE

In order to run the condition monitor insert the disc with the programs "ADAPT.EXE" and "FIX.EXE" into the "A" drive. At the DOS A> prompt type "adapt <ENTER>". The program will prompt you for the desired maximum signal frequency to be analysed (in kHz). Enter an integer from 1 to 27. After you have done this, hit the <ENTER> key and the program will start running with the correlated signal displayed. The screen will look like figure 4.1.

The list of softkeys available is displayed on the right-hand side of the screen and the kurtosis of the signal is shown in the bottom right-hand corner. On the bottom of the screen a scale in milliseconds is displayed. This changes according to the value of the bandwidth selected.

The softkeys used in the program are:

1) F1: This halves the vertical scale of the display.

2) F2: This doubles the vertical scale of the display.

3) F3: Changes the displayed domain of the signal. The order in which the domains are displayed is: time; then frequency; then the cepstral domain.

4) F4: Switches the displayed signal between the correlated and the uncorrelated signal.

5) F5: Changes the signal displayed between the primary and the reference signal.

6) F6: Allows the user to record values of kurtosis. To record these values insert your data disc into the "A" drive after removing the system disc, and enter the name of the file required. The filename must not already be present on your disc, or else you will be prompted to enter a new name. Press any key to start the recording process. The program will record the maximum value of kurtosis after every two seconds for a duration of two minutes and will write these values to your file after the two minutes are over.

   If during the recording period the analogue signal is somehow interrupted, then recording will resume immediately the signal recommences.

7) F7: Pressing this key allows the user to change the signal bandwidth to be analysed. If a signal bandwidth of 20 kHz is needed then at the prompt, which will ask for the required bandwidth of the signal, enter "20". It is important to note that no automatic anti-aliasing filtering takes place, besides that which occurs in the multiplexer (a bandwidth of 25 kHz) which is documented in chapter 5. If lower bandwidths are required then anti-aliasing filters must be supplied by the user.

8) F8: Exit the monitoring program.

9) F10: Pause the program for examination of the screen trace or for a Screen Dump to a printer.

4.3 THE CONDITION MONITORING SYSTEM PROGRAMS

4.3.1 The System Manager Program

This program, written in Turbo-C, and henceforth referred to as 'the manager', loads and runs the TMS 320C25 programs written in Assembler. These programs are stored in the form of hexadecimal arrays in the manager. Modification to the assembler programs are performed by the manager, depending on what mode the operator chooses, by changing the code at the appropriate positions in the TMS 320C25 program memory.

This technique of changing one or two lines of program, usually branching addresses, is necessary because there is no provision for detecting whether a key has been pressed in the TMS 320C25 assembler language and hence the TMS 320C25 program can only perform one function. The program has to be physically altered for different functions to be executed.

The program displays the time domain plot, logarithmic spectrum or logarithmic cepstrum of any one of four signals:

1) The primary signal,
2) The reference signal,
3) That part of the primary signal correlated with the reference signal, or
4) The part of the primary signal uncorrelated with the reference signal.

The kurtosis of the signal is permanently displayed in the bottom right-hand corner of the screen. The program can also be paused to allow close examination of the screen trace.

A listing and flow-diagram of the manager can be found in Appendix D. However, the global variables and hexadecimal arrays that form the TMS 320C25 programs have been omitted

from Appendix D, since a listing of these items would only be tedious. They are, however, situated in a program called ADAPT.H, which can be found on the disc at the back of this dissertation.

4.3.2 Changes in the filter length

The managing program changes the length of the filter depending on what signal bandwidth is required. The reasons for this are twofold. Firstly, varying the filter length utilises fully the time available to update the filter's weights.

Secondly, each filter tap takes up a small portion of time so that the length of time that a sample of the signal is resident in the filter is proportional to the length of the filter. If the reference sensor is placed too far away from the primary sensor then the time taken for the signal to travel from the reference to the primary sensor is longer than the time taken for the reference signal to travel through the filter.

This would result in the loss of any correlation between the two signals and therefore no noise cancelling would result. Maximising the length of the filter helps to reduce this possibility. A table of allowed distances between the two sensors for different sampling rates and media is given in Table 4.1. These values were calculated as follows:

The adaptive filter program takes 4N+84 cycles to perform one iteration, which consists of filtering the signal and updating all the filter weights once. This number of cycles can be split into two portions: the part between the inputs of the reference and primary signals (N+54), and the rest (3N+30), where N is the number of weights in the filter. Since 3N+30 increases faster than N+54 when N is increased, 3N+30 was chosen as the parameter by which the number of weights is selected.

Since 3N+30 instructions must be performed during one A/D conversion, 3N+30 can be equated to the number of machine cycles per second (1×10^7) divided by the sampling rate of the A/D converter. If we call the sampling rate 'S' and the number of filter weights 'N' then we can see that:

$$3N+30 = 10^7/S$$

Therefore: $N = (10^7/S - 30)/3$

or $N \approx 3.2 \times 10^6/S - 10$ 4.1

It was decided that the noise from the reference signal should take, at most, half as long to reach the primary sensor as to propagate through the filter. This would mean that half the filter would still be effective as a noise-canceller. Therefore the delay allowed was calculated to be the number of filter weights divided by the sampling rate of the A/D. The sampling rate is four times the signal bandwidth since there are two signals being sampled.

$$\text{Delay} = N/S \qquad 4.2$$

This can be simplified to be a function only of the A/D sampling rate. Since

$$N = 3.2 \times 10^6 / S - 10 \qquad 4.3$$

A substitution into equation 4.2 yields:

$$\text{Delay} = [3.2 \times 10^6/S - 10] / S \qquad 4.4$$

Table 4.1 uses this delay along with the following speeds through different media to calculate the various distances.

Air:	330 m/s
Water:	1480 m/s
Steel:	5200 m/s

BANDWIDTH (kHz)	DISTANCE BETWEEN SENSORS (m)		
	AIR	WATER	STEEL
27	0.059	0.26	0.91
26	0.063	0.27	0.99
25	0.073	0.31	1.10
24	0.079	0.34	1.20
23	0.086	0.37	1.30
22	0.099	0.43	1.60
21	0.11	0.47	1.70
20	0.12	0.54	2.00
19	0.13	0.60	2.20
18	0.15	0.67	2.40
17	0.18	0.77	2.80
16	0.21	0.89	3.30
15	0.24	1.00	3.70
14	0.27	1.20	4.40
13	0.32	1.40	5.10
12	0.38	1.60	6.10
11	0.46	2.00	7.30
10	0.57	2.50	9.10
9	0.71	3.00	11.0
8	0.92	4.00	14.0
7	1.20	5.20	19.0
6	1.70	7.30	26.0
5	2.10	9.00	33.0
4	2.60	11.0	41.0
3	3.50	15.0	55.0
2	5.20	22.0	86.0

Table 4.1. Maximum distances allowed between sensors.

4.4 THE ASSEMBLER-LANGUAGE PROGRAMS

The assembler program is split into different modules: an adaptive filter, a kurtosis calculation program, a Fast Fourier Transform (FFT) and a program to spread 256 data points over 512 memory addresses. Listings and flow diagrams of all these programs, except the FFT program, are found in Appendix C. The FFT program was supplied with the TMS 320C25 chip. To obtain a logarithmic display of the Spectrum and Cepstrum, a program was written to generate the logarithmic look-up table found at the end of the TMS 320C25's program memory and in the file ADAPT.H.

The assembler-language programs were typed on an ordinary

ASCII editor, the same one used for TURBO C. When assembling, using the A320 assembler, it was found that there were a few differences in the programs found in [12] compared with the programs that could be compiled with A320. These were:

a) [12] uses asterisks to start comment lines as well as semi-colons. However only semi-colons are accepted by A320, the asterisks return an error.
b) It is assumed in [12] that auxiliary registers can be entered as 'ARx' (where 'x' is the number of the register) whereas A320 will accept the number of the auxiliary register only, unless the equate-line "ARx equ x" is included in the editor file.

4.4.1 The Adaptive-filtering program

This program performs adaptive noise cancelling on 1024 points, irrespective of whether time or frequency domain has been selected from the TURBO C control program "ADAPT.EXE", and calculation of kurtosis is performed all the points recorded.

The program is based on the routine found in [12] on pages 5-45 and 5-46. When this program was tested there were a few mistakes found. The first of these occurred near the bottom of page 5-45. The third-last line reads:

```
LRLK    AR3, LASTAP
```

This loads auxiliary-register 3 with the value LASTAP. It should, in fact, load Auxiliary Register 3 with a value that is one more than LASTAP, otherwise the adaptive algorithm becomes unstable.

The following errors occurred in the weight-update section of the program on page 5-46 of [12]. The program is divided into groups of three lines each, all of which should read:

```
ZALR    *, AR3
MPYA    *-, AR2
SACH    *+
```

The final group of lines before the 'RET' statement should read:

```
ZALR    *, AR2
APAC
SACH    *+
```

The program in [12] uses only one signal and is thus not an adaptive noise canceller but an adaptive predictor. Therefore it was modified to use two signals, primary and reference. This change required that an external multiplexer be built (see chapter 5) since there is only one analogue input on the TMS 320C25 board.

The multiplexer is switched using a signal generated in the program itself and fed via the analogue output on the TMS 320C25 board. It is toggled immediately after the A/D converter has sampled the signal so that it has time to settle before the next sample is taken.

4.4.2 The Kurtosis Calculation Program

Kurtosis is a measure of the peakedness of the signal and is a factor in determining the condition of a machine element. The program has been simplified on the assumption that the mean value of the signals is zero. The correct discrete calculation is:

$$k(x) = T \left[\sum_{j=1}^{T} (x_j - \bar{x})^4 \right] / \left[\sum_{j=1}^{T} (x_j - \bar{x})^2 \right]^2$$

which has been simplified to:

$$k(x) = T \left[\sum_{j=1}^{T} x_j^4 \right] / \left[\sum_{j=1}^{T} x_j^2 \right]^2$$

for the sake of simplicity. T is equal to 1024 points and 'x' is the signal currently stored. The top and bottom of

the expression are calculated in the assembler program but the actual division is performed in the system manager program for simplicity, since division in assembler is tedious.

During the calculation of each part of the expression the results were normalized to ensure the largest possible 16-bit numbers for the greatest accuracy. This means that the numbers calculated are not $(T * \Sigma x^4)$ or $(\Sigma x^2)^2$ but their ratios are correct. The flow-diagram of this program is given in figure C.5.

4.4.3 The spreading utility

When the cepstrum of the signal is taken only a 512-point FFT is needed. This means that the output consists of 256 points of information. In order to display this on the computer screen without changing the format of the display the points must be spread out over 512 places. This is done by taking each point and writing it into two consecutive places in memory. One way of doing this is to take the 256th data point and write it into the 511th and 512th memory positions, write the 255th data point to the 509th and 510th positions, and so on for all the other data points.

In practice this method would not be very efficient, since 13 instruction cycles per data point would have been needed. The actual program takes just over three cycles per point and makes full use of the pipelining characteristics of the Table-Read (TBLR) and Table-Write (TBLW) instructions. Each of these instructions take four machine cycles to complete when executed one at a time, but when performed in a block, each one takes just one machine cycle.

4.5 NOTES ON WRITING PROGRAMS FOR THE TMS 320C25

4.5.1 Writing TURBO-C Programs using the Dalanco Spry Routines

To perform all the functions listed in the Dalanco Spry linker package the library provided, called TCTS320.LIB, needs to be linked with the TURBO-C program. However, a project file cannot be created since the TURBO-C linker does not link TCTS320.LIB automatically. Also, do not attempt to create an Include File called TCTS320.H, since it is not needed.

To progress from the source code to an executable file first compile the program to an object file then exit the TURBO-C editor. Now use the command-line linker TLINK by typing the following command at the DOS-prompt when in the TURBO-C subdirectory:

```
tlink /x c0s <prog>,<prog>,<prog>,tcts320 emu maths cs
```

where <prog> is the name of your file, without the filename extension. Other libraries can be added to the above command line after 'tcts320', e.g. graphics. All the libraries that are to be linked must be in the TURBO-C subdirectory. The program is now ready to run.

4.5.2 Writing Assembler Programs For Incorporation into TURBO-C

Use any ASCII editor to write a TMS 320C25 assembler program. To prevent compiling errors, include a comment semicolon in the last line of assembler code. This is necessary because some ASCII editors append a Ctrl-Z character to the end of every file. When the program is compiled the compiler will read the Ctrl-Z as part of the program if the last statement in your program is not part of a comment. When your program has been saved, compile it using "A320.EXE", the compiler provided with the Dalanco Spry package. Compile the program as a hexadecimal file by

typing the following command at the DOS prompt:

    a320 /h<prog.ext>

where <prog.ext> is the name of the assembly-language file including the path name and filename extension. When A320.EXE has finished running it will have created a file called <prog>.hex.

At the moment the hexadecimal file consists of 512 lines with 43 characters in each line. To format the file so that it can be used in TURBO C the following operations must be performed:

1) All the rows with '00' at the end must be deleted.
2) For each remaining row, the first nine and the last two characters must be erased.
3) A comma, a space and '0x' must be inserted after every forth digit of each row.
4) '0x' must be appended to the beginning of each row and a comma must be placed at the end of each row.

After all this, each row should have the format:

0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007,

The hexadecimal numbers in the program will probably be different to the ones in this example. The file is now ready for incorporation into TURBO C. However, there is a short-cut to all this!

4.5.3 A Short Cut For Preparing Assembler Programs for TURBO-C

The separate compiling and formatting can take over a hour for programs of more than about thirty lines of hexadecimal code so a utility called "ASSEMBLE" was written to perform the assembly and formatting automatically. The flow-diagram

of this program can be found in Appendix E. To use this utility the following conditions apply:

1) If your computer does have a hard-drive then it is assumed that this is the drive from which it has been booted.The computer need not have a hard-drive, although it will be used if it does.

2) The Assembler file A320.EXE and the formatting program ASSEMBLE.EXE may be located either in the A-drive or on your hard-drive. They need not both be in the same sub-directory or on the same drive, but if they are in separate sub-directories then A320.EXE must be in a sub-directory listed in the PATH statement of your AUTOEXEC.BAT file. If you are in doubt then keep them together.

After writing and saving your TMS 320C25 assembler program, type at the correct DOS prompt:

```
assemble prog.ext
```

where "prog" is your program name, with or without its drive and pathname, and ".ext" is its extension. If you do not enter the drive or pathname then ASSEMBLE will search in the same way that it did for A320.EXE. Enter the pathname if your file is located in a subdirectory that is not in the PATH statement, or if it is located in a drive other than the "A" or "C" drive.

If there are any errors in your file, a hexadecimal file will not be created and the program will halt after displaying a list of errors in the assembler program. If, however, all is well then after a minute the program will tell you that the formatting of the file is complete and you may incorporate it in your TURBO-C program as an 'unsigned int' array.

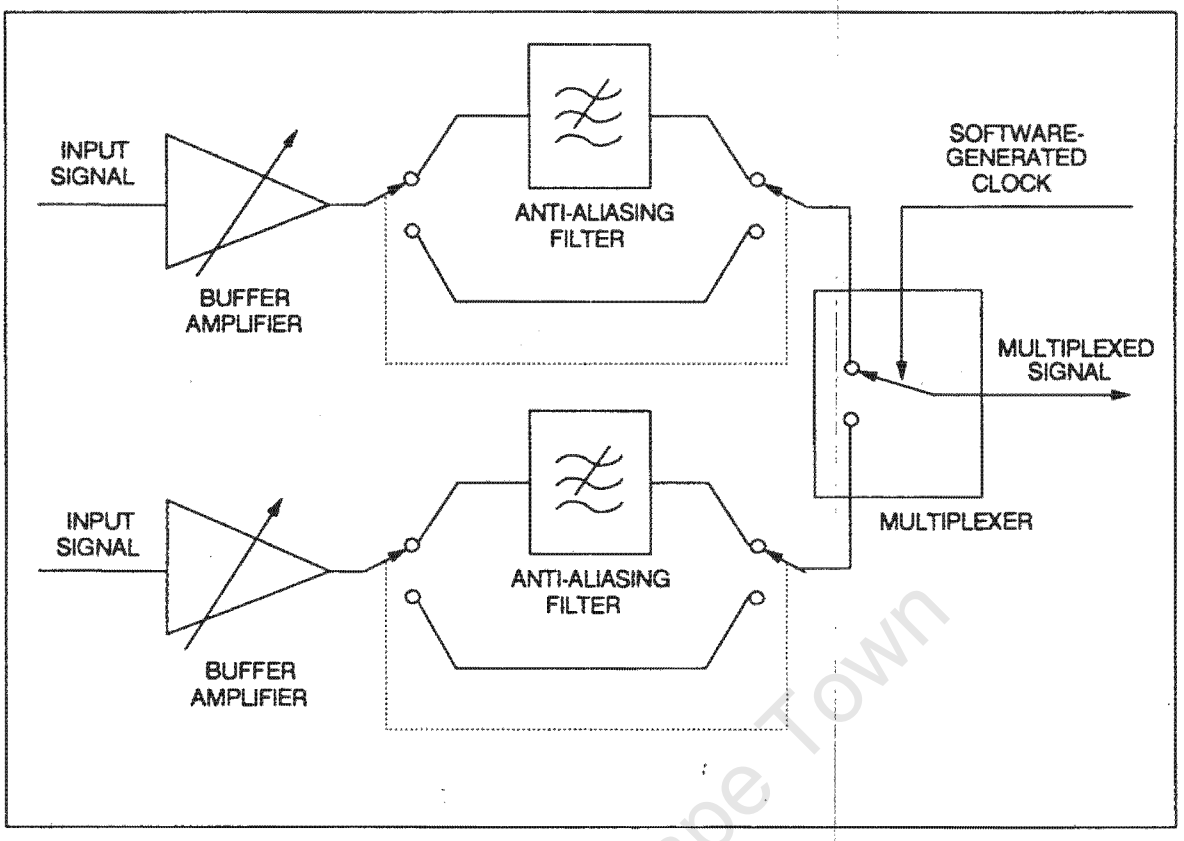


Figure 5.1 General Diagram of the Multiplexer System.

CHAPTER 5

THE MULTIPLEXER

5.1 INTRODUCTION

The TMS 320C25 board has only one analogue input. However, in order to implement the adaptive noise canceller, two analogue inputs are necessary: one each for the Primary and Reference signals. It was because of this that a multiplexer was built, in order to switch the analogue input of the TMS 320C25 to either signal. The multiplexer incorporates two other types of circuits namely buffer amplifiers and anti-aliasing filters, there being one of each for each signal. The general diagram of the system is given in figure 5.1.

5.2 THE AMPLIFIER

Amplification of the signals is necessary because the voltages from different sensors may not be large enough to use the full scale available. Also, the output impedance of the sensors may be too large for driving the anti-aliasing filters, which have an input impedance of 150 ohms so, in addition, the amplifiers act as buffers.

The amplifier circuit, shown in figure 5.2, is a simple push-pull output stage with gain variable from 0 to 11 using VR1. The maximum gain is set to be 1 + R5/R2.

R1 and the diodes D1 and D2 form a voltage limiter, the maximum potential difference allowed being 5.3V, positive or negative. The value of R1 was chosen so that the power dissipated in the diodes would not exceed 250 mW in overload conditions.

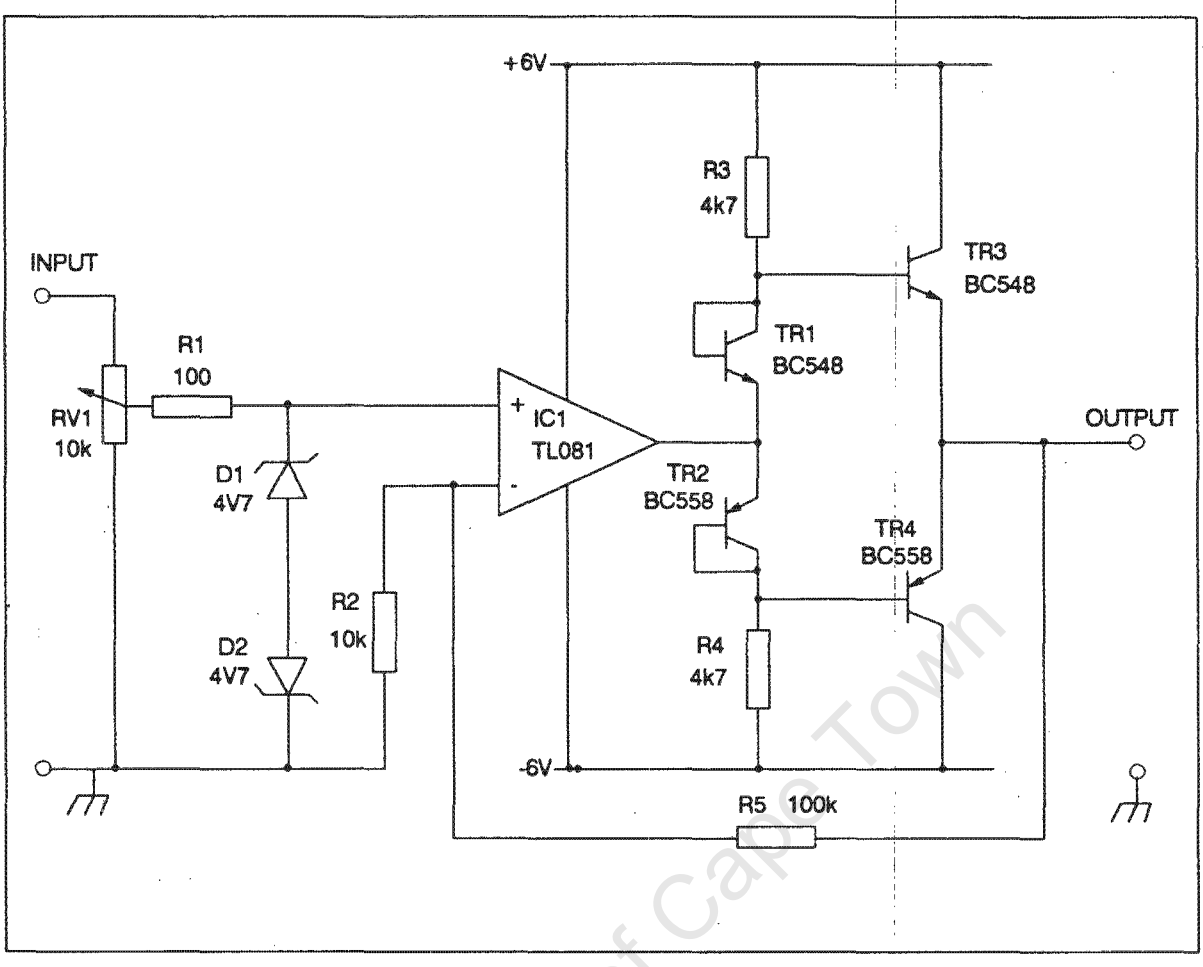


Figure 5.2 Schematic Diagram of the Amplifier Circuit

During tests it was found that the values of R3 and R4 were too large. Their high resistance means that their current sourcing ability is decreased during large voltage swings and so a compliance of only ±4.2 volts was obtained when driving a 150 Ω load, whereas 5V would have been preferred. This situation could be remedied by substituting Darlington transistors for TR3 and TR4 or decreasing the size of R3 and R4.

The correct values of R3 and R4 can be calculated in the following way. During a 4.2V positive swing:

output current i_{out} = 4.2V / 150Ω = 28mA.
current through R3 = 1.2V / 4700Ω = 0.26mA.

Therefore the current gain of the output transistors, β is:

β = 28mA / 0.26mA = 110.

During a 5V positive swing:

voltage across R3 = 0.4V
output current i_{out} = 5V / 150Ω = 33mA.

Therefore the resistance of R3 is

(0.4V / i_{out}) * β = 1320Ω.

This would give a quiescent current of 4.5mA per amplifier whereas the present quiescent current is 1.3mA per amplifier. Therefore the value of the resistors was left unchanged at 4k7 since the amplifiers are battery-powered and it was decided to keep the quiescent current low. Also a compliance of ±4.2 volts is better than 80 percent of the A/D converter's range so a change was not thought necessary.

Crossover distortion was eliminated using TR1 and TR2 in combination with IC1, which also served as a voltage

amplifier, the gain of which was controlled, as mentioned earlier in this section, by R2 and R5. The output impedance of the amplifier was calculated as follows:

No-load output voltage: 5.4 V
Voltage swing driving 150Ω: 4.2 V

So $4.2 = 5.4 * [150 / (150 + R_{out})]$
Therefore $R_{out} = 43\Omega$.

5.3 THE ANTI-ALIASING FILTERS

5.3.1 Physical Properties

The anti-aliasing filters were required to have a sharp cutoff at about 25 kHz. The reflection coefficient of the filter, the power reflected back into the load, was also required to be small. This was achieved by building a seventh-order elliptical filter found in [13] with a reflection-coefficient of eight percent and a modular angle of 60 degrees. The modular angle Θ is defined as:

$$\sin \Theta = 1 / \Omega_S$$

where Ω_S is the normalized transition bandwidth of the filter. In the case of the filters used in the multiplexer, Ω_S is 1.1547 radians.

The start of the transition band is at 23 kHz and the end is at 26.5 kHz. The theoretical characteristics of the filters are given in Appendix F.

All the components were chosen to within approximately two percent, the measurements being made on a radiometer. The terminal resistance of 150Ω has been found to cause a heavy drain on the power supply: the maximum current drawn by each filter is 28 mA. However, the advantage of having a low output impedance is that it enables polyester capacitors to be used instead of unstable ceramic ones. This is due to the

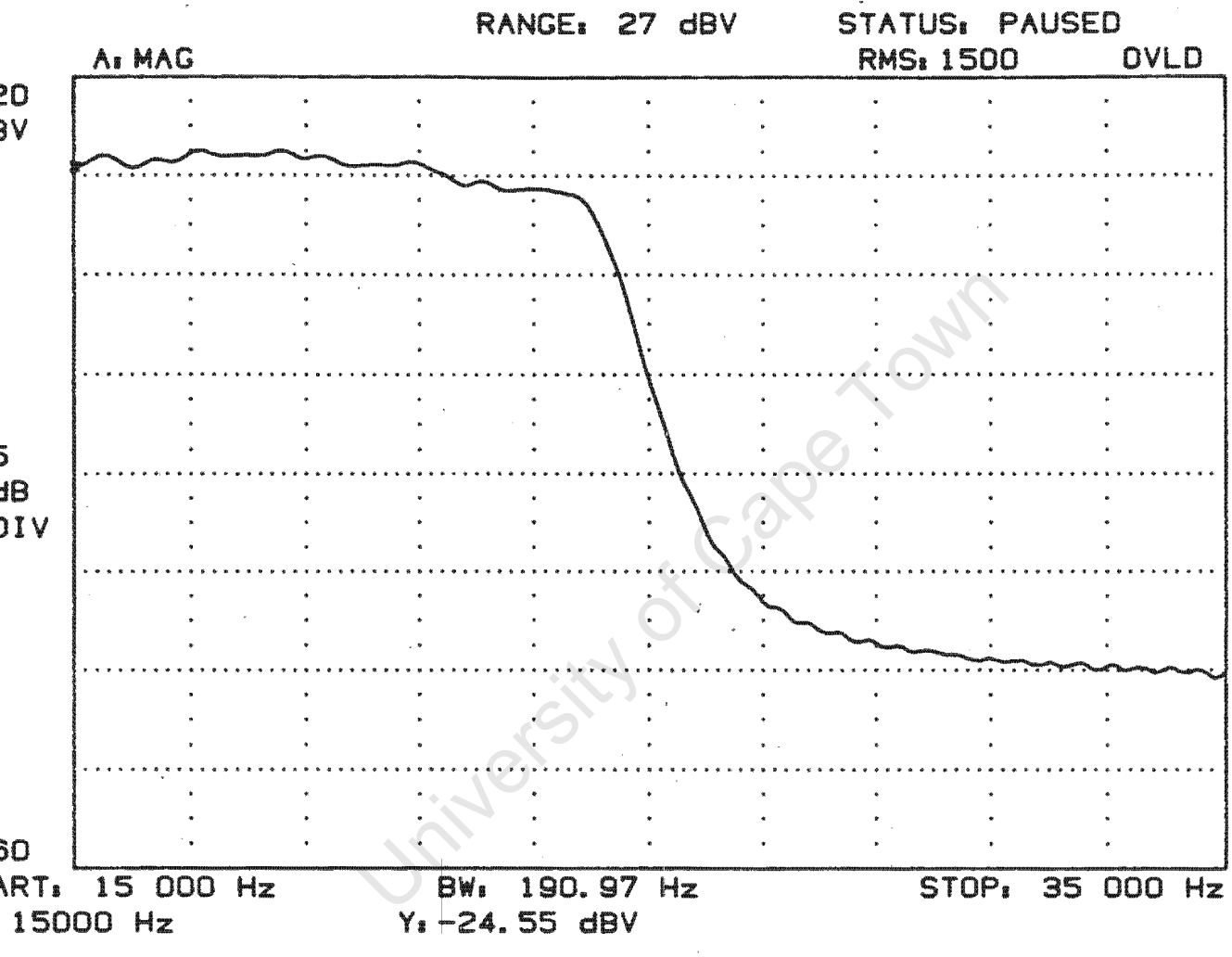


Figure 5.3 Frequency Response of the Anti-Aliasing Filters.

reasonably high capacitance required. Also, large-gauge wire can be used for the inductances since their values are fairly small and this raises the q-factor of the filter, allowing a sharper transition band.

Each inductor was wound with approximately 60 turns of 0.4 mm enamelled copper wire and the resistance of each was found to be between 0.3Ω and 0.4Ω. This correponds to an insertion loss of better than -40 dB.

5.3.2 Insertion loss

Originally, one of the advantages of having a low output impedance for each filter was thought to be that the insertion loss would decrease. However, the insertion loss of the filter does not change if the terminal resistance is changed while keeping the filter response constant.

If, for example, the terminal resistance is quadrupled then the inductances will also quadruple, which means the number of turns in the inductor will double. To fit the new windings into the same space as the old ones, the cross-sectional area of the inductor wire will have to be halved. Therefore the resistance of each inductor will quadruple, leaving the insertion loss unchanged.

The response of the filters is shown in figure 5.3. The graph was plotted from a Hewlett Packard spectrum analyser and is an average of 1500 spectra. A Bruel and Kjaer white-noise generator up to 100 kHz was used as the filter input.

5.4 THE POWER SUPPLY

The power supply was required to drive both the amplifiers and the multiplexer. The multiplexer requires a 12V source since it is a CMOS device and the high voltage produces a high speed. The amplifiers need a split supply of at least ±5 volts since this is the range of the A/D

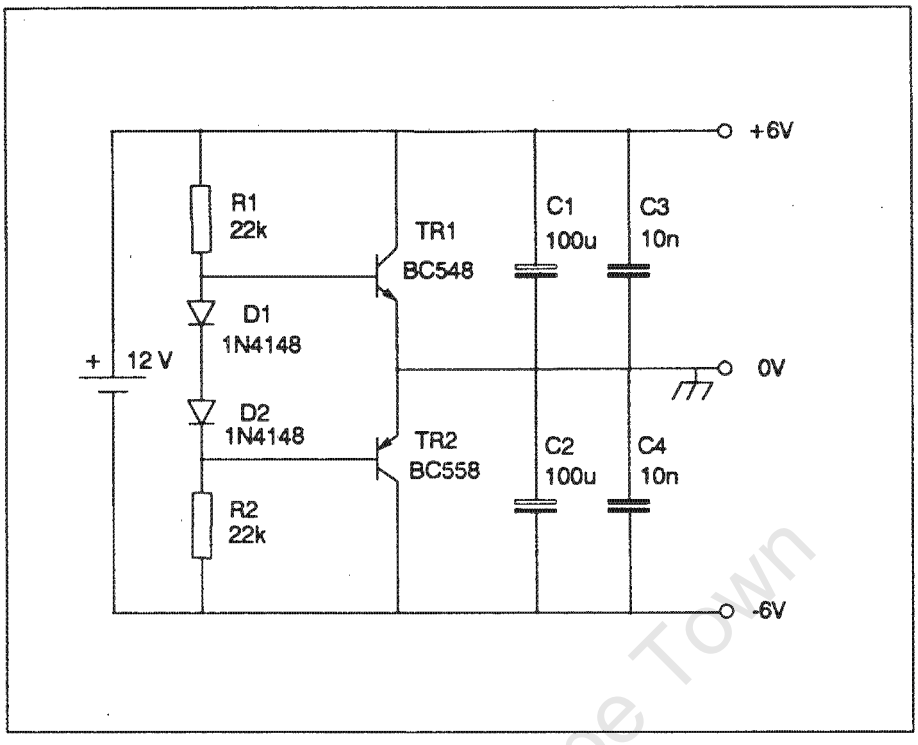


Figure 5.4 Circuit Diagram of the Multiplexer Power Supply

converter on the TMS 320C25 board. It was decided, for these reasons, that the power source should consist of ±6 volt split supply to accomodate both requirements. The circuit diagram is shown in figure 5.4.

It was found that when a constant D.C shift occurred in the input signal, the earth line of the power supply moved towards either the positive or negative power lines, depending on whether the D.C shift was positive or negative. This was due to the high current drain caused by the anti-aliasing filters.

At first, the power supply consisted only of two 100k resistors arranged as a potential divider, to create the ±6 volt split, and two 100μF capacitors for decoupling the supply. With this arrangement the earth line wandered by up to 3.5 volts each way when a large D.C. shift was applied and so the circuit in figure 5.4 was devised to reduce this. In this new circuit, with a D.C. shift of 5 volts, the earth line wanders by only one volt which is suitable for the requirements of the amplifiers. The reason for this improvement is that the present power source has a lower output impedance than the original one (220Ω versus 100k, assuming a transistor gain of 100), enabling it to supply higher currents to the earth line.

This D.C. shift is negligible in real signals, however, and the decoupling capacitors filter out any deviations in the supply for signal frequencies greater than 50 Hz.

It will be noted that there are two small capacitors, C3 and C4 of 0.01μF each placed at the output of the circuit. It is common practice to put these capacitors in parallel with electrolytic ones to stop resonance of the circuit at high freqencies. This resonance is due to the 100μF capacitors acting as if they were inductors at high frequencies.

5.5 THE MULTIPLEXER CHIP

The chip chosen to perform the task of switching the two signals was the CMOS 4051 eight-channel multiplexer. This choice was made because it has been designed to switch analogue signals with as little distortion as possible.

A two channel analogue multiplexer is not available and so the 4051 was converted into such a chip by connecting all three of its channel-selecting lines together so that the codes used for selecting the two channels were either '0' or '7' (three zeroes or three ones).

The next chapter deals with experiments performed in order to verify that Adaptive Noise Cancelling does work in Machine Monitoring.

CHAPTER 6

EXPERIMENTS AND RESULTS THEREOF

6.1 INTRODUCTION

This thesis is primarily to test whether Adaptive Noise Cancelling can reliably separate the signals from different machine elements it is when used in Machine Monitoring. Therefore it was decided to conduct experiments to verify that this was the case. The first experiment was concerned with detecting knock in an internal combustion engine and the second one was performed to determine the state of a bearing on the shaft of a machine.

6.2 DETECTING CAR ENGINE KNOCK ON THE U.C.T. ROLLING ROAD

The Energy Reseach Unit at the University of Cape Town is involved in a project to discourage the lowering of the octane number in petrol. The reason for this, they say, is that lower octane ratings cause car engines to knock at high speed, thus placing large strains on the engine. If this high-speed knock were to be detected electronically, and not just by a trained mechanic, then their arguments could be presented effectively.

It was suggested that the knock could be detected by using a sound-level meter to record the engine noise. The impulsiveness of the noise could then be obtained by calculating the kurtosis of the signal. This is a statistical measure of the signal's impulsiveness: the higher the value, the more impulsive the signal. A value of 3.0 means the signal is Gaussian, while a significantly impulsive signal would give a kurtosis value of about 6.0 and over.

This experiment was suited to the Machine Monitoring package described in this thesis and so an attempt was made to detect high-speed knock in this way.

6.2.1 Detecting Engine Knock Without Noise-Cancelling

A car was placed on the rolling road and, with the engine placed under load, the noise produced was measured using a sound-level meter placed close to the engine. The noise was fed into the Machine Monitoring package and its kurtosis was measured.

It was expected that whenever knock occurred the kurtosis of the signal would increase due to the impulsiveness of the noise created. However, there was no difference in the average kurtosis recorded either during knock or no-knock conditions. This was put down to two possible causes.

The first possibility may be that there is genuinely no change in the kurtosis of the engine noise under knock conditions, in which case this experiment's basic assumption that kurtosis should increase during knock is incorrect. This is unlikely, however, since knock produces very distinctive impulses that are audible.

The other possibility is that other noise drowned out the knocking monitored by the sound-level meter. This noise came from the engine, the engine fan, the tyres and exhaust. If this was true then some kind of noise cancelling would be needed to extract the engine signal, and this was attempted in the experiment described in section 6.2.2.

6.2.2 Detecting Engine Knock With Noise-Cancelling

Due to the failure to detect any knock without noise cancelling taking place, it was decided to use the Adaptive Noise Canceller in the Machine Monitoring package to see if any improvements would result. Two sound-level meters were

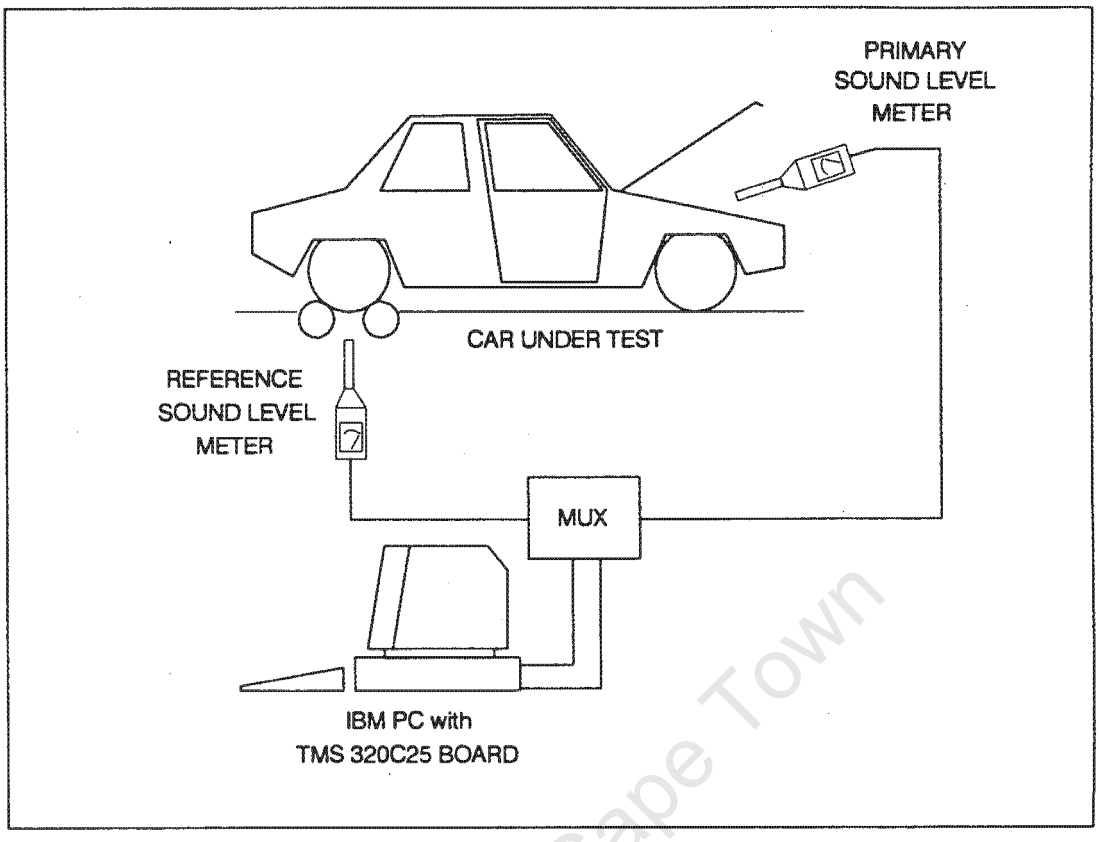


Figure 6.1 The Test Performed On The Rolling Road

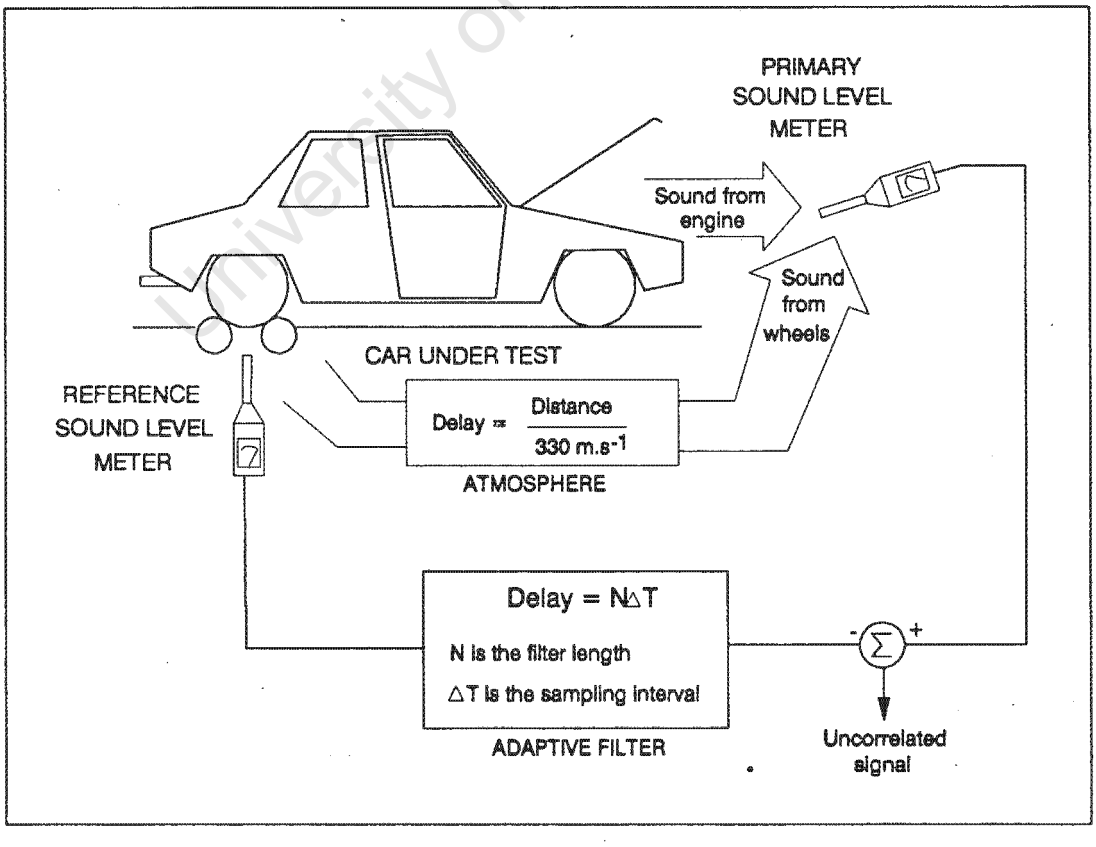


Figure 6.2 Delays in the Paths Taken by the Wheel Noise

used: one was placed near the car engine (the Primary signal) and the other was placed near the driving wheels of the car (the Reference signal), since it was thought that the tyres were the principal source of noise. The two signals were fed into the Machine Monitoring package and the kurtosis of the part of the engine noise that was uncorrelated to the wheel noise was recorded. This configuration is shown in figure 6.1.

The results, however, were the same as before noise-cancelling: there was no significant change measured in the kurtosis. This was assumed to be due to the slow speed at which sound travels in air and also due to the long paths that the sound took to reach the sound-level meters, since the rolling road is in an enclosed environment. This assumption is explained in section 6.2.3.

6.2.3 Possible Reasons for the Failure to Detect Engine Knock

In order to separate the noise of the engine from the noise of the car tyres it was necessary to place the sound-level meters at least two metres apart, one at the engine and one near the tyres. This means that the time that the sound took to travel between the two meters, or rather the time for the tyre noise to travel to the meter near the engine, was 6 mS, the speed of sound in air being 330 meters per second.

However, the time taken for the tyre noise to pass through the adaptive filter was much less, the exact time depending on the sampling rate and the number of weights in the filter. This meant that no noise cancelling took place since the algorithm could not see any correlation between the tyre noise in the filter and the tyre noise arriving from the engine sensor. The times taken for the wheel noise (the reference signal) to pass through the system is depicted in in figure 6.2.

Figure 6.2 is only a simplified version of the actual case, however. Since the entire rolling road is inside a building, the sound coming from the car wheels is deflected off the building's walls and ceiling. This echoing of the interference meant that the time taken for the noise to reach the Primary sensor is much longer than would otherwise be the case is outdoor conditions.

From this experiment it was concluded that the times taken for the signals to travel between the sensors was too long for the Adaptive Filter to be effective. The remedies for this are either to use a signal processor that is faster than the TMS 320C25 to allow an increase in the filter length, or to limit the bandwidth of the signals to be analysed.

6.2.4 Changes Made To the Noise Cancelling Package

The main problem in the effort to detect engine knock was the length of time that the tyre signal remained in the adaptive filter. To make this as long as possible, up to the limits imposed by the speed of the TMS320C25, the filter length was made to change according to how fast the sampling rate was set. For a full explanation, refer to section 4.3.2 of this thesis.

External low-pass filters have to be added if the sampling rate is less than 25 kHz, the cutoff frequency of the filters present in the multiplexer. A table of maximum distances between sensors for different sampling rates and three different media (air, water and steel) is shown in Table 4.1. No experiments have been done to test these distances because of time constraints.

The other change in the program which came as a result of this experiment was a provision to record kurtosis values in an ASCII file. It was decided to monitor the kurtosis and, after two seconds, display the maximum value that had

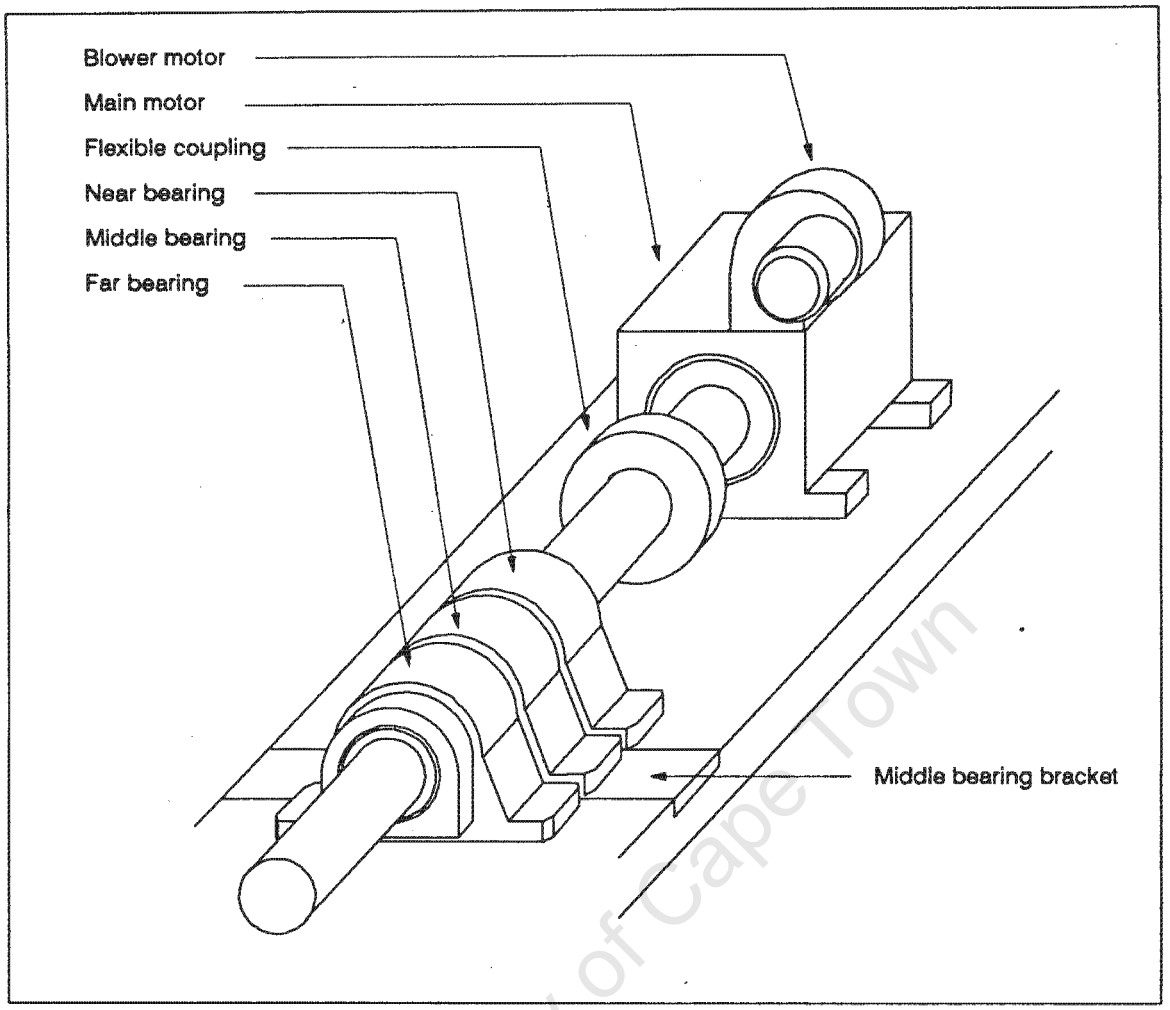


Figure 6.3 The Motor and Bearings used in Finding the Faulty Bearing.

occurred during those two seconds. This value would also be stored in the computer's memory. After two minutes of recording, these values would be written to a file on a floppy disc. Two minutes was chosen as the recording duration because it was thought that this would be long enough to cycle the car engine through its test sequence of acceleration and deceleration. The maximum values of kurtosis were recorded rather than the average values, since it was required that transient spikes be detected in the signal.

It was decided to limit the number of readings recorded by recording one every two seconds since roughly ten kurtosis values are calculated every second. If all these values were recorded there would be 1200 of them to process instead of only 60.

6.3 MONITORING A BEARING IN A NOISY ENVIRONMENT

6.3.1 Introduction

A further experiment to test whether Adaptive Noise Cancelling can be used to detect the state of a machine was conducted. A rig has been constructed by a Mechanical Engineering student at the University of Cape Town for the study of different bearing lubricants. A diagram of this rig is shown in figure 6.3 and the recording equipment used is shown in figure 6.4.

The main motor, which has a blower motor to cool it, is connected to the three bearings via a shaft that has a flexible coupling placed between the motor and the bearings. This flexible coupling was needed to allow for any misalignment between the motor and the bearings. The middle bearing was placed on a bracket that could be hydraulically elevated to simulate a heavy load. However, this bracket was not moved for any of these experiments.

The three bearings were in varying conditions. The two outer

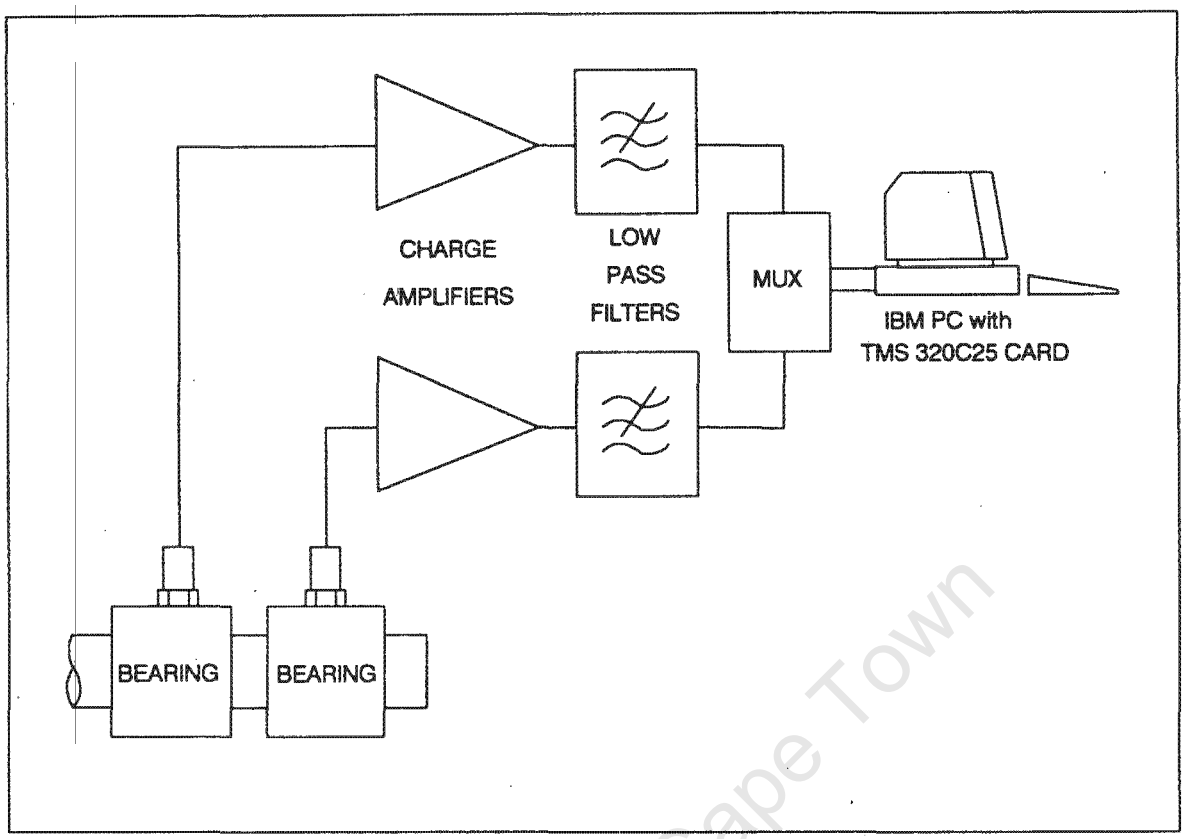


Figure 6.4 Equipment Used In Isolating a Faulty Bearing

bearings were in good order. The middle bearing, on the other hand, was very worn and produced a large amount of impulsive noise which could be heard and was also seen on the screen of the Machine Monitoring package. The aim of the experiments on this rig was to see if the Adaptive Noise Canceller could detect if the middle bearing was worn.

For simplicity it was decided to call the bearing nearest the motor the "near bearing" and to call the bearing furthest from the motor the "far bearing".

As in the experiment conducted on the Rolling Road, kurtosis was the measure used to determine the state of repair of the machine element under test. The conventional way of determining a bearing's condition is to observe the R.M.S. value of its signal.

However, this method is sensitive to changes in operating conditions, being dependent on the bearing load, speed, housing tightness, quality of lubricant and the bearing clearance [14]. It is therefore difficult to assess the bearing's condition using this method without a record of the past history of the bearing. As a result, it was decided to use kurtosis instead, since it is only dependent on the impulsiveness of the signal. It also detects wear earlier on than the R.M.S. method.

6.3.2 Determining the Source of Vibrations in the Rig

The first experiment on this rig was concerned with determining where the different vibrations came from within the rig since it was anticipated that not only the bearings and motors would produce signals. Having determined this, it could be decided where to place the primary and reference sensors so as to separate most effectively the three bearing signals using adaptive noise cancelling.

In order to determine where the different vibrations

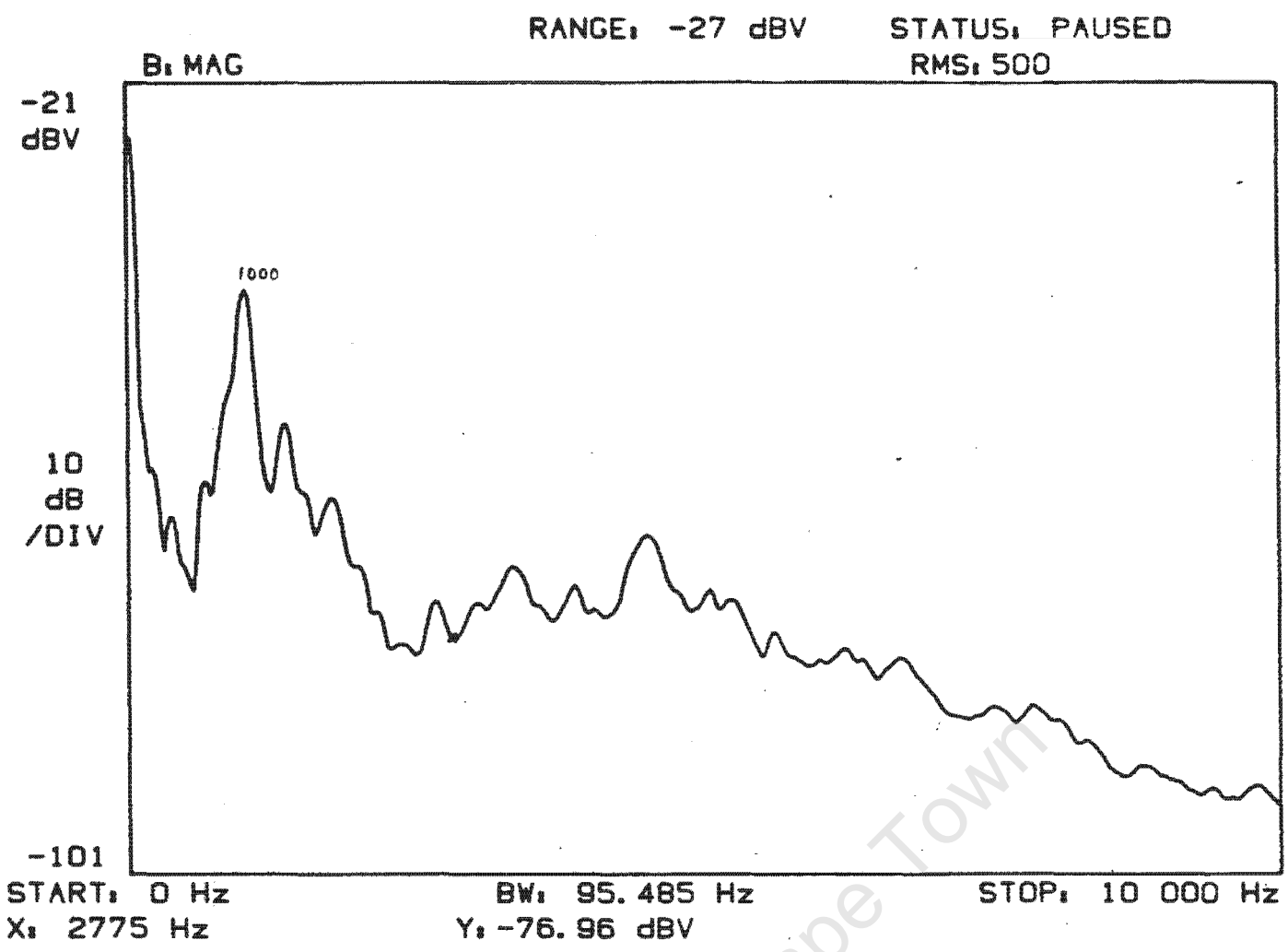
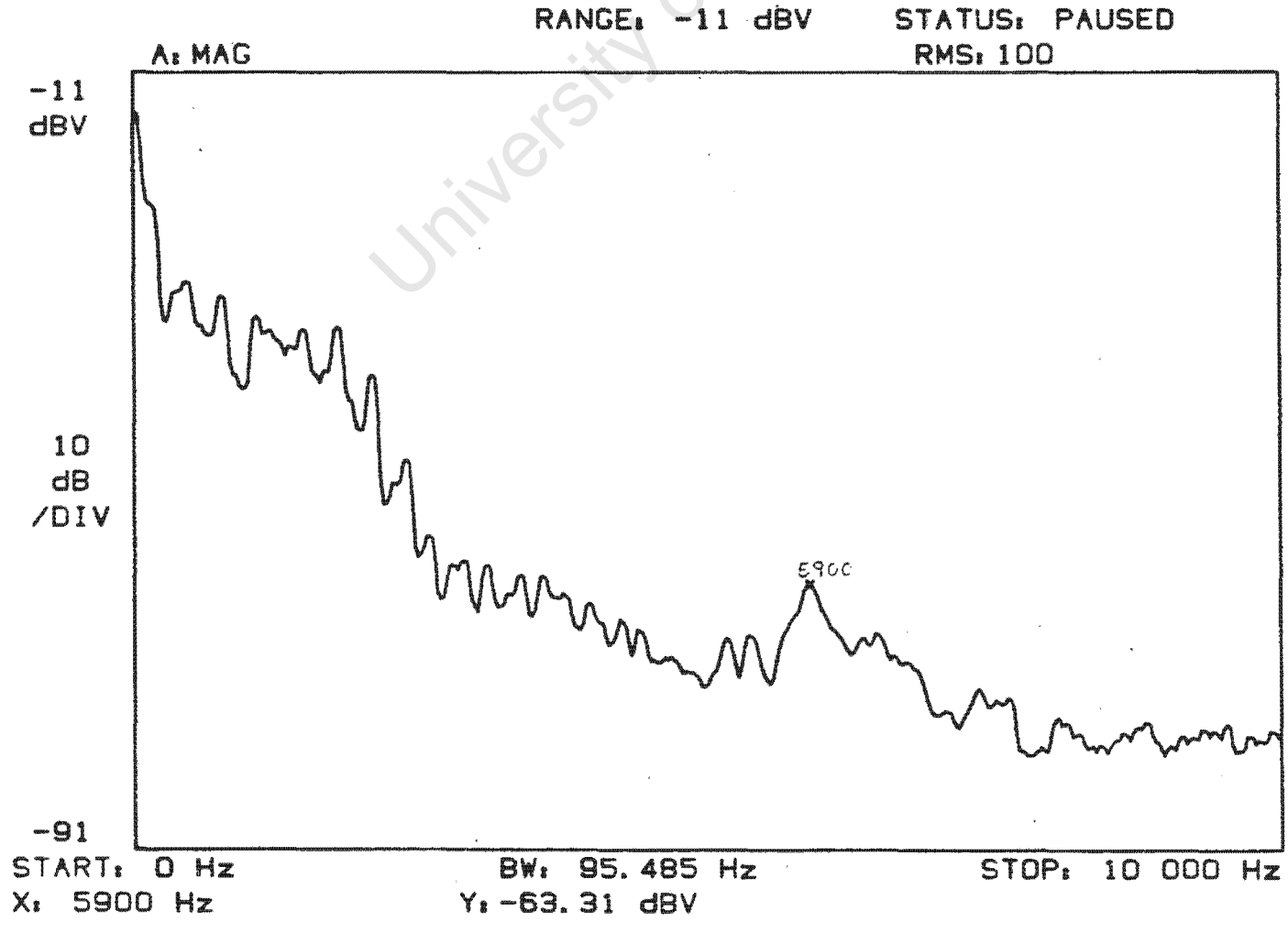


Figure 6.5 Spectrum: Sensor On Main Motor. Speed = 0 rpm.

Figure 6.6 Spectrum: Sensor On Near Bearing. Speed = 0 rpm.



originated from, it was decided to examine the spectra of signals recorded at different places around the rig. An accelerometer (a Bruel and Kjaer 4334S) was placed at the points shown in figure 6.3. With the motor rotating at various speeds, the spectra were recorded. These spectra are shown in figures 6.5 to 6.20.

The spectra shown in figures 6.5 to 6.8 were recorded with the main motor at rest. Figures 6.9 to 6.14 were recorded with the motor rotating at 490 rpm (a field voltage of 8V), while figures 6.15 to 6.20 were recorded with the motor rotating at 1000 rpm (a field voltage of 15V).

Each plotted spectrum is the average of 500 spectra. In some cases this has been reduced to 200 or 100 averages to minimise the likelyhood of external interference. The spectra have a range of 0 to 10 kHz since the response of the accelerometer was flat from 0 to just above 10 kHz. Above this the accelerometer starts to resonate, which places an upper limit on the frequencies that can be measured.

Figures 6.6 - 6.8 do not exhibit any definite peaks, but figure 6.5 shows that the main motor casing resonated at 1.0 kHz. The rotational speed of the blower motor is 1500 rpm (25 Hz), since it is a two-pole motor. However, it is unlikely that it was one of the blower moter harmonics that excited the main motor casing, since it would have to have been the 40th harmonic that caused the resonance.

When the main motor was energised, different peaks in the signals were recorded. These peaks remained constant in frequency regardless of motor speed: only the amplitude varied. This meant that the strong peaks in the spectra were not due to rotation, but due to resonances in the rig structure.

Signals recorded on the main motor casing, the rig frame and

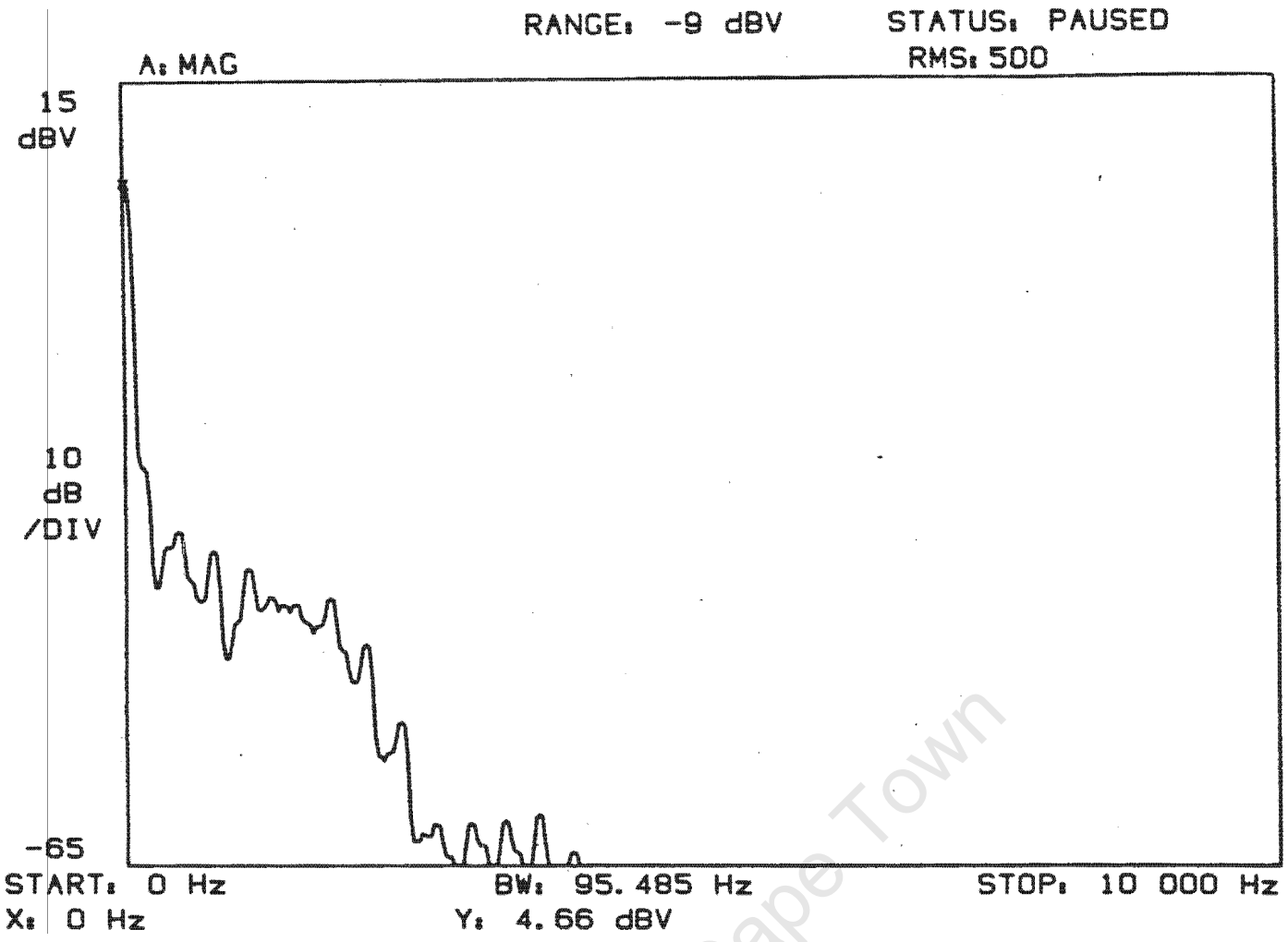
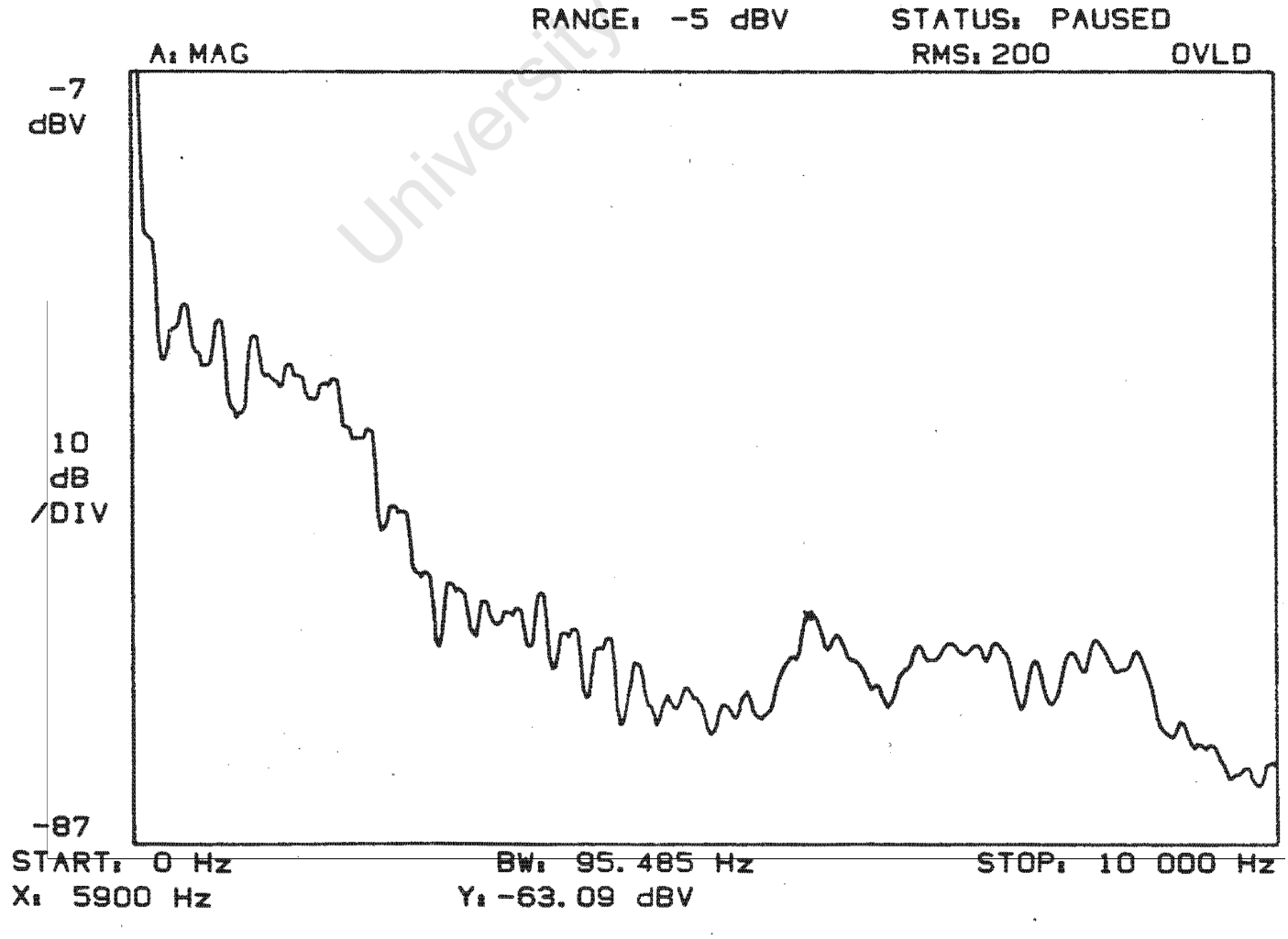


Figure 6.7 Spectrum: Sensor On Middle Bearing. Speed = 0 rpm.

Figure 6.8 Spectrum: Sensor On Far Bearing. Speed = 0 rpm.



the middle-bearing bracket showed peaks at 4.7 kHz and 2.7 kHz (figures 6.9 - 6.11, 6.15 - 6.17). The 4.7 kHz peak was almost certainly generated inside the main motor since the strongest occurrence of this frequency was on the main motor casing. The casing was rigidly attached to the rig frame, which explains why the 4.7 kHz peak was observed on the rig frame and on the middle- bearing bracket.

By contrast, those signals recorded on the three bearing housings showed a peak at 5.9 kHz (figures 6.12 - 6.14, 6.18 - 6.20). Close examination of the spectra obtained from opposite the far bearing (figures 6.11 and 6.17) also showed a small component at 5.9 kHz. This frequency was therefore strongest near the bearings, but the near bearing did not show as strong a peak as the other two.

It was thought that this frequency came from the shaft that connected the three bearings. The shaft extended for 30 cm past the end of the far bearing and was unsupported at this end. To test whether this was plausible, the resonant frequency of a shaft unsupported at one end was calculated using the following equation [15]:

$$f_1 = \frac{\pi}{2l^2} \sqrt{\frac{Q\ k^2}{\sigma}}\ \beta_1^2$$

where f_1 is the fundamental resonant frequency, l is the length of the shaft in centimetres, Q is the elasticity modulus in Dynes per square centimetre, k is half the radius of the shaft in centimetres, σ is the density of the shaft and β is dependent on the harmonic needed to be calculated. The following values were used in the calculation:

$l = 30$ cm
$Q = 19.5 \times 10^{11}$ Dynes/cm^2
$k = 0.75$ cm
$\sigma = 7.7$ g/cm^3
$\beta = 0.597$

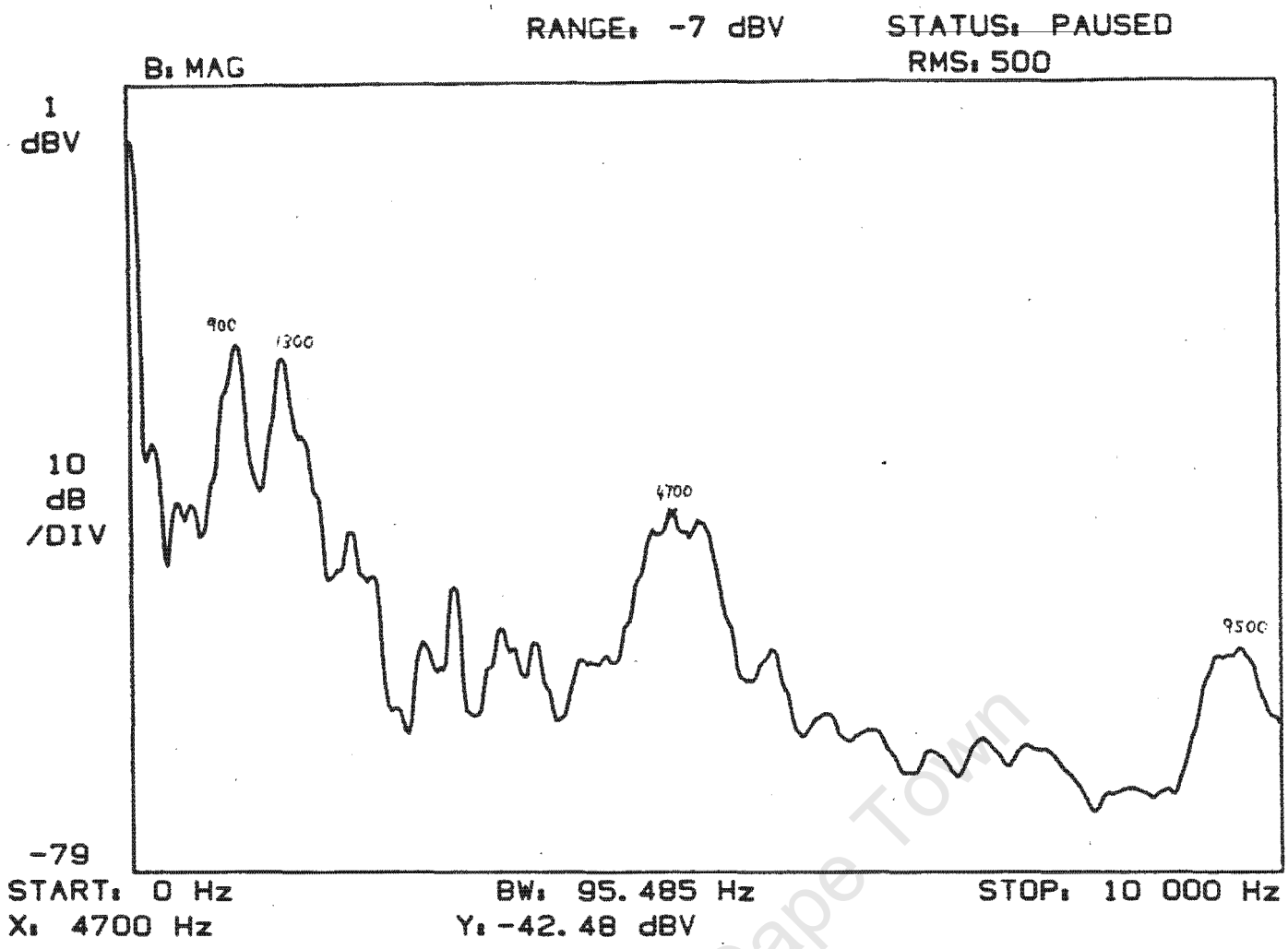
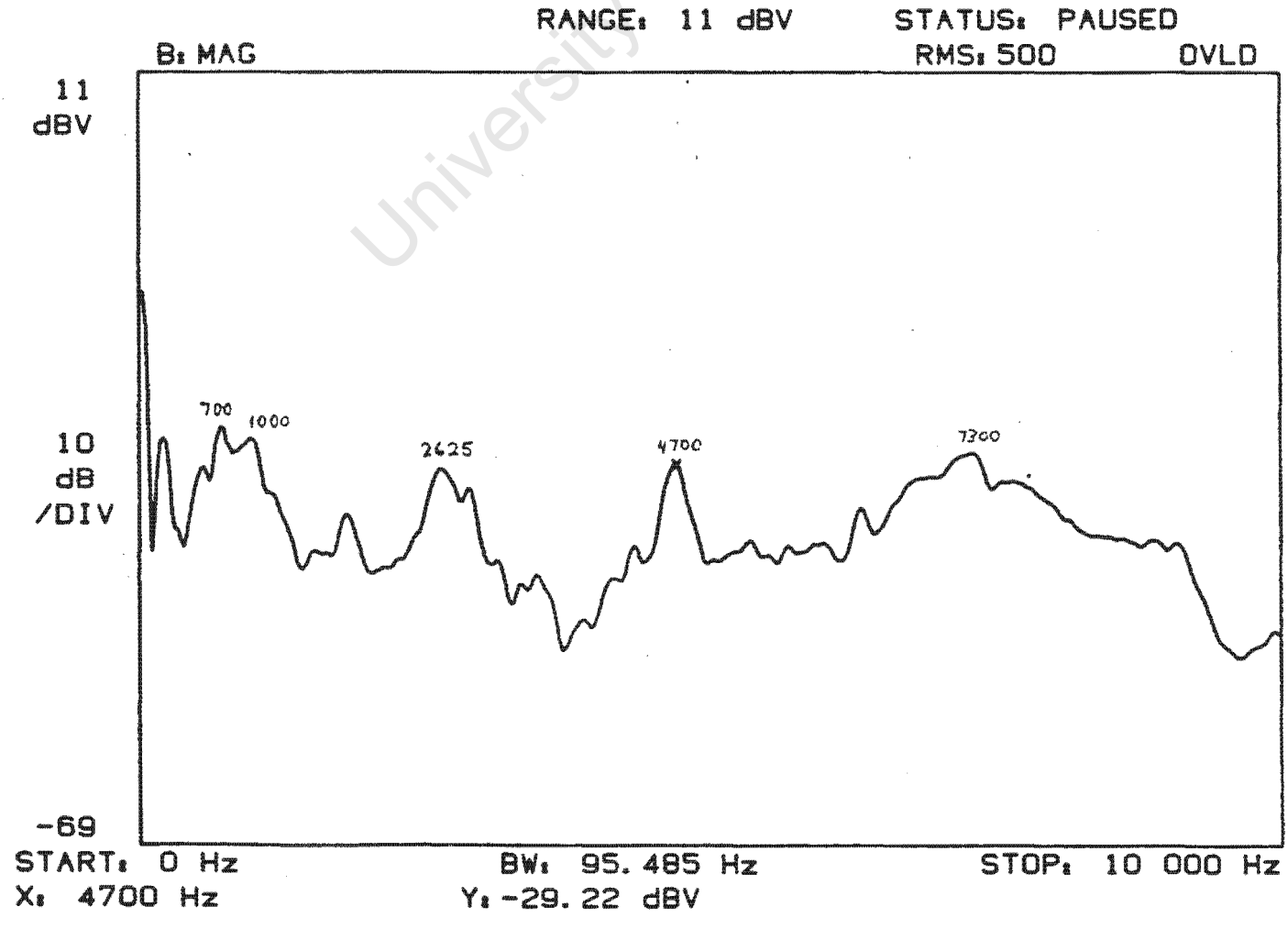


Figure 6.9 Spectrum: Sensor On Main Motor. Speed = 490 rpm.

Figure 6.10 Spectrum: Sensor On Middle Bearing Bracket. Speed = 490 rpm.



With these values, the resonant frequency of the shaft is computed to be 235 Hz and this result disproves the theory that the shaft produced a frequency of 5.9 kHz.

In the signals obtained from the main motor casing (figures 6.9 and 6.15) there is a small peak at 9.6 kHz. This seems to be simply the second harmonic of the primary resonance of the main motor casing, since 9.6 kHz is roughly two times 4.7 kHz.

Using these results it was decided to use the bearing housings as the positions for the Primary and Reference sensors in the experiment to determine the state of each bearing (section 6.3.3) since the signals obtained from these three positions were the most similar and would therefore produce the best noise cancelling.

6.3.3 Using Noise Cancelling to Find the State of Each Bearing

The aim of this experiment was to verify that adaptive noise cancelling could separate the signals of the three bearings so as to allow us to determine what state each bearing was in.

This experiment was concerned with resolving the three bearing signals from the noise around them and from each other. The Primary sensor was placed on the bearing whose signal was to be resolved and the Reference sensor was placed on each of the other two bearings in turn. Since there were three bearings present on the shaft, there were six different ways of arranging the Primary and Reference sensors. This series of experiments was repeated for three different speeds of the motor. The signals from the two sensors were fed into the Adaptive Noise cancelling package for analysis. A diagram of the equipment used to record the data is shown in figure 6.4.

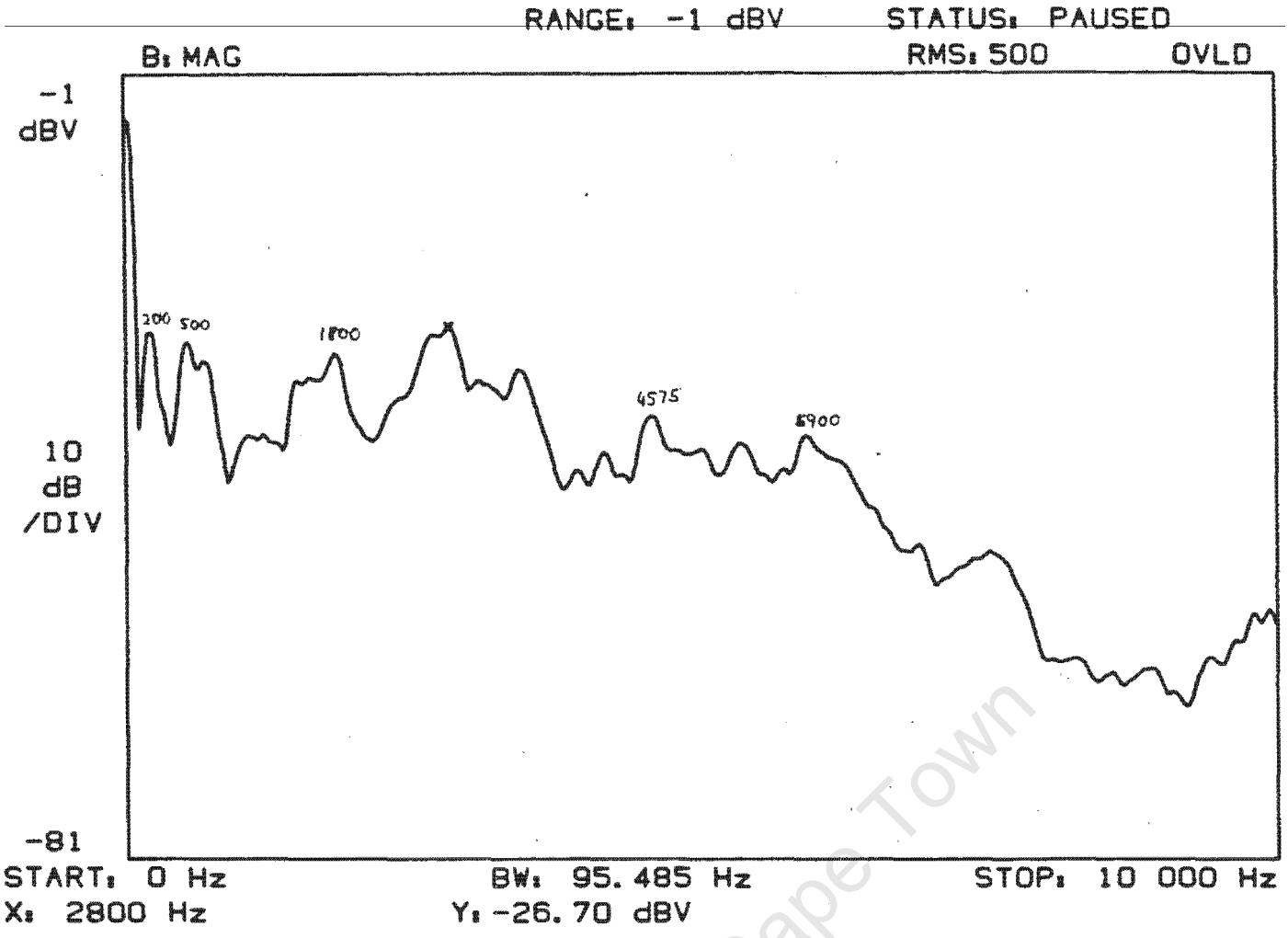
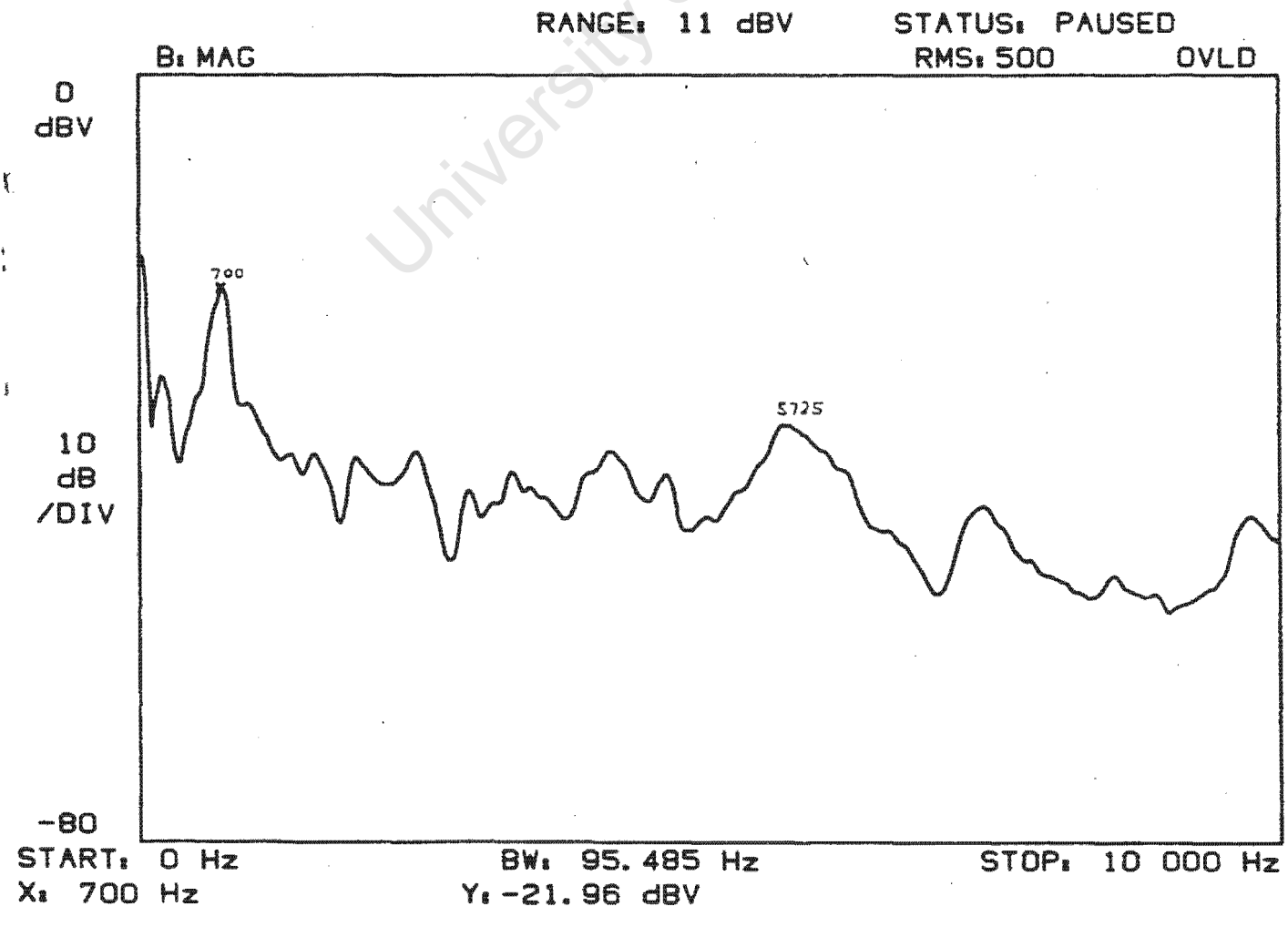


Figure 6.11 Spectrum: Sensor Opposite Far Bearing. Speed = 490 rpm.

Figure 6.12 Spectrum: Sensor On Near Bearing. Speed = 490 rpm.



The kurtosis of the uncorrelated signal was recorded for each sensor position in the manner described in section 6.3.1. The spectra of all the signals in the adaptive filter (Primary, Reference, Correlated and Uncorrelated) were also observed. The spectra could not be printed, however, because the computer used was not equipped with a parallel port.

The kurtosis values that were recorded were stored in files that are present on the disc at the back of this thesis. The names of these files tell which test they came from.

Files recorded without Noise cancelling.

1) The first letter of the file determines on which bearing the sensor was placed. 'F' means the far bearing, 'M' means the middle bearing and 'N' means the near bearing.

2) The files that were recorded without noise cancelling operating have the letters 'NC' as the second and third characters of the file name.

3) The numbers after the letters determine at what voltage the field windings of the main motor was set. A field voltage of 8V corresponds to a speed of 490 rpm, 10V to 650 rpm and 15V to 1000 rpm.

   e.g. If the file had been recorded on the near bearing with the field voltage set at 10 volts and with no noise cancelling being used, then the file would be called 'NNC010'.

Files recorded with Noise cancelling.

1) The first character of the file name determines on which bearing the Primary sensor was placed. This is the bearing which was having the noise cancelled from its signal. As with the non-noise cancelling files,

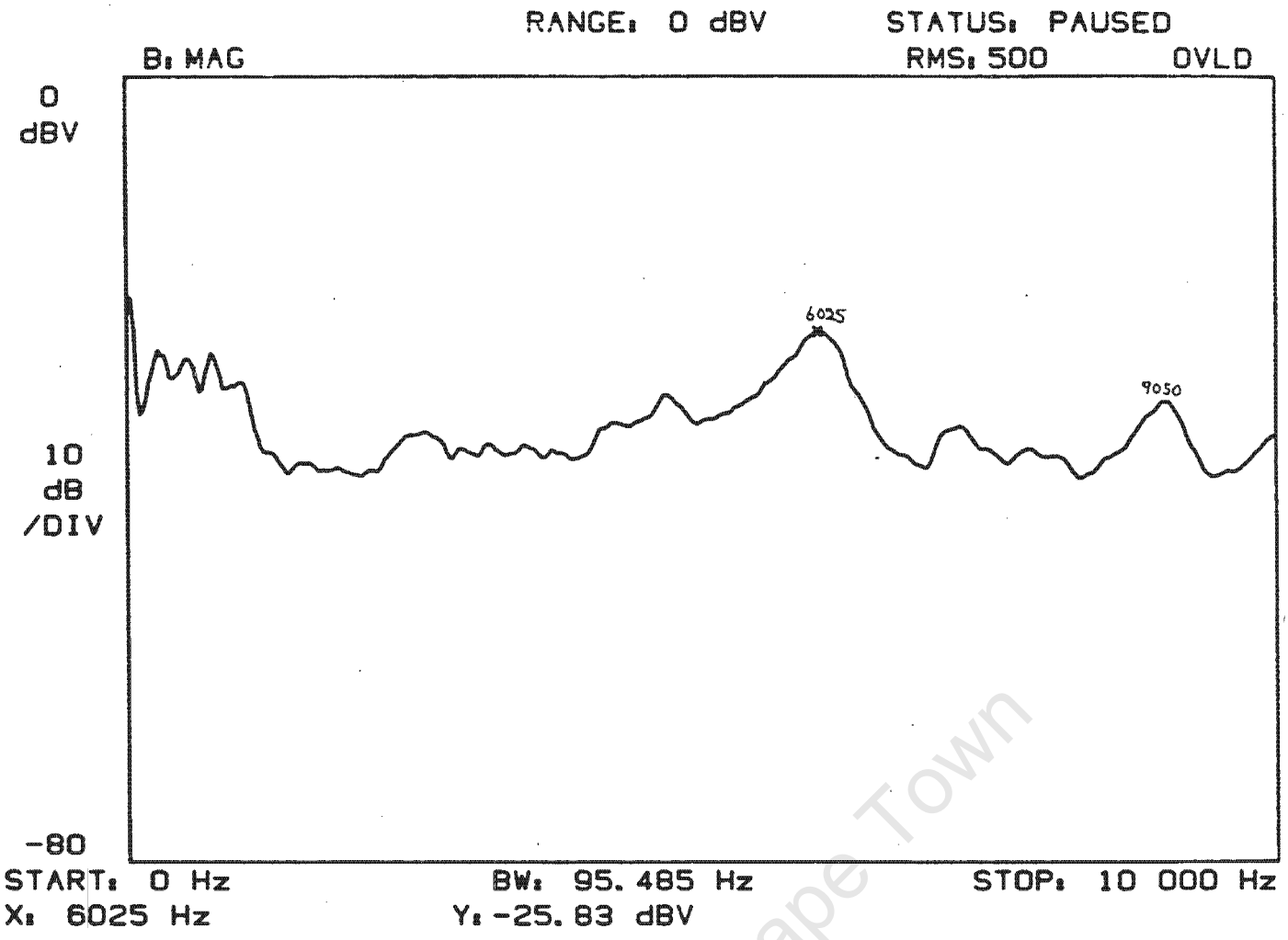
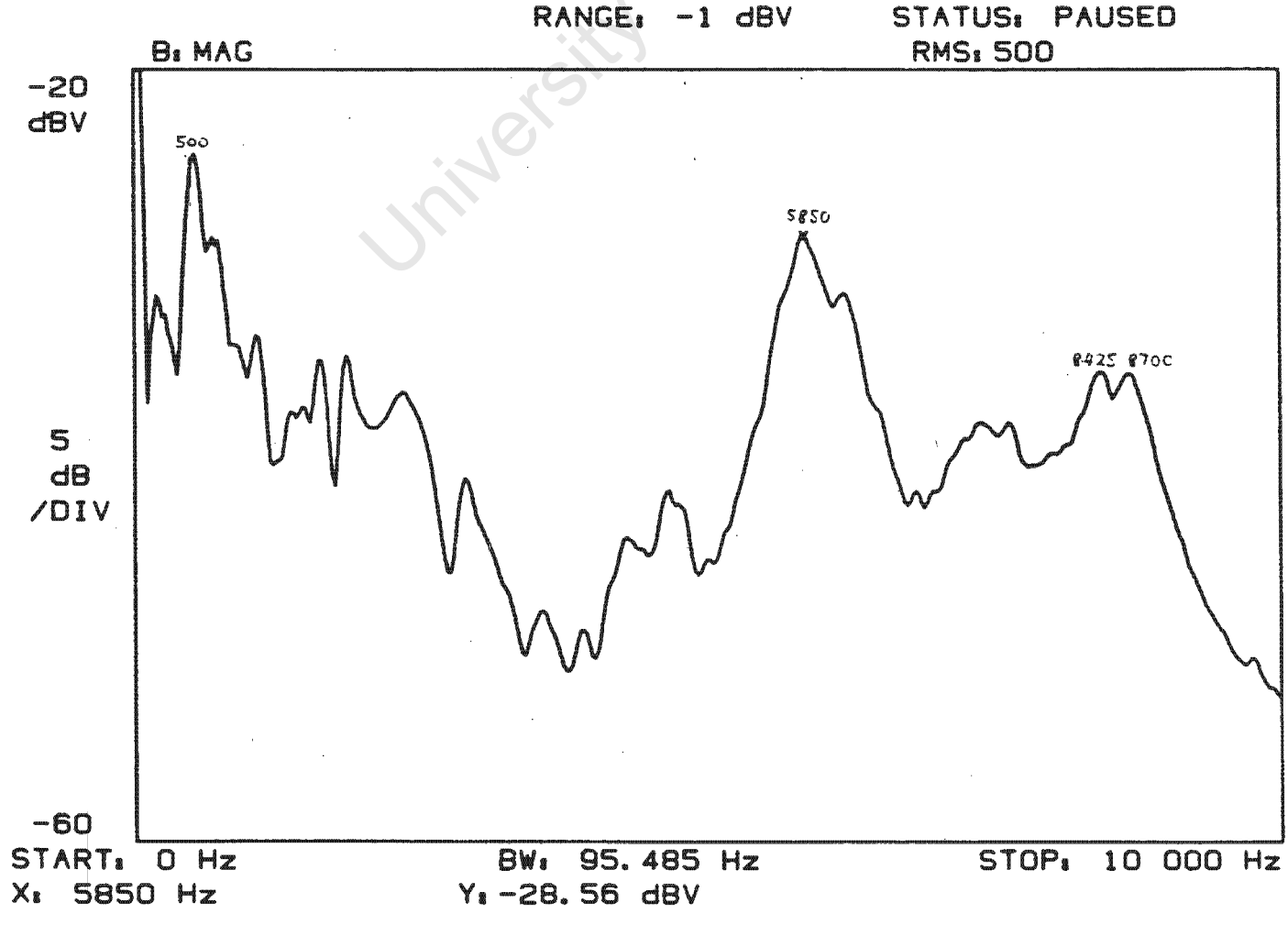


Figure 6.13 Spectrum: Sensor On Middle Bearing. Speed = 490 rpm.

Figure 6.14 Spectrum: Sensor On Far Bearing. Speed = 490 rpm.



'F' means the far bearing, 'M' means the middle bearing and 'N' means the near bearing.

2) The second character shows on which bearing the Reference sensor was placed.

3) The numbers after the letters determine at what voltage the field windings of the main motor was set. A field voltage of 8V corresponds to a speed of 490 rpm, 10V to 650 rpm and 15V to 1000 rpm.

   e.g. If the bearing under observation had been the middle one and the reference sensor had been placed on the far bearing, with the field voltage set at 15 volts, then the file name would be 'MF015'.

There are two exceptions to this rule. The files MID001 and MID002 were recorded with the Primary sensor placed on the middle bearing and the Reference sensor placed on the middle bearing bracket. This was to test whether any noise coming from the bracket was contaminating the bearing signal. Both these records were recorded with the motor rotating at 650 rpm (the field voltage set to 10V).

All the files recorded are listed in tables 6.1, 6.2 and 6.3. For brevity, only the average kurtosis value is given, along with the variance of the readings in the file. Also given is a test statistic, **z**, which is a measure of how different the records using noise cancelling are from the ones without noise cancelling. The formula used to calculate **z** is given by [16]:

$$\mathbf{z} = \frac{\bar{X}_1 - \bar{X}_2}{[(\sigma_1^2 + \sigma_2^2)/N]^{0.5}} \qquad 6.1$$

$\bar{X}_1$ and $\bar{X}_2$ are the average values of the kurtosis in the record without and with noise cancelling respectively. The

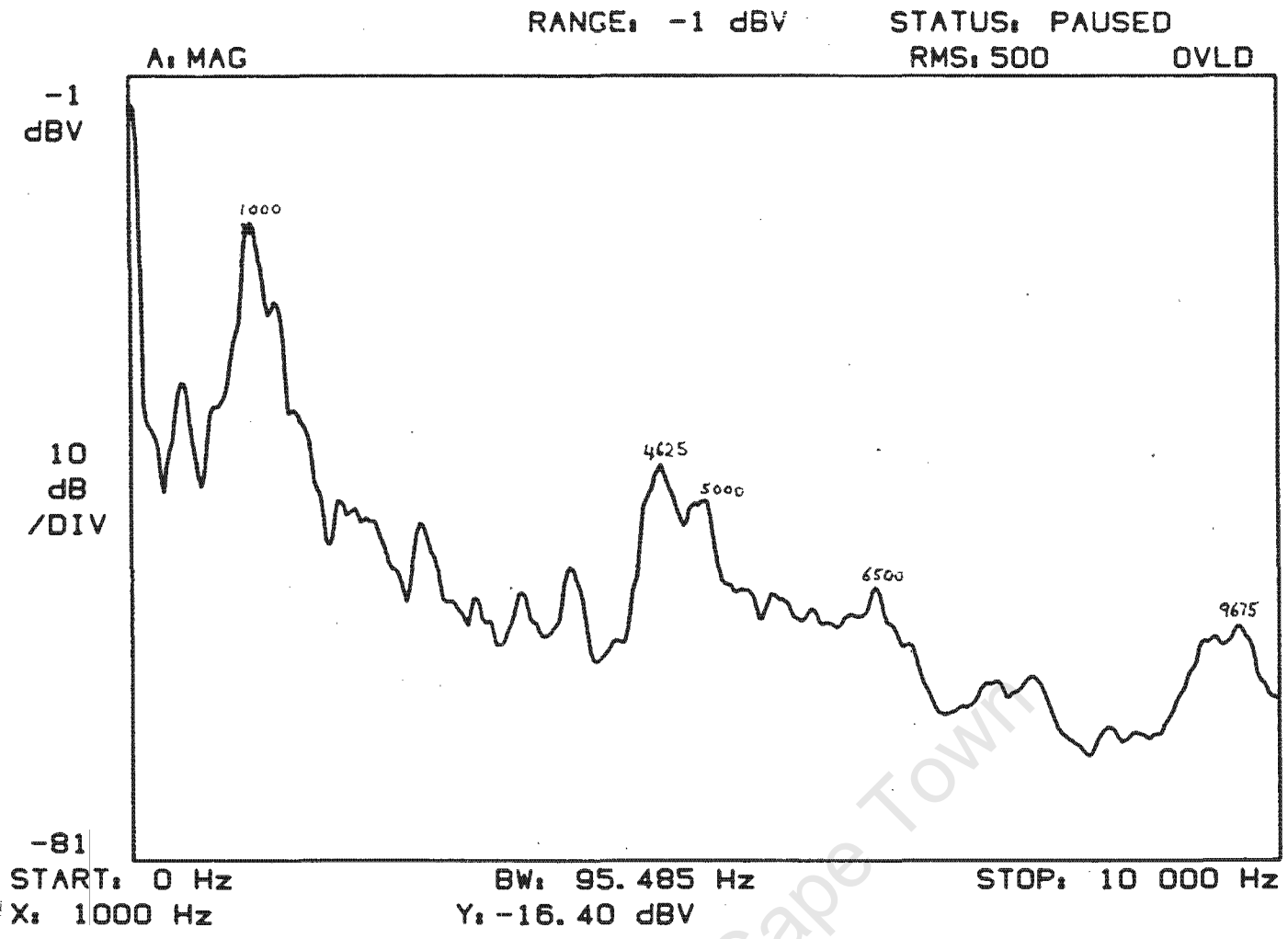
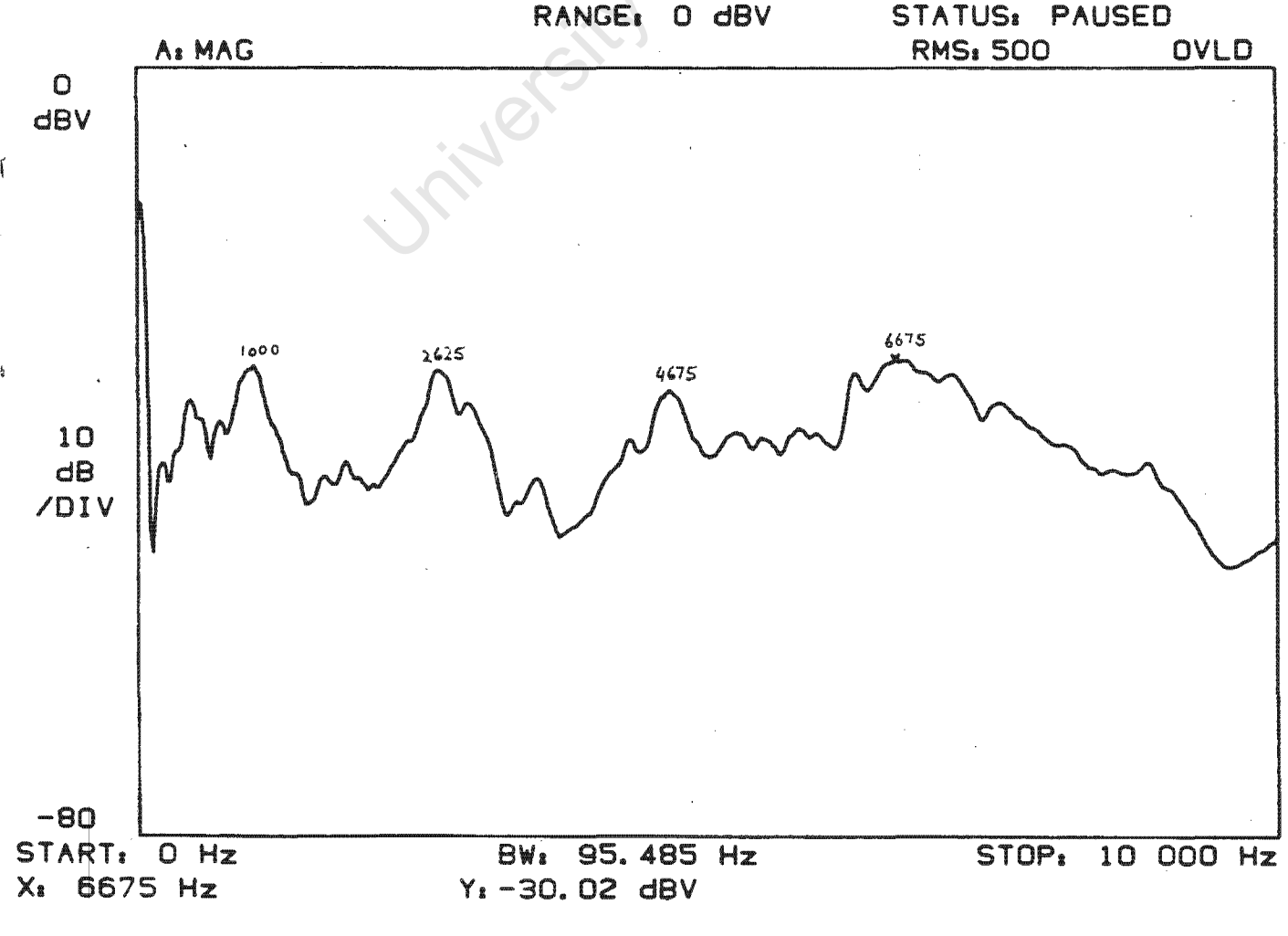


Figure 6.15 Spectrum: Sensor On Main Motor. Speed = 1000 rpm.

gure 6.16 Spectrum: Sensor On Middle Bearing Bracket. Speed = 1000 rpm.



σ^2's are the variances of the data in the two records and N is the number of kurtosis values in each record. As a benchmark, a **z** value of 3.48 means that the probability of the two records under comparison being the same is 1 in 2000. A **z** value of 5.00 gives a probability of less than 1 in a million. In Tables 6.1, 6.2 and 6.3 each of the **z** values is calculated by comparing the record under whose name the **z** value is displayed with the record to its extreme left. e.g. The **z** value under the record FN008 was calculated by comparing the records FN008 and FNC008.

MAIN MOTOR FIELD VOLTAGE = 8V (490 rpm)			
FILE NAME:	FNC008	FM008	FN008
AVERAGE KURTOSIS:	6.33	3.92	3.78
VARIANCE:	6.75	0.564	0.335
z		6.90	7.42
FILE NAME:	MNC008	MF008	MN008
AVERAGE KURTOSIS:	14.0	17.1	15.8
VARIANCE:	6.84	9.49	7.40
z		5.94	3.69
FILE NAME:	NNC008	NF008	NM008
AVERAGE KURTOSIS:	9.95	6.29	3.06
VARIANCE:	22.9	4.38	0.775
z		5.43	11.0

Table 6.1

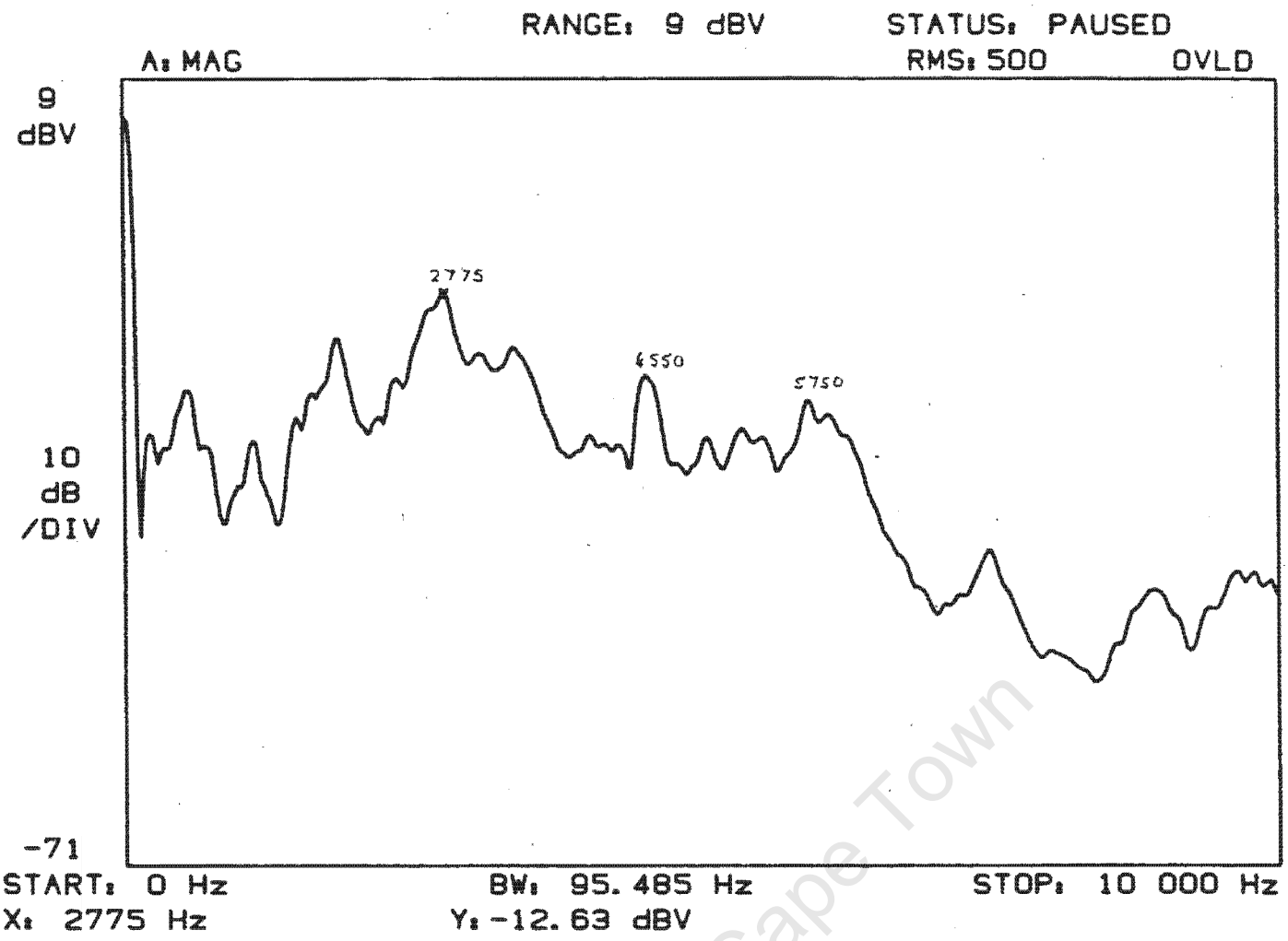
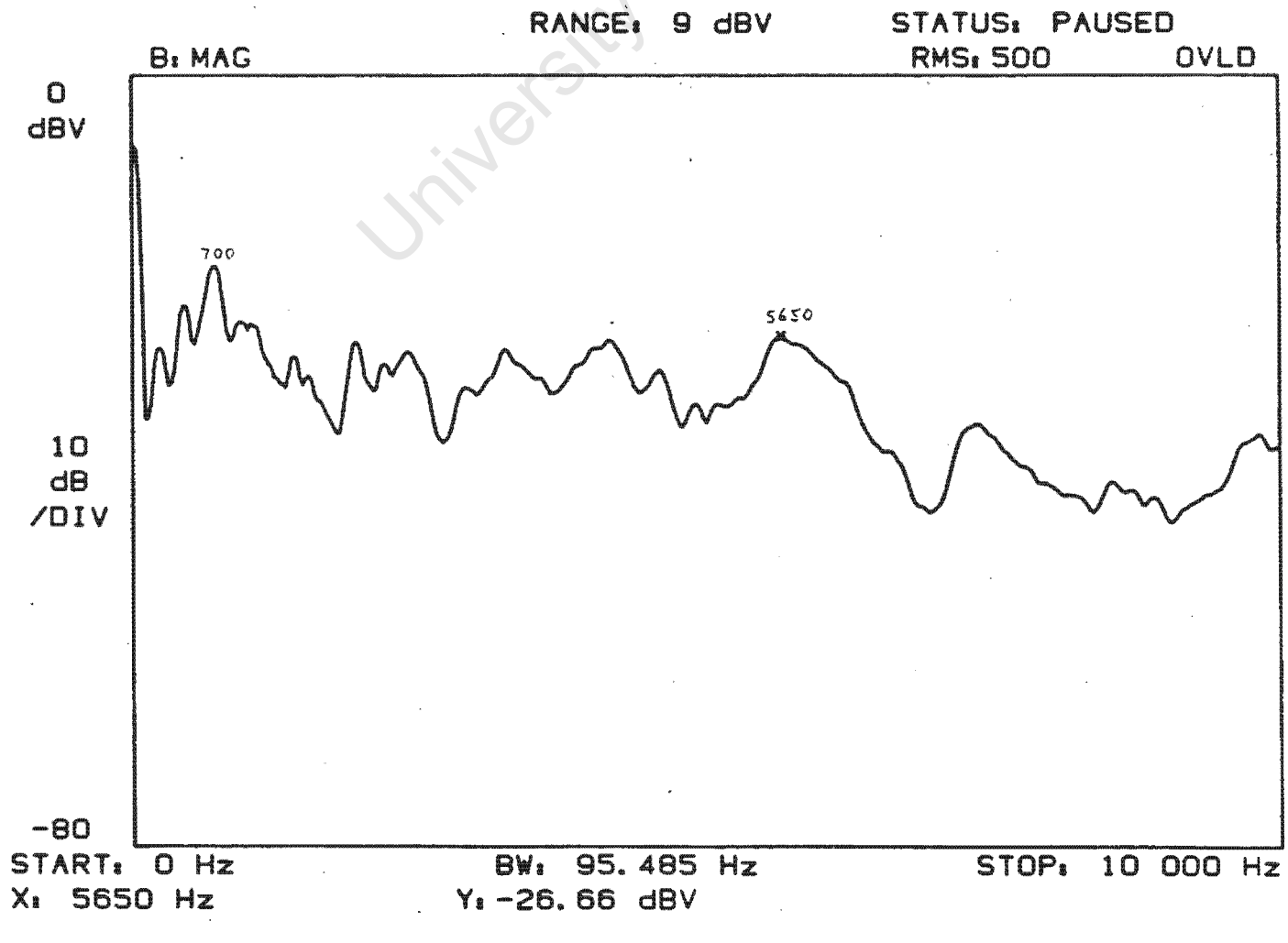


Figure 6.17 Spectrum: Sensor Opposite Far Bearing. Speed = 1000 rpm.

Figure 6.18 Spectrum: Sensor On Near Bearing. Speed = 1000 rpm.



MAIN MOTOR FIELD VOLTAGE = 10V (650 rpm)					
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	FNC010 10.4 20.3	FM010 3.54 0.0823 11.8	FN010 3.71 0.149 11.5		
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	MNC010 7.02 1.06	MF010 6.09 0.541 5.69	MN010 21.8 17.4 26.6	MID201 8.06 2.54 4.25	MID202 10.8 14.9 7.33
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	NNC010 13.0 15.7	NF010 5.24 2.09 14.3	NM010 5.02 4.65 13.7		

Table 6.2

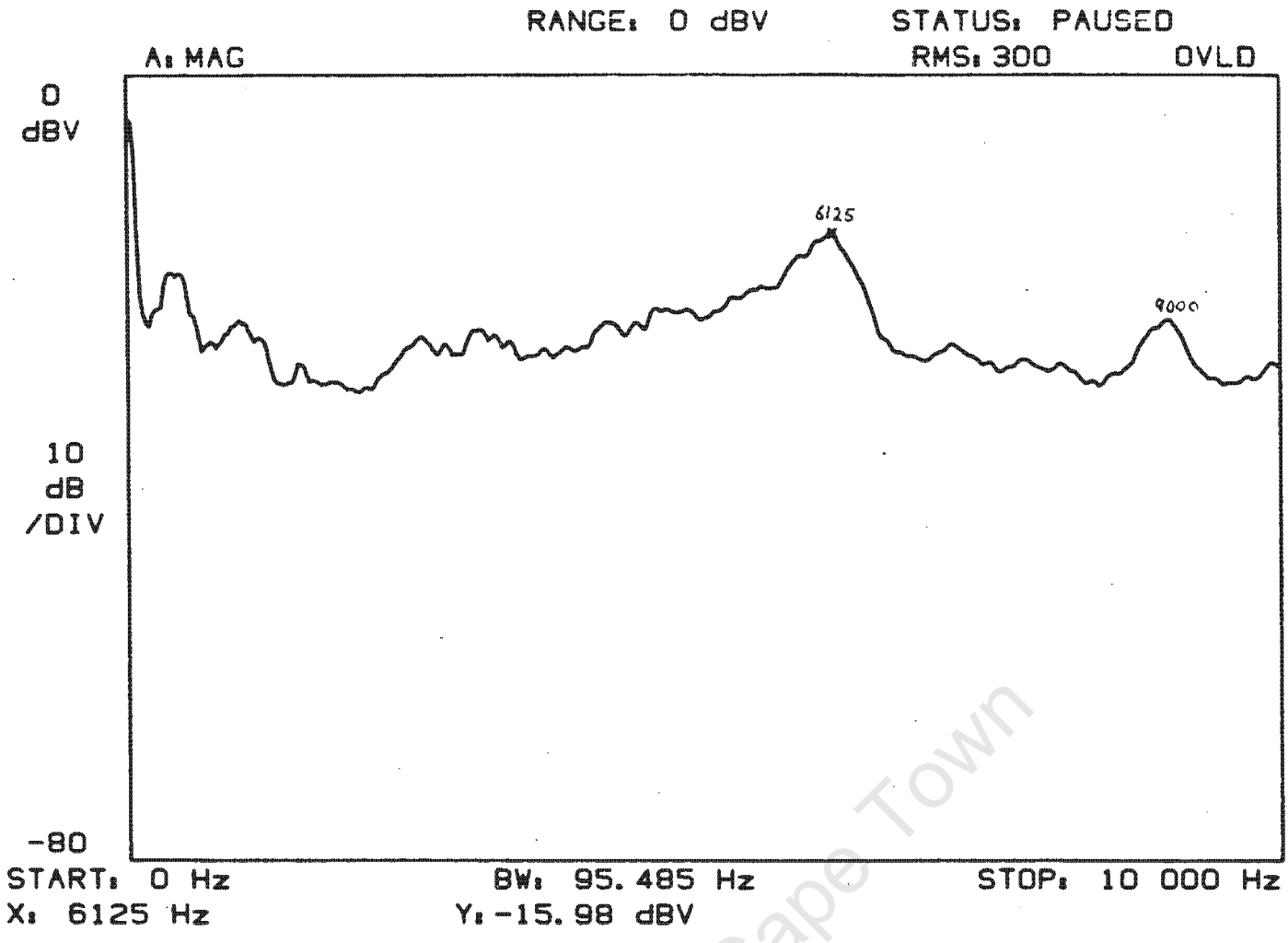
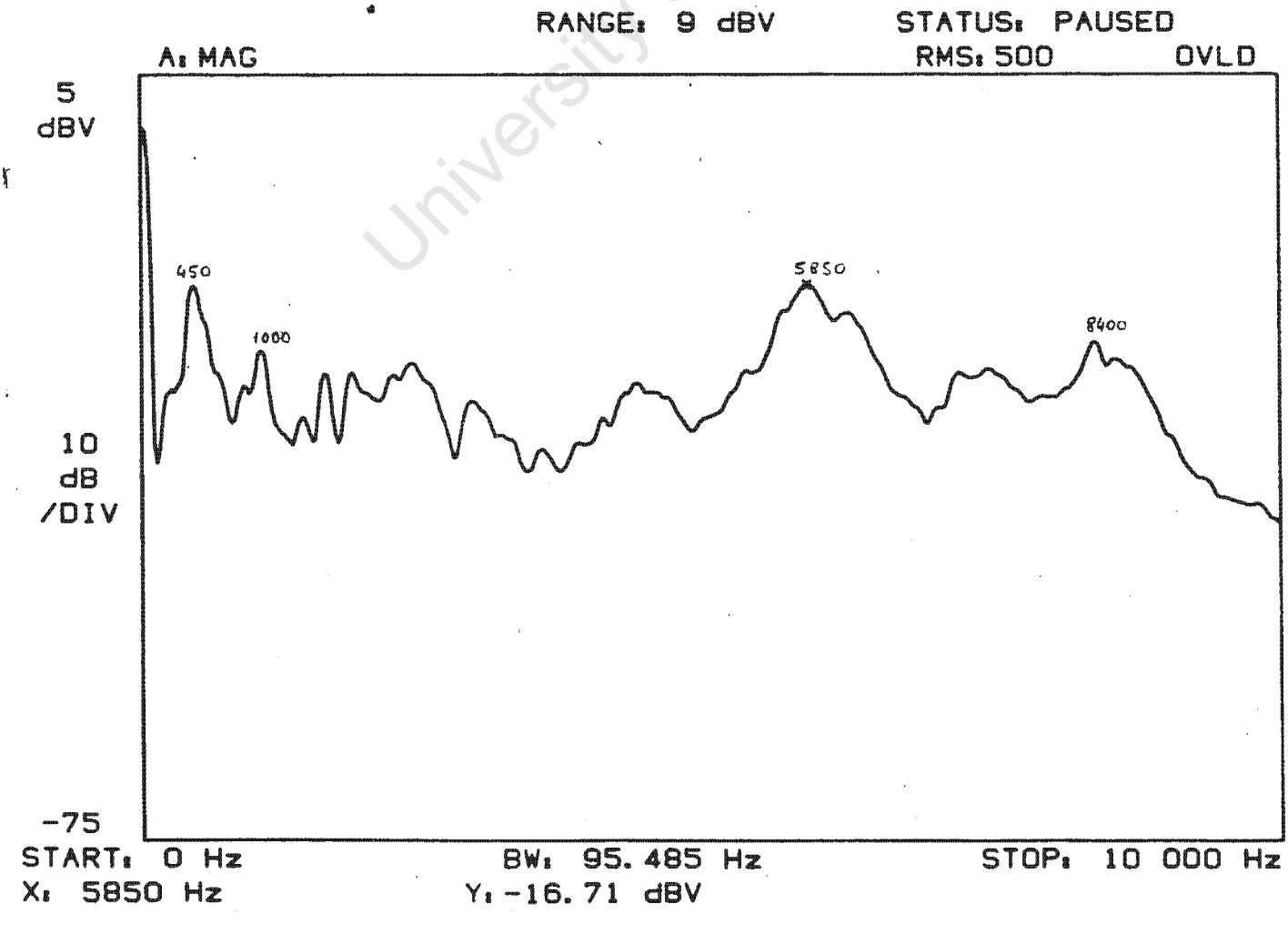


Figure 6.19 Spectrum: Sensor On Middle Bearing. Speed = 1000 rpm.

Figure 6.20 Spectrum: Sensor On Far Bearing. Speed = 1000 rpm.



MAIN MOTOR FIELD VOLTAGE = 15V (1000 rpm)			
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	FNC015 11.6 6.02	FM015 3.52 0.153 25.2	
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	MNC015 4.12 0.133	MF015 6.06 0.576 17.8	MN015 5.76 0.344 18.4
FILE NAME: AVERAGE KURTOSIS: VARIANCE: z	NNC015 9.94 8.52		NM015 2.94 0.238 18.3

Table 6.3

6.3.4 Analysing the Kurtosis Results of The Noise-Cancelling Experiment

The results concerning the far bearing were analysed first of all. This is a good bearing, as previously stated. Without noise cancelling operating, the average values of kurtosis were between 6.33 and 11.6, showing a very impulsive signal. However, with the noise cancelling in operation, the kurtosis values dropped to between 3.52 and 3.92. This means that external noise was affecting the signal obtained from the far bearing when there was no noise cancelling taking place.

The noise gave a spurious reading that could be construed as meaning that the far bearing was worn when in fact it was not. With the noise cancelling operational it can be seen

that the signal is not impulsive at all. In fact it has a Gaussian probability curve since the kurtosis values are about 3.0, the value for a Gaussian signal. This kind of signal is what was expected since new bearings produce a pressure distribution that is Gaussian about a mean [14].

The near bearing was also expected to produce a signal with a kurtosis of around 3.0 with noise cancelling taking place, since this bearing was also new. This did not occur but a significant reduction in the kurtosis values compared to those recorded without noise cancelling was observed. Without noise cancelling the kurtosis values lay between 9.94 and 13.0 and with noise cancelling the values lay between 2.94 and 6.29. This suggests that the clean signal was more impulsive than a Gaussian signal, but not much.

It was concluded, in the case of the near bearing, that the noise cancelling had worked despite the kurtosis values being slightly higher than expected. This conclusion was made after examination of the **z** values associated with the records recorded with the Primary sensor on the near bearing, the lowest value being 5.43. This value means that the probability of the noise cancelling not operating is less than 1 in 17 million.

The two outer bearings' kurtosis decreased in value when noise cancelling was used. However, the middle bearing's kurtosis increased under the same conditions. These values are much higher than those recorded from the two outer bearings (from 6.06 to 21.8) and it can be concluded that this bearing is the one which is worn.

An interesting observation is that the kurtosis of the middle bearing decreased as the rotational speed increased. With the motor rotating at 490 rpm the average kurtosis was approximately 16.0. When the speed of the motor was raised to 650 rpm the kurtosis dropped to around 10.0. With the speed set to 1000 rpm the average kurtosis was about 6.0.

This decrease in kurtosis was because of the following reason. When the bearing is rotating slowly, the impulses it produces will be far apart and a large kurtosis value will result. As the bearing speeds up, however, the impulses become closer together until they are so close that they are indistinguishable from ordinary Gaussian noise, which has a low kurtosis value.

With the Primary sensor placed on the far bearing and the Reference sensor placed on the near bearing it was not expected that good results would occur. This was because it was assumed that there would be no correlated signal coming from the near bearing.

However, noise cancelling occurred as well as in the cases where the two sensors were placed on adjacent bearings. This was because the middle bearing was producing most of the interference. This noise was received at both the far and near bearings simultaneously and through virtually identical signal paths. The correlated noise therefore did not come mainly from the near bearing but from the middle bearing, which explains why the noise cancelling worked in this instance.

6.3.5 Analysing the Spectra Observed in The Noise-Cancelling Experiment

No graphics-dumping facilities were available and also, the Noise Cancelling Package could not average the spectra displayed. Therefore, only rough estimates of the spectra could be obtained while noise cancelling was in progress.

The spectrum of the Primary signal for every record was compared to that of the Uncorrelated signal (the clean signal). However, there were no differences seen between these two spectra and it was concluded that simply observing the spectrum of the Uncorrelated signal could not tell us whether any noise cancelling was taking place. It was hoped

that some decrease in the size of the frequency peaks present in the Primary signal would occur, but none was observed.

It might be argued that, since bearings produce random noise, it is expected that there will be no observable change in the spectra observed. The spectra referred to in the previous paragraph are the resonant frequencies generated, not by the bearings, but by the rig structure, which can easily be observed on a spectrum-analyser.

6.3.6 Conclusions Regarding The Noise Cancelling Experiment

It has been shown by analysis of the kurtosis values recorded that Adaptive Noise Cancelling did separate the signals generated by three closely spaced bearings. This enabled us to conclude that the middle bearing was the one which was worn and needed replacing. Without the Noise Cancelling working it was impossible to tell in what condition each bearing was in.

It was also concluded that it cannot be seen if the noise cancelling algorithm is working simply by comparing the spectra of the Primary and Uncorrelated signals.

Finally, it was shown that kurtosis is a good measure of the impulsiveness of a signal and is an efficient way of determining the present state of a machine component.

CHAPTER 7

CONCLUSIONS AND IMPROVEMENTS

7.1 CONCLUSIONS

1) From the tests done in chapter 3 it can be concluded that the LMS algorithm gives comparable results to the RLS method when there is little correlated signal present in the system.

2) The RLS algorithm is not suitable for real-time use for the following reasons. The RLS algorithm becomes unstable when time-updates are included in the algorithm. The order-update RLS algorithm of [11] does not become unstable, but it is an $O(N^2)$ algorithm whenever the filter weights need updating. Also, the RLS method has too many calculations to perform to allow the full performance of the A/D converter on the TMS320C25 board to be used.

3) Noise cancelling does not function if the distance between the Primary and Reference sensors is too large. In other words, if the time taken for the signal originating at the Reference sensor to travel to the Primary sensor is longer than it takes to travel through the adaptive filter then no noise cancelling takes place.

4) From the bearing experiment in chapter 6 it has been found that kurtosis is a good measure of the condition of a bearing. On the other hand, it is not possible to discover a bearing's condition simply by observing the spectrum obtained when the bearing is rotating.

5) We have conclusive proof that adaptive noise-cancelling does separate the desired machine element signal from surrounding noise. Therefore it can be stated that Adaptive Noise-Cancelling is suitable for use in this sort of environment and also, that this thesis has achieved what was intended. These intentions were:

   a) To test whether Adaptive Noise-Cancelling would work in the area of machine-condition monitoring,
   b) To see whether kurtosis was a sufficient criterion in estimating machine element conditions and
   c) To develop a real-time machine monitoring package that could test these two theories.

7.2 SUGGESTED IMPROVEMENTS TO THE MACHINE MONITORING PACKAGE

It would have been easier to develop a machine monitoring system with a signal-processing board that contained more than one analogue input and output. If this condition had been met, then no external multiplexer would have been necessary.

The graphics display is of rather poor quality, speed being the prime requisite. At the moment, the signal is represented by dots instead of a continuous line. When the user wishes to freeze the display for closer analysis, it would be advantageous to connect the dots that form the display to make the display clearer.

BIBLIOGRAPHY

[1]: Paul Bremer, "Adaptive Noise Cancelling For Bearing Analysis in Road Vehicles", BSc. Dissertation, Dept. of Electronic Engineering, University of Cape Town, 1987.

[2]: Bernard Widrow and Samuel Stearns, "Adaptive Signal Processing", Prentice-Hall, United States of America, 1985.

[3]: D.D. Falconer and L. Ljung, "Application of Fast Kalman Estimation to Adaptive Equalization", IEEE Trans. Comms., vol COMM-26, pp. 1339-1446, October 1978.

[4]: George Carayannis, Dimitris G. Manolakis and Nicholas Kalouptsidis, "A Fast Sequential Algorithm for Least-Squares Filtering and Prediction", IEEE Trans. Acoust, Speech, Signal Processing, vol ASSP-31, pp 1394-1402, Dec. 1983.

[5]: John M. Cioffi and Thomas Kailath, "Fast, Recursive-Least-Squares Transversal Filters for Adaptive Filtering", IEEE Trans. Acoust, Speech, Signal Processing, vol ASSP-32, pp 304-337, Apr. 1984.

[6]: S.T. Alexander, "Fast Adaptive Filters: A Geometric Approach", IEEE ASSP Magazine, pp 18-28, Oct. 1986.

[7]: C.L. Lawson and R.J Hanson, "Solving Least Squares Problems", ppg 36-38, Englewood Cliffs, NJ: Prentice Hall, 1974.

[8]: Xiao-Hu Yu and Zhen-Ya He, "A Class of Mixed Transversal and Ladder Adaptive Filters with Pure Order Updates", IEEE Trans. Acoust., Speech, Signal Processing, vol 37, pp 1464-1468, Sept. 1989.

[9]: Jean-Luc Botto and George V. Moustakides, "Stabilizing the Fast Kalman Algorithms", IEEE Trans. Acoust., Speech, Signal Processing, vol 37, pp 1342-1348, Sept. 1989.

[10]: Michael G. Larimore, John R. Treichler and C. Richard Johnson Jr, "SHARF: An Algorithm for Adapting IIR Digital Filters", IEEE Trans. Acoust., Speech, Signal Processing, vol ASSP-28, pp 428-440, Aug. 1980.

[11]: Sasan Houston Ardalan and L. James Faber, "A Fast ARMA Transversal RLS Filter Algorithm", IEEE Trans. Acoust, Speech, Signal Processing, vol 36, pp 349-358, Mar. 1988.

[12]: "TMS320C25 User's Guide, Preliminary", Digital Signal Processor Products, France, Texas Instruments, 1986.

[13]: Anatol I. Zverev, "Handbook of Filter Synthesis". New York, John Wiley, 1967.

[14]: D Dyer and R.M. Stewart, "Detection of Rolling Element Bearing Damage by Statistical Vibration Analysis". Transactions of the ASME, Journal of Mechanical Design, April 1978, Vol 100, ppg 229-235.

[15]: Philip M. Morse, "Vibration and Sound". Published by the American Institute of Physics for the Acoustical Society of America, 1976.

[16]: L.G. Underhill "Introstat", 4th edition, Juta and Co. Ltd, Cape Town, 1985.

LIST OF APPENDICES

APPENDIX A

DESCRIPTIONS OF EACH OF THE TEN ALGORITHMS TESTED

Except where stated, the notation for each algorithm is based on that used by [5]. Each line of the algorithm is numbered so that the listings of each program in Appendix B may be followed more easily. In each algorithm, 'Complexity' refers to the number of multiplications needed for each step.

A.1 THE LMS ALGORITHM

This is the original algorithm formulated by Widrow and Hoff in 1960. It is widely used today because of its simplicity and stability, despite its relatively slow convergence and mediocre accuracy.

COMPUTATION	COMPLEXITY
1] $\epsilon_N^P(T) = d(T - N/2) - W_{N,T-1}Y_N(T)$	N
2] $W_{N,T} = W_{N,T-1} + 2\mu\epsilon_N^P(T)Y_N(T)$	N+1
Total:	2N+1

A.2 SMOOTHED-LMS ALGORITHM

This is the same as the ordinary LMS algorithm but the error is smoothed before it is used to update the filter weights.

	COMPUTATION	COMPLEXITY
1]	$\epsilon^{P}_{N}(T) = d(T - N/2) - W_{N,T-1}Y_N(T)$	N
2]	$\phi(T) = \epsilon^{P}_{N}(T) + \epsilon^{P}_{N}(T-1)$	1
3]	$W_{N,T} = W_{N,T-1} + \mu\phi(T)Y_N(T)$	N+1
Total:		2N+2

A.3 SHARF ALGORITHM

This method was proposed by [10] and is an IIR version of the smoothed-LMS method.

	COMPUTATION	COMPLEXITY
1]	$q_{N+M}(T) = A_{N,T-1}Y_N(T) + B_{M,T-1}Q_M(T-1)$	M+N
2]	$\epsilon^{P}_{N+M}(T) = d(T - N/2) - q_{N+M}(T)$	1
3]	$\phi(T) = \epsilon^{P}_{N+M}(T) + \epsilon^{P}_{N+M}(T-1)$	1
4]	$A_{N,T} = A_{N,T-1} + \mu\phi(T)Y_N(T)$	N+1
5]	$B_{M,T} = B_{M,T-1} + \mu\phi(T)Q_M(T-1)$	M
Total:		2(M+N)+3

A.4 THE A POSTERIORI FAST-RLS ALGORITHM [4]

This algorithm was derived by algebraic methods and so it is different in its properties from the RLS algorithms derived geometrically.

	COMPUTATION	COMPLEXITY
1]	$e^{P}_{N}(T) = A_{N,T-1}Y_{N+1}(T)$	N
2]	$e_N(T) = e^{P}_{N}(T) \;/\; \tau_N(T-1)$	1
3]	$A_{N,T} = A_{N,T-1} + e_N(T)\,[0 \;\; C_{N,T-1}]$	N
4]	$\alpha_N(T) = \alpha_N(T-1) + e^{P}_{N}(T)e_N(T)$	1

Continued Overleaf...

5] $C_{N+1,T} = [0 \quad C_{N,T-1}] - e_N^P(T)\alpha_N^{-1}(T-1)A_{N,T-1}$	N+1
6] $r_N^P(T) = -C_{N+1,T}^N\beta_N(T-1)$	1
7] $[C_{N,T} \quad 0] = C_{N+1,T} - C_{N+1,T}^N B_{N,T-1}$	N
8] $\tau_{N+1}(T) = \tau_N(T-1) + e_N^P(T)\alpha_N^{-1}(T-1)e_N^P(T)$	1
9] $\tau_N(T) = \tau_{N+1}(T) + C_{N+1,T}^N r_N^P(T)$	1
10] $r_N(T) = r_N^P(T) \;/\; \tau_N(T)$	1
11] $\beta_N(T) = \beta_N(T-1) + r_N^P(T)r_N(T)$	1
12] $B_{N,T} = B_{N,T-1} + r_N(T)[C_{N,T} \quad 0]$	N
Weight update:	
13] $\epsilon_N^P(T) = d(T) + W_{N,T-1}Y_N(T)$	N
14] $\epsilon_N(T) = \epsilon_N^P(T) \;/\; \tau_N(T)$	1
15] $W_{N,T} = W_{N,T-1} + \epsilon_N^P(T)C_{N,T}$	N
Total:	7N+9

A.5 THE ORDER N^2 RLS ALGORITHM OF [2].

This method, although unsuitable for real-time applications due to the high number of multiplications necessary, was included for comparison with the other RLS algorithms. The notation is not the same as used in [5], but is the same as that used by [2]. 'Q' is the autocorrelation matrix estimate and 'X' is the reference signal.

COMPUTATION	COMPLEXITY
1] $S = Q_{k-1}^{-1}X_k$	N^2
2] $\tau = \alpha + X_k S^T$	N
3] $Q_k^{-1} = (Q_{k-1}^{-1} - SS^T \;/\; \tau) \;/\; \alpha$	$3N^2$
4] $W_{k+1} = W_k + Q_k^{-1}\epsilon_k X_k$	N^2
Total:	$5N^2+N$

A.6 THE ROBUST RLS ALGORITHM OF [5].

This is slightly different from the algorithm described in Chapter 2 of this thesis due to changes made to improve the stability.

	COMPUTATION	COMPLEXITY
1]	$e_N^P(T) = A_{N,T-1}Y_{N+1}(T)$	N
2]	$e_N(T) = e_N^P(T)\tau_N(T-1)$	1
3]	$\alpha_N(T) = \alpha_N(T-1) + e_N^P(T)e_N(T)$	2
4]	$\tau_{N+1}(T) = \tau_N(T-1)\alpha_N(T-1)/\alpha_N(T)$	3
5]	$A_{N,T} = A_{N,T-1} + e_N^P(T)[0 \quad C_{N,T-1}]$	N
6]	$C_{N+1,T} = [0 \quad C_{N,T-1}] - e_N(T)\alpha_N^{-1}(T)A_{N,T}$	N+1
7]	$r_N^P(T) = -\beta_N(T-1)\tau_{N+1}^{-1}(T)C_{N+1,T}^N$	3
8]	$\tau_N(T) = [1 + r_N^P(T)C_{N+1,T}^N]^{-1}\tau_{N+1}(T)$	3
9]	$r_N(T) = r_N^P(T)\tau_N(T)$	1
10]	$\beta_N(T) = \beta_N(T-1) + r_N^P(T)r_N(T)$	1
11]	$[C_{N,T} \quad 0] = \{1 + C_{N+1,T}^N r_N^P(T)\}^{-1}\{C_{N+1,T} - C_{N+1,T}^N B_{N,T-1}\}$	2N
12]	$B_{N,T} = B_{N,T-1} + r_N^P(T)[C_{N,T}]$	N
	Weight update:	
13]	$\epsilon_N^P(T) = d(T) + W_{N,T-1}Y_N(T)$	N
14]	$W_{N,T} = W_{N,T-1} + \epsilon_N(T)C_{N,T}$	N
	Total:	8N+14

A.7 THE IIR FAST-RLS ALGORITHM OF [11].

The algorithm has no forgetting factor, since in adaptive system modelling the system is not expected to alter. Unlike the other

RLS algorithms, the predictions and residuals are not scalars but vectors containing 2 elements each.

COMPUTATION	COMPLEXITY
1] $e^{P}_{N}(T) = A_{N,T-1}\Phi_{f}\underline{z}(T)$	2N
2] $e_{N}(T) = e^{P}_{N}(T)\tau_{N}(T-1)$	2
3] $\alpha_{N}(T) = \alpha_{N}(T-1) + e^{P}_{N}(T)e'_{N}(T)$	3
4] $\alpha^{-1}_{N}(T)$ 2 x 2 matrix inversion	5
5] $\tau_{N+2}(T) = \tau_{N}(T-1) - e'_{N}(T)\alpha^{-1}_{N}(T)e_{N}(T)$	6
6] $\zeta_{N+2,T} = [0 \quad 0 \quad C_{N,T-1}] - e'_{N}(T)\alpha^{-1}_{N}(T)\tau^{-1}_{N+2}(T)A_{N,T-1}$	2N+2
7] $C_{N+2,T} = \zeta_{N+2,T}\Phi_{f}\Phi'_{b}$	0
8] $\mu(T)$ = last two elements of $C_{N+2,T}$	0
9] $A_{N,T} = A_{N,T-1} + e_{N}(T)[0 \quad 0 \quad C_{N,T-1}]$	2N
10] $r^{P}_{N}(T) = -\mu(T)\beta_{N}(T-1)$	4
11] $\tau_{N}(T) = [1 + \mu(T)r^{P}_{N}(T)\tau_{N+2}(T)]^{-1}\tau_{N+2}(T)$	4
12] $r_{N}(T) = r^{P}_{N}(T)\tau_{N}(T)$	2
13] $\beta_{N}(T) = \beta_{N}(T-1) + r^{P}_{N}(T)r'_{N}(T)$	3
14] $[C_{N,T} \quad 0 \quad 0] = C_{N+2,T} - \mu(T)B_{N,T-1}$	2N
15] $B_{N,T} = B_{N,T-1} + r_{N}(T)[C_{N,T} \quad 0 \quad 0]$	2N
Weight update:	
16] $\epsilon^{P}_{N}(T) = d(T) + W_{N,T-1}z(T)$	N
17] $\epsilon_{N}(T) = \epsilon^{P}_{N}(T)\tau_{N}(T)$	1
18] $W_{N,T} = W_{N,T-1} + \epsilon_{N}(T)C_{N,T}$	N
Total:	12N+32

A.8 THE EXACT INITIALISATION ALGORITHM OF [5].

COMPUTATION	COMPLEXITY
$T = 0;\quad A_{0,0} = B_{0,0} = 1;\quad C_{0,0} = 0;$	
$W_{1,0} = -d(0)/y(0);\quad \tau_0(0) = 1;\quad \alpha_0(0) = y(0)^2.$	
$1 \le T \le N$	
1] $e^P_{T-1}(T) = A_{T-1,T-1}[y(T),\ldots,y(1)]'$	N
2] $A^T_{T,T} = \begin{bmatrix} A_{T-1,T-1} & -e^P_{T-1}(T)/y(0) \end{bmatrix}$	1
3] $e'_{T-1}(T) = e^P_{T-1}(T)\tau_{T-1}(T-1)$	1
4] $\alpha_T(T) = \alpha_{T-1}(T-1)$	1
5] $\alpha_{T-1}(T) = \alpha_T(T) + e^P_{T-1}(T)e_{T-1}(T)$	1
6] $\tau_T(T) = \tau_{T-1}(T-1)\ \alpha_T(T)\ /\ \alpha_{T-1}(T)$	3
7] $C_{T,T} = [0 \quad C_{T-1,T-1}] - e^P_{T-1}(T)A_{T-1,T-1}\ /\ \alpha_T(T)$	N+1
When T=N:	
8] $B_{T,T} = [(y(0)\tau_T(T)C_{T,T}) \quad 1]$	
9] $\beta_{T,T} = y(0)^2\tau_T(T)$	
Weight update:	
10] $\epsilon^P_T(T) = d(T) + W_{T,T}Y_T(T)$	N
11] $\epsilon_T(T) = \epsilon^P_T(T)\tau_T(T)$	1
If T<N,	
12] $W_{T+1,T} = \begin{bmatrix} W_{T,T-1} & -\epsilon^P_T(T)/y(0) \end{bmatrix}$	1
If T=N,	
13] $W_{N,T} = W_{T,T-1} + \epsilon_T(T)C_{T,T}$	
Average total: about 1.5N	3N+10

A.9 THE MITAL III ALGORITHM OF [8].

The notation followed here is that of [8], since there is no equivalent in [5]. The MITAL III algorithm described here is the third in a series of MITAL algorithms derived in [8]. The transversal filter version of the MITAL III algorithm is the only MITAL method that can use the property of not having to update the filter at every time step. The filter can be updated when desired: only the auto- and cross-correlation functions need to be calculated at every time interval.

COMPUTATION	COMPLEXITY

For every time instant T, iterate:

$$\bar{G}_{M+1,T} = [\ \overset{f}{\epsilon}_{0,T} \mid G_{M,T}\ ] = \delta\bar{G}_{M+1,T-1} + y_T\overset{t}{Y}_{M+1}(T)$$

and

$$Q_{M,T} = \delta Q_{M,T-1} + d_T\overset{t}{Y}_M(T) \qquad 2M+2$$

For order zero, let $f_{0,T} = b_{0,T} = y_T$,

$$F_{0,T} = B_{0,T} = 1,\ \overset{b}{\epsilon}_{0,T} = \overset{f}{\epsilon}_{0,T}.$$

For m = 0, 1,..., M-1:

1] $b_{m,T} = \overset{t}{Y}_{m+1}(T)B_{m,T}$ $(M^2+M)/2$

2] $p_{m,T} = Q_{m+1,T}B_{m,T}$ $(M^2+M)/2$

3] $\overset{e}{k}_{m,T} = p_{m,T}\overset{-b}{\epsilon}_{m,T}$ M

4] $B_{m,T-1} = B_{m,T} - [\ C_{m,T}\ \ 0\ ]\ \overset{-b}{\tau}_{m,T}b_{m,T}$ $(M^2+M)/2$

5] $\overset{b}{\epsilon}_{m,T-1} = (\ \overset{b}{\epsilon}_{m,T} - b_{m,T}\overset{-b}{\epsilon}_{m,T}b_{m,T}\ )\ /\ \delta$ 2M

6] $C_{m+1,T} = [\ C_{m,T}\ \ 0\ ] - B_{m,T}\overset{-b}{\epsilon}_{m,T}b_{m,T}$ $(M^2+M)/2$

7] $\tau_{m+1,T} = \tau_{m,T} - b_{m,T}\overset{-b}{\epsilon}_{m,T}b_{m,T}$ M

8] $k_{m,T} = G_{m+1,T}B_{m,T-1}$ $(M^2+M)/2$

9] $\overset{f}{k}_{m,T} = k_{m,T}\overset{-b}{\epsilon}_{m,T-1}$ M

10] $\overset{b}{k}_{m,T} = k_{m,T}\overset{-f}{\epsilon}_{m,T}$ M

Continued Overleaf...

11] $F_{m+1,T} = [\ F_{m,T} \quad 0\ ] - [\ 0 \quad B_{m,T-1}\ ]\, k^{f}_{m,T}$ $(M^2-3M+6)/2$

12] $B_{m+1,T} = [\ 0 \quad B_{m,T-1}\ ] - [\ F_{m,T} \quad 0\ ]\, k^{b}_{m,T}$ $(M^2-3M+6)/2$

13] $\epsilon^{f}_{m+1,T} = \epsilon^{f}_{m,T} - k^{f}_{m,T} k^{f}_{m,T}$ M

14] $\epsilon^{b}_{m+1,T} = \epsilon^{b}_{m,T-1} - k^{b}_{m,T} k^{b}_{m,T}$ M

15] $W_{m+1,T} = [\ W_{m,T} \quad 0\ ] - B_{m,T} k_{m,T}$ $(M^2-3M+6)/2$

Total: $4M^2+(19/2)M+11$

A.10 THE STABILIZED RLS ALGORITHM OF [9].

This algorithm is based on the RLS method found in [5] in Table III.

COMPUTATION	COMPLEXITY
1] $e^{P}_{N} = x(T) - A'_{N,T-1} X_{N,T-1}$	N
2] $r^{P}_{N} = x(T-N) - B'_{N,T-1} X_{N,T}$	N
3] $\tau_{N+1}(T) = \delta\alpha_N(T-1)\ \tau_N(T-1) \ /\ \{\ \delta\alpha_N(T-1) + \tau_N(T-1)(e^{P}_{N}(T))^2\ \}$	4
4] $\tau_N(T) = \{1 + \tau_{N+1}(T) r^{P}_{N}(T) (C^{N}_{N,T+1} + e^{P}_{N}(T) A_{N,T-1}/\delta\alpha_N(T-1))\}^{-1} \tau_{N+1}(T)$	5
5] $k_N(T-1) = \delta^{-N} \tau_N(T-1)$	1
6] $\Phi_N(T) = r^{P}_{N}(T) + A^{N}_{N,T-1} k_{N,T-1} e^{P}_{N}(T) + \delta\beta_N(T-1) C^{N}_{N,T-1}$	3
7] $\underline{\Phi}_N(T) = \{1 + \sigma(\tau^{-1}_{N}(T)-1) + \sigma(A_{N,T-1} k_N(T-1))^2 (\tau^{-1}_{N}(T-1)-1) + 2\sigma A^{N}_{N,T-1} k^{2}_{N}(T-1) C^{N}_{N,T-1} e^{P}_{N}(T)\}^{-1}\ \Phi_N(T)$	13

Correction of the transversal filter $B_{N,T-1}$

8] $\underline{e}^{P}_{N}(T) = (1-\sigma k_N(T-1) C^{N}_{N,T-1} \underline{\Phi}_N(T)) e^{P}_{N}(T) - \sigma\delta^{-N} A_{N,T-1} (1-\tau_N(T-1)) \underline{\Phi}_N(T)$	7
9] $C_{N+1,T} = [0 \quad C_{N,T-1}] - \underline{e}^{P}_{N}(T)\ [1 \quad -A_{N,T-1}]/\delta\alpha_N(T-1)$	N+1

Continued Overleaf...

$$10]\ [\underline{B}_{N,T-1}\ \ -1] = \{[B_{N,T-1}\ \ -1] - \sigma\underline{\Phi}_N(T)C_{N+1,T}\} \ /\ \{1 + \sigma C^{N+1}_{N+1,T}\underline{\Phi}_N(T)\} \qquad 2N+1$$

$$11]\ \overset{P}{\underline{r}}_N(T) = \overset{P}{r}_N(T) - \sigma(\tau_N^{-1}(T) - 1)\underline{\Phi}_N(T) \qquad 2$$

Classical RLS algorithm

$$12]\ e_N(T) = \tau_N(T-1)\overset{P}{\underline{e}}_N(T) \qquad 1$$

$$13]\ \alpha_N(T) = \delta\alpha_N(T-1) + e_N(T)\overset{P}{\underline{e}}_N(T) \qquad 1$$

$$14]\ A_{N,T} = A_{N,T-1} - e_N(T)C_{N,T-1} \qquad N$$

$$15]\ r_N(T) = \tau_N(T)\underline{r}_N(T) \qquad 1$$

$$16]\ \beta_N(T) = \delta\beta_N(T-1) + r_N(T)\overset{P}{\underline{r}}_N(T) \qquad 1$$

$$17]\ [C_{N,T}\ \ 0] = C_{N+1,T} - C^{N+1}_{N+1,T}[-B_{N,T-1}\ \ 1] \qquad N$$

$$18]\ B_{N,T} = \underline{B}_{N,T-1} - r_N(T)C_{N,T} \qquad N$$

Filtering of the Reference Signal y(T)

$$19]\ \overset{P}{\epsilon}_N(T) = y(T) - W'_{N,T-1}X_{N,T} \qquad N$$

$$20]\ \epsilon_N(T) = \tau_N(T)\overset{P}{\epsilon}_N(T) \qquad 1$$

$$21]\ W_{N,T} = W_{N,T-1} - \epsilon_N(T)C_{N,T} \qquad N$$

Total: $10N+40$

APPENDIX B

LISTINGS OF THE ADAPTIVE-FILTER SUBROUTINES USED ON THE HP9836 AND IBM PS/2.

These listings are merely the formulae in Appendix A translated into program languages, either TURBO C or HP-BASIC 5.0 and do not include signal-generating or display subroutines. The line numbers on the right-hand side of these listings correspond to those in Appendix A to aid understanding of the programs.

B.1 LMS ALGORITHM (LISTING IN TURBO C).

```
void lmsfilter(int t, int LEN)
{
    out[t] = 0.0;
    for (j=1; j<=LEN; j++)
        out[t] = out[t] + weight[j] * ref[t-j];

    err[t] = mu * (prim[t-LEN/2] - out[t]);          /* Line 1 */
    for (j=1; j<=LEN; j++)
        weight[j] = weight[j] + err[t]*ref[t-j];     /* Line 2 */
}
/*****************************************************************/
```

B.2 SMOOTHED-LMS ALGORITHM (LISTING IN TURBO C).

```
void smoothfilter(int t, int LEN)
{
    out[t] = 0.0;
    for (j=1; j<=LEN; j++)
        out[t] = out[t] + weight[j] * ref[t-j];

        err[t] = prim[t-LEN/2] - out[t];             /* Line 1 */
        phi = err[t] + err[t-1];                     /* Line 2 */
    for (j=1; j<=LEN; j++)
        weight[j] = weight[j] + mu*phi*ref[t-j];     /* Line 3 */
}
/*****************************************************************/
```

B.3 SHARF ALGORITHM

```
4450 SUB Sharf(W(*),Ref(*),Prim(*),E(*),Y(*),U,U1,L,T)
4460     FOR J=1 TO L                              !\
4470         Y(T)=Y(T)+W(J)*Ref(T-J+1)             ! \
4480     NEXT J                                    !  \ Line 1.
4490     FOR J=1 TO L-1                            !  /
4500         Y(T)=Y(T)+W(J+L)*Y(T-J)               ! /
4510     NEXT J                                    !/
4520     !
4530     E(T)=Prim(T-L/2)-Y(T)                     ! Line 2.
4540     Phi=E(T)+E(T-1)                           !\
4550     Phi1=Phi*2*U                              ! > Line 3.
4560     Phi2=Phi*2*U1                             !/
4570     !
4580     FOR J=1 TO L
4590         W(J)=W(J)+Phi1*Ref(T-J+1)             ! Line 4.
4600     NEXT J
4610     FOR J=1 TO L-1
4620         W(J+L)=W(J+L)+Phi2*Y(T-J)             ! Line 5.
4630     NEXT J
4640 SUBEND

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

B.4 THE 'A POSTERIORI' RLS ALGORITHM OF [4]

This program can be found in FORTRAN in the paper of [4]. Here it is written in HP BASIC.

```
4450 SUB Faest(X(*),W(*),A(*),B(*),Xin,Alpha,Psi,Ab,Af,
INTEGER Ip)
4460     DIM We(40)
4470     Rl=.99
4480     Ip1=Ip+1
4490     Ef=Xin                                    !\
4500     FOR J=1 TO Ip                             ! \ Line 1.
4510         Ef=Ef+A(J)*X(J)                       ! /
4520     NEXT J                                    !/
4530     !
4540     Help=Ef/Rl/Af
4550     We(1)=-Help                               !\
4560     FOR J=1 TO Ip                             ! \ Line 5.
4570         We(J+1)=W(J)-Help*A(J)                ! /
4580     NEXT J                                    !/
4590     !
4600     Efg=Ef*Psi                                ! Line 2.
4610     FOR J=1 TO Ip
4620         A(J)=A(J)+W(J)*Efg                    ! Line 3.
4630     NEXT J
4640     !
4650     FOR J=1 TO Ip                             !\
4660         W(J)=We(J)-We(Ip1)*B(J)               ! > Line 7.
4670     NEXT J                                    !/
4680     !
4690     Eb=-Rl*Ab*We(Ip1)                         ! Line 6.
```

```
4700      Af=Rl*Af+Ef*Efg                          ! Line 4.
4710      Alpha=Alpha+We(Ip1)*Eb+Help*Ef           ! Line 8.
4720      Psi=1/Alpha                              ! Line 9.
4730      Ebg=Eb*Psi                               ! Line 10.
4740      Ab=Rl*Ab+Eb*Ebg                          ! Line 11.
4750      FOR J=1 TO Ip                            !\
4760          B(J)=B(J)+W(J)*Ebg                   ! > Line 12
4770      NEXT J                                   !/
4780      !
4790      Ipm1=Ip-1
4800      FOR J=1 TO Ipm1                          ! Lines 13 to 15 are
4810          X(Ip1-J)=X(Ip-J)                     ! not included.
4820      NEXT J
4830      X(1)=Xin
4840  SUBEND

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

B.5 THE ORDER N-SQUARED METHOD OF [2] (LISTING IN BASIC).

```
2210 SUB Rls(Filtlen,T,S(*),Ref(*),Q_inv(*),Out(*),Prim(*),
Weight(*),Err(*))
2220      ALLOCATE Tapref(Filtlen),Sst(Filtlen,Filtlen),
Q_inv_x(Filtlen)
2230      !
2240      FOR J=1 TO Filtlen                       !\
2250          Tapref(J)=Ref(T-J+1)                 ! \
2260      NEXT J                                   !  \
2270      !                                        !   \
2280      MAT S= (0)                               !    \ Line 1.
2290      FOR J=1 TO Filtlen                       !    /
2300          FOR K=1 TO Filtlen                   !   /
2310              S(J)=S(J)+Tapref(K)*Q_inv(J,K)   !  /
2320          NEXT K                               ! /
2330      NEXT J                                   !/
2340      !
2350      Xts=0                                    !\
2360      FOR J=1 TO Filtlen                       ! \
2370          Xts=Xts+Tapref(J)*S(J)               !  > Line 2.
2380      NEXT J                                   ! /
2390      Gamma=Alpha+Xts                          !/
2400      !
2410      FOR J=1 TO Filtlen                       !\
2420          FOR K=1 TO Filtlen                   ! \
2430              Sst(J,K)=S(J)*S(K)/Gamma         !  \
2440          NEXT K                               !   \
2450      NEXT J                                   !    \
2460      !                                        !     \
2470      MAT Q_inv= Q_inv-Sst                     !      \
2480      MAT Q_inv= Q_inv/(Alpha)                 !       > Line 3.
2490      !                                        !      /
2500      MAT Q_inv_x= (0)                         !     /
2510      FOR J=1 TO Filtlen                       !    /
2520          FOR K=1 TO Filtlen                   !   /
2530              Q_inv_x(J)=Q_inv_x(J)+Q_inv(J,K)*Tapref(K)
2540          NEXT K                               ! /
```

```
2550      NEXT J                                          !/
2560      !
2570      FOR J=1 TO Filtlen                              !\
2580          Out(T)=Out(T)+Weight(J)*Tapref(J)           !  \
2590      NEXT J                                          !   \
2600      !                                               !    \
2610      Err(T)=Prim(T-Filtlen/2)-Out(T)                 !     >Line 4.
2620      !                                               !    /
2630      FOR J=1 TO Filtlen                              !   /
2640          Weight(J)=Weight(J)+Err(T)*Q_inv_x(J)       !  /
2650      NEXT J                                          !/
2660 SUBEND
/*************************************************************/
```

B.6 THE ROBUST RLS ALGORITHM OF [5] (LISTING IN TURBO C).

```
void fast(float *psi, float *af, float *ab, float *alpha,
          float *rescue, float xin, float rl)
{
    int i;
    float we[17], len1, efg, af1, help;
    float ef, eb, ebg, temp;

    len1 = LEN + 1;
    ef = xin;                                   /*           */
    for (i=1; i<=LEN; i++)                      /* Line 1.   */
        ef += f_pred[i] * filtsig[i];           /*           */

    efg = ef * *psi;                            /* Line 2.   */
    af1 = *af*rl + ef*efg;                      /* Line 3.   */
    *alpha = *psi*rl * *af/af1;                 /* Line 4.   */

    for (i=1; i<=LEN; i++)                      /* Line 5.   */
        f_pred[i] += kal[i]*ef;                 /*           */
    help = efg/af1;
    we[1] = -help;
    for (i=1; i<=LEN; i++)                      /* Line 6.   */
        we[i+1] = kal[i] - help*f_pred[i];      /*           */

    eb = -rl* *ab * we[len1]/ *alpha;           /* Line 7.   */
    temp = eb * we[len1] + 1;
    *psi = 1 / temp * *alpha;                   /* Line 8.   */
    ebg = eb * *psi;                            /* Line 9.   */
    *ab = rl* *ab + eb*ebg;                     /* Line 10.  */

    *rescue = 1 - we[len1]*we[len1] * ef*ef;
    *af = af1;
    for (i=1 ; i<=LEN; i++)                     /* Line 11.  */
       kal[i] = (we[i] - we[len1]*b_pred[i]) / temp;

    for (i=1; i<=LEN; i++)
        b_pred[i] += kal[i]*eb;                 /* Line 12.  */
    for (i=1; i<=LEN-1; i++)
        filtsig[len1-i] = filtsig[LEN-i];
    filtsig[1] = xin;
}/*************************************************************/
```

B.7 THE IIR FAST-RLS ALGORITHM OF [11] (LISTING IN TURBO C).

```
void ardalan (float *psi, float xin, float yin)
/*
    This routine is the fast-RLS algorithm of Ardalan and Faber
    (IEEE Trans. Acoust., Speech and Sig. Proc. March 1988).
*/
{
    int i, j, tlen2;
    float *we, efg[2], help[2], alpha, det;
    float ef[2], eb[2], ebg[2], temp;
    float aflip[2][2], lmb = 0.98;

    tlen2 = tfilt+2;
    we = Calloc (tlen2, float);

/**** Obtain the Forward-Prediction Residual ****/

    ef[0] = xin;
    ef[1] = yin;

    for (i=0; i<tfilt; i++)
        for (j=0; j<2; j++)
            ef[j] += (f_pred[j][i] * filtsig[i]);

/**** Obtain the Forward Filtered Residual ****/

     for (i=0; i<2; i++)
        efg[i] = ef[i] * *psi;

/**** Update the Squared-length of the
      Forward residual error vector ****/
    af[0][0] = af[0][0]*lmb + (ef[0]*efg[0]);
    af[1][1] = af[1][1]*lmb + (ef[1]*efg[1]);
    af[0][1] = af[0][1]*lmb + (ef[0]*efg[1]);
    af[1][0] = af[0][1];

/**** Find the inverse of the previous matrix ****/

    det = 1.0/(af[0][0] * af[1][1] - af[1][0] * af[0][1]);
    aflip[0][0] = af[1][1] * det;
    aflip[1][1] = af[0][0] * det;
    aflip[0][1] = aflip[1][0] = af[0][1] * -det;

/**** Find the squared-length of the
      Pinning-vector residual error ****/
    alpha = *psi*lmb;
    for (i=0; i<2; i++)
    {
        help[i] = 0.0;
        for (j=0; j<2; j++)
            help[i] += (efg[j] * aflip[i][j]);
        alpha -= (help[i] * efg[i])*lmb;
    }
```

```
/**** Update the Kalman-gain vector ****/

    for (i=0; i<2; i++)
    {
        help[i] /= alpha;
        we[i] = -help[i]/lmb;
    }
    for (i=0; i<tfilt; i++)
        we[i+2] = kal[i] - (help[0] * f_pred[0][i]
                         +  help[1] * f_pred[1][i])/lmb;

/**** Permute Forwards ****/

    temp = we[1];
    for (i=1; i<filt; i++)  we[i] = we[i+1];
    we[filt] = temp;

/**** Permute Backwards ****/

    temp = we[filt+1];
    for (i=filt+1; i<tfilt; i++)  we[i] = we[i+1];
    we[tfilt] = temp;

/**** Update the Forward-prediction filter ****/

    for (i=0; i<tfilt; i++)
        for (j=0; j<2; j++)
            f_pred[j][i] += (efg[j] * kal[i]);

    for (i=0; i<2; i++)
        eb[i] = -lmb*we[tfilt] * ab[0][i] - lmb*we[tfilt+1]
                  * ab[1][i];
    *psi = alpha / (1.0 + (we[tfilt]*eb[0] + we[tfilt+1]*eb[1])
                     * alpha);
    for (i=0; i<2; i++)
        ebg[i] = eb[i] * *psi;

    ab[0][0] = ab[0][0]*lmb + (eb[0]*ebg[0]);
    ab[1][1] = ab[1][1]*lmb + (eb[1]*ebg[1]);
    ab[0][1] = ab[0][1]*lmb + (eb[0]*ebg[1]);
    ab[1][0] = ab[0][1];

    for (i=0; i<tfilt; i++)
    {
        kal[i] = we[i];
        for (j=0; j<2; j++)
            kal[i] -= we[tfilt+j] * b_pred[j][i];
    }

    for (i=0; i<tfilt; i++)
        for (j=0; j<2; j++)
            b_pred[j][i] += (kal[i]*ebg[j]);

    for (i=filt-1; i>=1; i--)
        filtsig[i] = filtsig[i-1];
    for (i=tfilt-1; i>=filt+1; i--)
        filtsig[i] = filtsig[i-1];
```

```
    filtsig[0] = xin;
    filtsig[filt] = yin;

    free (we);
}
/* end 'ardalan' ***************************************************/
```

B.8 THE EXACT INITIALISATION PROCEDURE OF [5].

```
void init(float *psi, float *af, float *ab, float rl, int ti)
{
    int i, t;
    float ef, efg, af1, errpost;

    for (i=1; i<=LEN; i++)
    {
        kal[i] = 0.0;
        f_pred[i] = 0.0;
        b_pred[i] = 0.0;
        weight[i] = 0.0;
    }
    weight[1] = -prim[1+ti] / ref[1+ti];
    *psi = 1.0;
    *af = ref[1+ti] * ref[1+ti];
    for (t=2; t<=LEN+1; t++)
    {
        ef = ref[t+ti];                              /* Line 1.  */
        for (i=1; i<=t-2; i++)
            ef += f_pred[i] * ref[t-i+ti];
        f_pred[t-1] = -ef / ref[1+ti];               /* Line 2.  */

        efg = ef * *psi;                             /* Line 3.  */
        *af *= rl;                                   /* Line 4.  */
        af1 = *af + ef*efg;                          /* Line 5.  */
        *psi *= *af/af1;                             /* Line 6.  */
        for (i=2; i<=t-1; i++)                       /* Line 7.  */
            kal[i+LEN-t+1] -= f_pred[i-1] * ef/ *af;

        kal[2+LEN-t] = -ef/ *af;
        if (t == LEN+1)
        {
            for (i=1; i<=LEN; i++)                   /* Line 8.  */
                b_pred[i] = kal[i] * ref[1+ti]* *psi;
            *ab = ref[1+ti]*ref[1+ti] * *psi;        /* Line 9. */
        }
        err[t+ti] = prim[t+ti];                      /* Line 10. */
        for (i=1; i<=t-1; i++)
            err[t+ti] += weight[i] * ref[t-i+1+ti];

        errpost = err[t+ti] * *psi;                  /* Line 11. */
        if (t == LEN+1 )                             /* Line 12. */
           for (i=1; i<=LEN; i++)  weight[i] += errpost * kal[i];
        else
        weight[t] = -err[t+ti] / ref[1+ti];          /* Line 13. */
        out[t+ti] = prim[t+ti] - err[t+ti];
    }
```

```
    for (i=1; i<=LEN; i++)  kal[i] *= *psi;
}
/*****************************************************************/
```

B.9 THE MITAL 3 ALGORITHM OF [8].

```
void crossmult (float lmb, float din)
/*
    Correlate the two input signals of the MITAL III algorithm.
*/
{
    int i;

    for (i=0; i<=tfilt; i++)
    {
        q[i] = q[i] * lmb + din * filtsig[i];
        g[i] = g[i] * lmb + filtsig[0] * filtsig[i];
    }
}
/* end 'crossmult' ***********************************************/


void mitalinit (float ref, float prim, float lmb, int ti)
/*
    This routine initializes the MITAL III filter.
*/
{
    int i, t;

    if (ti == 0)
    {
        weight  = (float *) calloc (tfilt, sizeof (float));
        q       = (float *) calloc (tfilt+1, sizeof (float));
        g       = (float *) calloc (tfilt+1, sizeof (float));
        filtsig = (float *) calloc (tfilt+1, sizeof (float));

        for (i=0; i<=tfilt; i++)
            q[i] = g[i] = filtsig[i] = 0.0;
        times = 1;
    }

    if (ti < tfilt) t = ti;
    else t = tfilt;

    for (i=t; i>0; i--)
        filtsig[i] = filtsig[i-1];
    filtsig[0] = ref;
    crossmult (lmb, prim);
}
/* end 'mitalinit ************************************************/
```

```
int stable3 (float din, float lmb)
/*
    This routine is the MITAL III algorithm of Yu and He.
*/
{
    int i, m, m1;
    float epsf, pm, kem, km, kfm, kbm, tau;
    float eb, btau, bm;
    float *epsb, *epsb1, *cm;
    float *a, *a1;
    float *b, *b1;

    crossmult (lmb, din);

    if (times != 1) return (0);

    b     = (float *) calloc ((tfilt+1)*(tfilt+1), sizeof (float));
    b1    = (float *) calloc ((tfilt+1)*(tfilt+1), sizeof (float));
    a     = (float *) calloc (tfilt+1, sizeof (float));
    a1    = (float *) calloc (tfilt+1, sizeof (float));
    epsb  = (float *) calloc (tfilt+1, sizeof (float));
    epsb1 = (float *) calloc (tfilt+1, sizeof (float));
    cm    = (float *) calloc (tfilt+1, sizeof (float));

    epsf = epsb[0] = g[0];
    a[0] = a1[0] = b[0] = tau = 1.0;

    for (m=0; m<tfilt; m++)
    {
        bm = 0.0;
        for (i=0; i<=m; i++)
            bm += filtsig[i] * b[m*tfilt+i];          /* Line 1  */

        m1 = m+1;
        pm = 0.0; km = 0.0;
        for (i=0; i<=m; i++)
            pm += (q[i] * b[m*tfilt+i]);              /* Line 2  */
        kem = pm / epsb[m];                           /* Line 3  */

        btau = bm/tau;
        for (i=0; i<m; i++)
            b1[m*tfilt+i] = b[m*tfilt+i] - cm[i] * btau;
        b1[m*tfilt+m] = 1.0;                          /* Line 4  */
        epsb1[m] = (epsb[m] - bm*btau) / lmb;         /* Line 5  */

        eb = bm/epsb[m];
        for (i=0; i<m; i++)
            cm[i] -= (b[m*tfilt+i] * eb);             /* Line 6  */
        cm[m] = -eb;
        tau -= (bm * eb);                             /* Line 7  */

        for (i=0; i<=m; i++)
            km += (g[i+1] * b1[m*tfilt+i]);           /* Line 8  */
        kfm = km / epsb1[m];                          /* Line 9  */
        kbm = km / epsf;                              /* Line 10 */
```

```
        for (i=1; i<=m; i++)
            a1[i] = a[i] - b1[m*tfilt+i-1] * kfm;    /* Line 11 */
        a1[m+1] = -kfm;

        b[m1*tfilt+0] = -kbm;
        for (i=1; i<=m; i++)                          /* Line 12 */
            b[m1*tfilt+i] = b1[m*tfilt+i-1] - a[i] * kbm;
        b[m1*tfilt+m1] = 1.0;

        for (i=0; i<=m+1; i++)
            a[i] = a1[i];

        epsf -= (kfm * km);                           /* Line 13 */
        epsb[m1] = epsb1[m] - kbm * km;               /* Line 14 */

        weight[m] = -kem;
        for (i=0; i<m; i++)
            weight[i] -= (b[m*tfilt+i]*kem);          /* Line 15 */
    }
    free (b); free (b1);
    free (a); free (a1);
    free (epsb); free (epsb1); free (cm);
}
/* end 'stable3' ************************************************/


int mital3 (float ref, float prim, float *output, int t)
/*
    The coordinating routine of the MITAL III algorithm.
*/
{
    int i, skip = 1; /* Only update the weights every */
                     /* "skip" times.                 */
    float lmb = 0.985;

    if (t < 2*tfilt)
    {
        mitalinit (ref, prim, lmb, t);
        return (0);
    }

    for (i=tfilt; i>0; i--)        /* Shift the signal values */
        filtsig[i] = filtsig[i-1]; /* in the filter.          */
    filtsig[0] = ref;

    stable3 (prim, lmb);

    *output = 0.0;
    for (i=0; i<tfilt; i++)
        *output -= weight[i] * filtsig[i];

    ++times;
    if (times > skip) times = 1;
}
/* end 'mital3' *************************************************/
```

B.10 THE EXTRA-STABLE RLS ALGORITHM OF [9].

```
void stable (float *gamm1, float *al, float *beta,
             float *rho, float lmb, float xin)
{
    int i, len = FILT-1;
    float epn, rpn, gamp1, gam, kn, er, erbar, lmb1;
    float epnbar, rpnbar, en, rn;
    float help, help1;
    float we[FILT1];

/*----- COMPUTE "ER" AND "ERBAR" -----*/

    epn = xin;
    rpn = x[len];

    for (i=0; i<FILT; i++)
        epn -= a[i]*x[i];                                  /* Line 1 */
    for (i=FILT-2; i>=0; i--)
        rpn -= b[i+1]*x[i];                                /* Line 2 */
    rpn -= b[0]*xin;

    *al *= lmb;
    *beta *= lmb;
    lmb1 = pow (lmb, -FILT);

    gamp1 = *al * *gamm1 /(*al + *gamm1 * epn * epn);
                                                           /* Line 3 */
    gam = gamp1 / (1.0 + gamp1 * rpn *
                   (c[len] + epn*a[len]/ *al));
                                                           /* Line 4 */
    kn = lmb1 * *gamm1;                                    /* Line 5 */
    er = rpn + a[len] * kn * epn + *beta * c[len];         /* Line 6 */
    erbar = er / (1.0 + *rho * ((1.0/gam-1.0)              /* Line 7 */
          + pow (a[len]*kn, 2) * (1.0/ *gamm1-1.0)
          + 2.0 * a[len] * kn*kn * c[len] * epn));

/*----- CORRECT THE FILTER "B" ----*/

    help = *rho * erbar;
    epnbar = epn * (1.0 - help*kn*c[len])
                 - help*lmb1*a[len]*(1.0- *gamm1);
                                                           /* Line 8 */
    help1 = epnbar / *al;
    we[0] = -help1;
    for (i=0; i<FILT; i++)
        we[i+1] = c[i] + help1 * a[i];                     /* Line 9 */

    help1 = 1.0 / (1.0 + we[FILT] * help);
    for (i=0; i<FILT; i++)
        b[i] = help1 * (b[i] - help * we[i]);              /* Line 10 */

    rpnbar = rpn - help * (1.0/gam - 1.0);                 /* Line 11 */
```

```
/*---- CLASSICAL FTF ALGORITHM ----*/

    en = *gamm1 * epnbar;                                   /* Line 12 */
    *al += en * epnbar;                                     /* Line 13 */
    for (i=0; i<FILT; i++)
        a[i] -= en * c[i];                                  /* Line 14 */

    rn = gam * rpnbar;                                      /* Line 15 */
    *beta += rn * rpnbar;                                   /* Line 16 */
    for (i=0; i<FILT; i++)
        c[i] = we[i] + we[FILT] * b[i];                     /* Line 17 */

    for (i=0; i<FILT; i++)
        b[i] -= rn * c[i];                                  /* Line 18 */

    *gamm1 = gam;

    for (i=1; i<FILT; i++)
        x[FILT-i] = x[len-i];
    x[0] = xin;
}
/*************************************************************************/
```

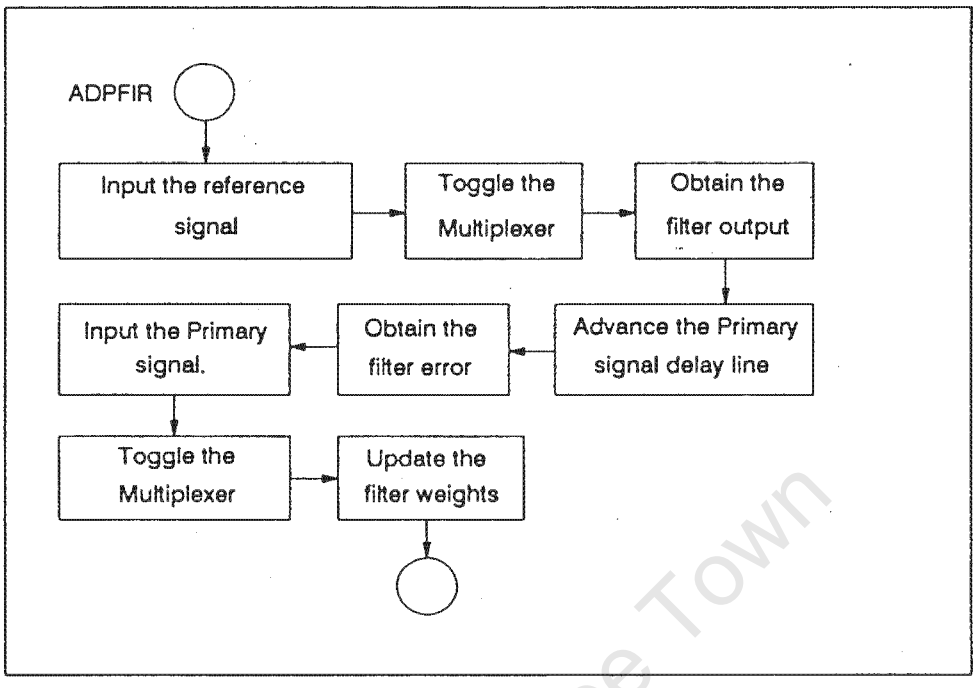


Figure C.1 Flow diagram of the subroutine ADPFIR found in section C.1.

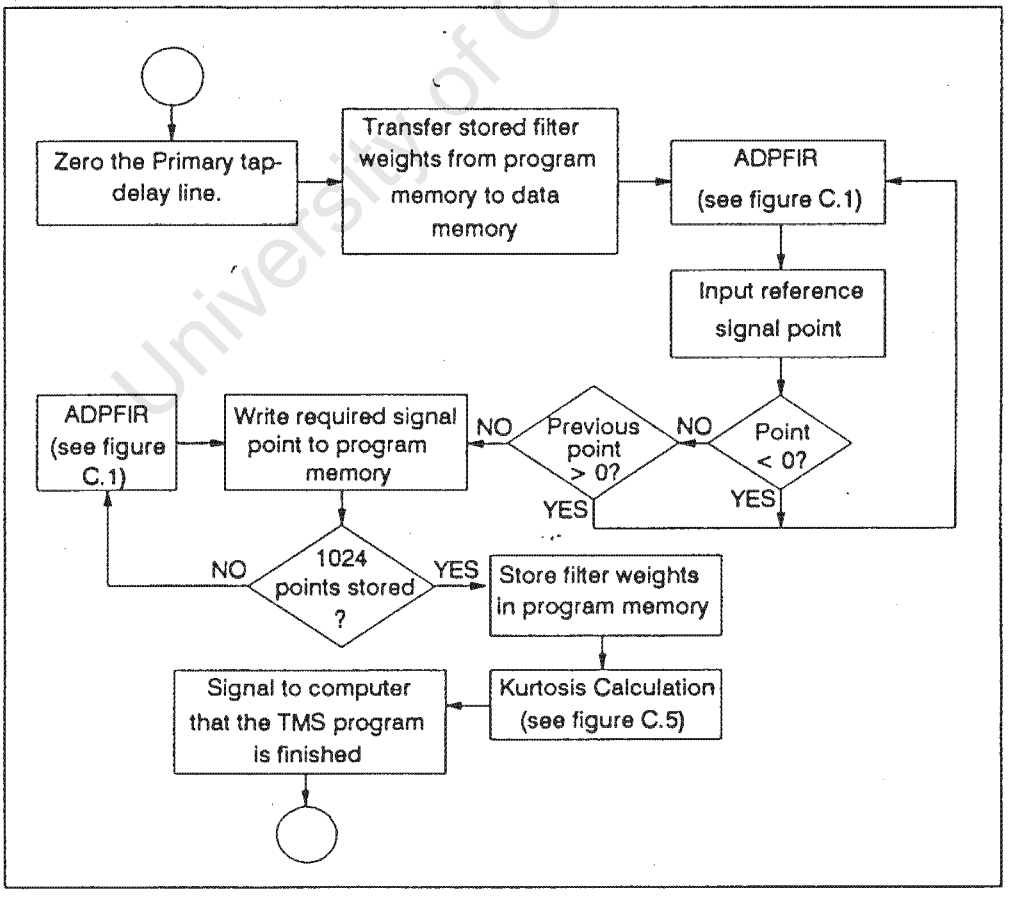


Figure C.2 The TMS program when set to display the Time Domain.

APPENDIX C

LISTINGS AND FLOW DIAGRAMS OF THE TMS ASSEMBLER PROGRAMS.

As the user selects different options in the Machine-Monitoring Package, so the managing program ADAPT.EXE alters the TMS 320C25 program. The flow-diagram of the TMS 320C25 program when the user has opted to display the time-trace of the signal is shown in figure C.2. When displays of the frequency or cepstral domains have been selected, then the program flows in the way shown in figures C.3 and C.4 respectively.

C.1 THE ADAPTIVE FILTER AND KURTOSIS PROGRAM.

The first part of this program is an adaptive noise canceller. Two signals are input via an external multiplexer, the toggling signal of which is generated in this program. The kurtosis of the recorded signal is also calculated. The program is designed to run in conjuction with the Turbo-C program ADAPT.EXE. The flow diagram of the subroutine ADPFIR is shown in figure C.1.


```
                                ;
        tms     320c25          ; The signal processor model.

        aorg    >108            ; The program starts at 108H in TMS
                                ; memory
filt    equ     18              ; 19 filter coeffs (initially).
entry   equ     >66             ; The first point of the primary
                                ; signal.
prima   equ     >67             ;
prim    equ     >68             ; Length to make the filter non-
                                ; causal.
toggle  equ     >76             ; Toggles the multiplexer.
mem     equ     >77             ; Counts the number of points stored.
finish  equ     >78             ; Signals the end of the program to
                                ; the PC.
one     equ     >79             ; A rounding constant.
beta    equ     >7a             ; LMS convergence factor.
err     equ     >7b             ; Current filter error.
x       equ     >7c             ; Current point of the reference
                                ; signal.
```

```
psamp   equ     >7d         ; Previous point of the reference
                            ; signal.
errf    equ     >7e         ; Error signal multiplied by beta.
y       equ     >7f         ; Current filter output.

temp    equ     >20         ;
m4l     equ     >21         ; Lower bytes of the signal's 4th power
m4h     equ     >22         ; Higher  "   "   "    "        "    "
m2h     equ     >23         ; Higher  "   "   "    "      2nd    "
m2l     equ     >24         ; Lower   "   "   "    "        "    "
prgmem  equ     >25         ; Beginning of each block of 128 pts.
aux4    equ     >26         ; Store auxiliary register number 4.

frstap  equ     >0300       ; Address of previous reference signal
                            ; sample.
lastap  equ     >0312       ; Address of oldest reference signal
                            ; sample.
lastdat equ     >0313
coeffp  equ     >ff00       ; Program memory address of 1st filter
                            ; weight.
coeffd  equ     >0200       ; Data memory address of 1st filter
                            ; weight.
                            ;
        dint                ; Disable interrupts.
        cnfd                ; Configure memory B0 as data memory.
        ldpk    0           ; Point to data page 0
        lack    0           ;\
        sacl    entry       ; \
        larp    1           ;  \  Zero all the places where the
        lark    1,entry     ;  /  primary signal is stored.
        rptk    7           ; /
        dmov    *+          ;/
        lalk    >200
        sacl    mem         ; Load 200H into the first storage
                            ; address.
        lack    >49
        sacl    finish      ; Load 'finish' with 49H.
        out     finish,7    ; Output '49H' to the PC. 49H is
                            ; arbitrary.
        lack    >58         ; Load 'finish' with 58H (not
                            ; arbitrary).
        sacl    finish
        lack    1
        sacl    one         ; Load 'one' with you-know-what.
        lalk    >2000
        sacl    beta        ; Load the convergence factor
        lalk    >1
        sacl    x           ; Set the first point of the reference
                            ; signal.

Recall: lalk    >D80        ;\
        lrlk    1,>200      ; \  Recall the filter weights.
        rptk    filt        ; /
        tblr    *+          ;/
Start:
        dmov    x           ; Move 'x' into 'psamp'
        call    Adpfir      ; Filter the signal and update the
```

```
                                ; filter weights.
        lac     x               ;\
        blez    Start           ; \  Test for a zero-crossing of the
        lac     psamp           ; /  ref. signal in the positive
        bgz     Start           ;/   direction (acts as a trigger).
                                ;
                                ; STORE 1024 POINTS IN PROGRAM MEMORY.
                                ;
Again:  lac     mem             ;\  Store the filter output at the
        tblw    y               ; \ address specified by 'mem' and
        addk    1               ; / then increment 'mem'.
        sacl    mem             ;/
        sblk    >600            ;\ If 1024 points have been stored
        bgez    Ends            ;/ then exit the program.
        call    Adpfir          ;
        b       Again           ; Go to the label 'Again'.
                                ; ADAPTIVE NOISE CANCELLER USING
                                ; THE LMS ALGORITHM.
Adpfir:
        bioz    Got             ;\
        b       Adpfir          ; > Get the reference signal
Got:    in      x,3             ;/  from the A/D.

        zac
        sacl    toggle          ; Zero 'toggle' for a -5V output.
        out     toggle,4        ; Output -5 volts to the multiplexer
        lalk    >fff            ;
        sacl    toggle          ; Store >fff in 'toggle' for a 5V
                                ; output.
        lac     x               ;\
        sblk    >800            ; > Subtract 800H from the ref. signal.
        sacl    x               ;/

        cnfp                    ; Configure B0 as program memory.
        mpyk    0               ; Clear the product register.
        lac     one,14          ; Load the rounding bit.
        larp    3               ; Choose Auxiliary register number 3
        lrlk    3,lastap        ; Point to the oldest sample. Fir:
        rptk    filt            ;\ Filter the reference signal.
        macd    coeffp,*-       ;/
        cnfd                    ; Re-configure B0 as data memory.
        apac                    ; Add the final product to the rest.
        sach    y,1             ; Store the result as the filter output.
                                ;
        larp    1               ;\
        lrlk    1,prima         ; \   Shift the primary signal register
        rptk    7               ;  >  one place.
        dmov    *-              ; /
        larp    3               ;/
                                ;
        neg                     ;\  Subtract the filter output from the
        add     prim,15         ; > primary signal and store as the
        sach    err,1           ;/  filter error.
                                ;
        lt      err             ;\
        mpy     beta            ; \
        pac                     ;  > errf = err * beta
```

```
        add     one,14      ; /
        sach    errf,1      ;/
                            ;
        mar     *+          ;\
        lac     x           ; > Include the newest sample
        sacl    *           ;/
                            ;
Notyet: bioz    Gottim      ; Input the primary signal from the A/D
        b       Notyet      ;
Gottim: in      entry,3     ;
        out     toggle,4    ; Output -5 volts to the multiplexer
        lac     entry       ;\
        sblk    >800        ; > Get rid of the D.C. bias of the A/D
        sacl    entry       ;/
                            ;
        lrlk    2,coeffd    ; Point to the filter coefficients.
        lrlk    3,lastdat   ; Point to the data samples.
        lt      errf        ;
        mpy     *-,2        ; P = 2 * beta * err * oldest data
                            ; sample.
        b       >E00        ;
                            ; The part of the program that
                            ; calculates the KURTOSIS and stores
                            ; the filter for the next round.
Ends:   larp    1           ;\
        lrlk    1,>200      ; \
        lalk    >D80        ;  >  Store the filter weights in
        rptk    filt        ; /   program memory.
        tblw    *+          ;/
                            ;
Ktosis: ldpk    4           ; THE START OF THE KURTOSIS PROGRAM.
        zac
        sacl    m4l         ;\
        sacl    m4h         ; \ Zero the kurtosis variables.
        sacl    m2h         ; /
        sacl    m2l         ;/
        lalk    >200
        sacl    prgmem
Loop1:
        lrlk    0,>3ff
        larp    4
        lrlk    4,>300
        lac     prgmem
        rptk    255         ; Read the appropriate 256 data points
        tblr    *+          ; to data memory.
        lrlk    4,>300
Loop2:
        sqra    *+          ;\
        pac                 ; \
        rptk    6           ;  \
        sfr                 ;   \
        sacl    temp        ;    > Square the first point and
        adds    m2l         ;   /  add it to the second moment.
        addh    m2h         ;  /
        sacl    m2l         ; /
        sach    m2h         ;/
        sqra    temp        ;\
```

```
        pac                 ; \
        rptk    8           ;  \
        sfr                 ;   \   Calculate Σx².
        adds    m4l         ;   /
        addh    m4h         ;  /
        sacl    m4l         ; /
        sach    m4h         ;/
        cmpr    2
        bbz     Loop2       ; If 256 points have been calculated
                            ; then carry on.
        lac     prgmem      ;\
        adlk    >100        ; \   Set up the program for the next
        sacl    prgmem      ;  >  256 points. If 1024 points have
        sblk    >600        ; /   been done then carry on.
        blz     Loop1       ;/
                            ;
Normal:
        zals    m4l         ;\ Put the 4th moment into the
        addh    m4h         ;/ accumulator.
        lark    4,0         ;\ Initialize Aux. Register no. 4
        rptk    19          ;/
        norm    *+          ;\ Normalize m4.
        sbrk    1           ;/
        sar     4,aux4      ;  Store no. of Left-shifts required.
        sacl    m4l         ;\ Store the Normalized m4.
        sach    m4h         ;/

        zals    m2l         ;\
        addh    m2h         ; \
        rptk    9           ;  > Shift m2 by 10 places to the left.
        sfr                 ; /
        sacl    m2l         ;/
                            ;
        sqra    m2l         ;\
        pac                 ; >  Square m2.
        sfl                 ;/

        rpt     aux4        ;\
        norm    *-          ; \  Normalize and store m2.
        sach    m2h         ; /
        sar     4,aux4      ;/

        lac     aux4        ;\ If m2 has been shifted fewer times
        blz     Skip        ;/ than m4 then do the next part.

        zals    m4l         ;\
        addh    m4h         ; \  Shift m4 right until the total no.
        rpt     aux4        ;  > of left shifts equals that of m2.
        sfr                 ; /
        sach    m4h         ;/
Skip:
        lalk    >1fe        ;\
        lrlk    4,>222      ; \  Put Σx⁴ and (Σx²)² into
        rptk    1           ; /  program addresses >1fe and >1ff.
        tblw    *+          ;/
        ldpk    0           ;
```

```
        out     finish,7  ; Output '58H' to the two-way register
Stop:   b       Stop      ; (port 7) and waste time until the PC
                          ; sees this.
```

C.2 THE ABSOLUTE-VALUE AND LOGARITHMIC CONVERSION PROGRAM.

A program that finds the modulus of each point of data and returns its logarithm by looking up the value in the table located at >B80 to >D7F in program memory.

```
        tms     320c25
        aorg    >0ABF     ; Program starts at 0ABF in memory.

AR0     equ     0         ; Auxiliary register no. 0
AR1     equ     1         ;     "        "     "   1
Page0   equ     0         ; Data page no. 0.
finish  equ     >78       ; Defined in Adaptive-Filter Program.
dat1st  equ     >200      ; 1st data address in page 4.
prg1st  equ     >200      ; 1st point of spectrum or cepstrum.
prgmid  equ     >300      ; Halfway through spectrum or cepstrum.
points  equ     >400      ; No. of points to be processed (1024).
logadr  equ     >B80      ; Starting address of log table.
datlen  equ     255       ; Maximum no. of repetitions allowed.

        ldpk    Page0     ;\
        lrlk    AR1,dat1st; \
        lalk    prg1st    ; \
        larp    AR1       ; \
        rptk    datlen    ;  \  Read 512 points of the FFT
        tblr    *+        ;   > from program to data memory.
        lalk    prgmid    ;  /
        rptk    datlen    ; /
        tblr    *+        ;/
        lrlk    AR0,points; /
        lrlk    AR1,dat1st;/
                          ;
Loop:   lac     *         ;\
        abs               ; \  Take the modulus of each point
        sfr               ;  > and look up its logarithmic
        adlk    logadr    ; /  value.
        tblr    *+        ;/
        cmpr    0         ; See if 512 points have been
        bbz     Loop      ; processed.
                          ;
        lrlk    AR1,dat1st;\
        lalk    prg1st    ; \
        rptk    datlen    ;  \  Write the modified points to
        tblw    *+        ;   > program memory.
        lalk    prgmid    ;  /
        rptk    datlen    ; /
        tblw    *+        ;/
                          ;
        out     finish,7  ; Tell the PC we're finished.
Stop:   b       Stop      ;
```

C.3 THE SPREADING PROGRAM.

A program to spread 256 points of data over 512 points. It is desired that the cepstrum of 256 points be displayed on the same axes as the other traces, which are 512 points long.

```
        tms     320c25
        aorg    >b28        ; Start the program at address >b28.

AR0     equ     0           ; Auxiliary register 0.
AR1     equ     1           ;     "        "     1.
Page0   equ     0           ; Data-page 0.
finish  equ     >78         ; Defined in Adaptive-Filter Program.
dat1st  equ     >200        ; 1st data memory location in page 4.
dat2nd  equ     >201        ; 2nd  "     "        "     "   "   4.
prg1st  equ     >200        ; 1st address where Cepstrum is stored.
lastpg  equ     >300        ; Address half-way through Cepstrum.
datlen  equ     255         ; Maximum no. of times allowed for
                            ; repeating an instruction.

        lark    AR0,2       ; Set Data Memory Address increment.
        larp    AR1
        lalk    prg1st      ;\
        lrlk    AR1,dat1st  ; \
        rptk    datlen      ;  \
        tblr    *0+         ;   \  Load the first 256 points of
        lalk    prg1st      ;   /  program memory into 512 points
        lrlk    1,dat2nd    ;  /   of data memory.
        rptk    datlen      ; /
        tblr    *0+         ;/

        lalk    prg1st      ;\
        lrlk    AR1,dat1st  ; \
        rptk    datlen      ;  \   Load data memory back into
        tblw    *+          ;   >  program memory.
        lalk    lastpg      ;  /
        rptk    datlen      ; /
        tblw    *+          ;/

        ldpk    Page0       ; Change current data page to no. 0.
        out     finish,7    ; Tell the PC we're finished.
Loop:   b       Loop        ;
```

C.4 MEMORY ALLOCATION FOR THE TMS 320C25

Figure C.6 shows the position of programs in the TMS 320C25 when ADAPT.EXE is running. The numbers at the top right-hand corner of each block represent the respective block's starting address in hexadecimal. The numbers at the bottom right-hand corner of some of the blocks represents the end address of the respective block.

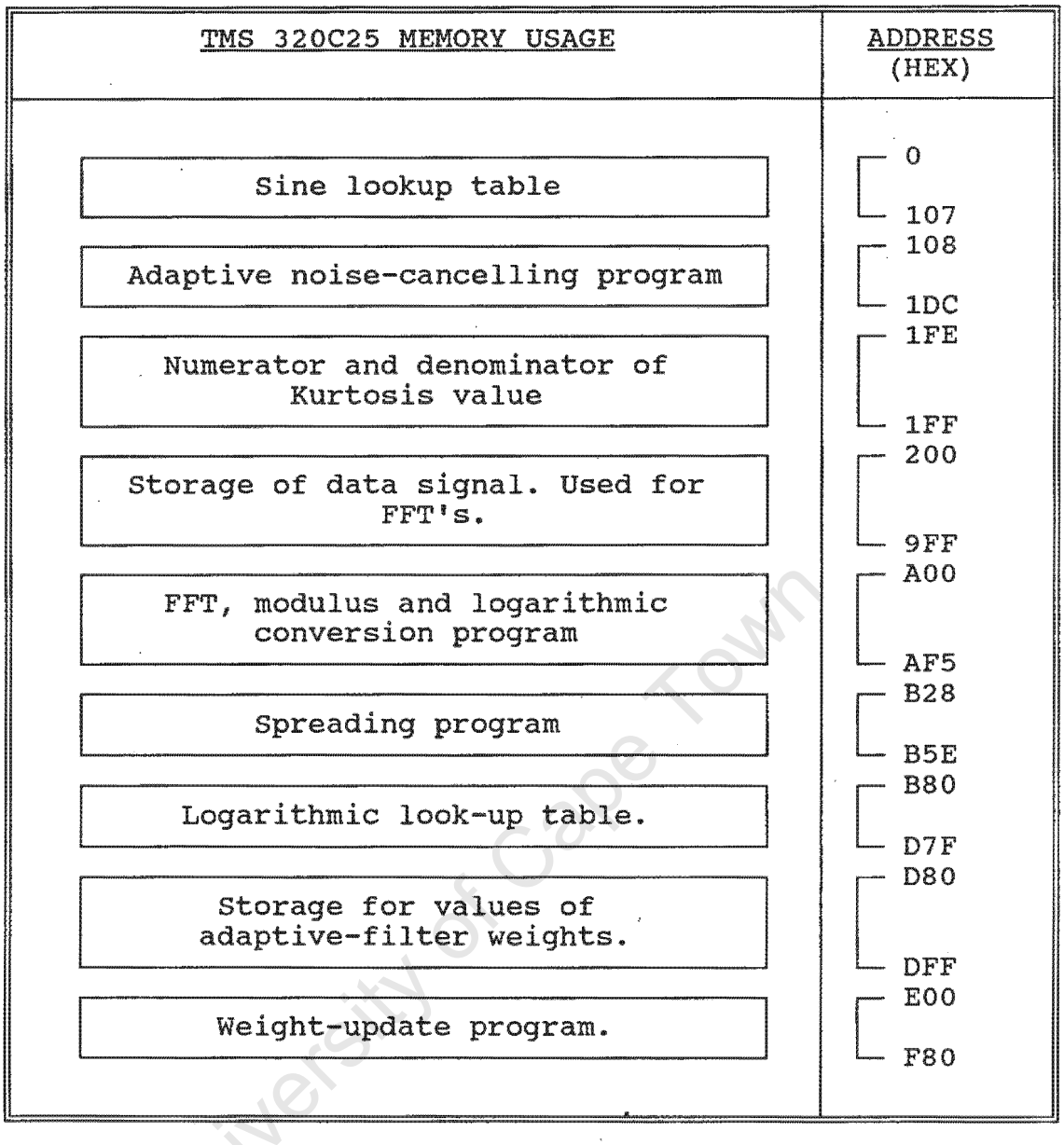


Figure C.6. Allocation of memory in the TMS 320C25

APPENDIX D

THE MANAGING PROGRAM OF THE MACHINE-MONITORING PACKAGE

This program performs the following functions:

1) Display either the time-, frequency- or cepstral-domain signal.
2) The signal displayed can either be the correlated part (filter output), the non-correlated part (filter error), the noise signal (reference signal) or the signal obtained from the component under test (primary signal).
3) The kurtosis of the signal is displayed at all times.
4) The display can be zoomed in or out at the user's will.

These keys perform the following functions:

F1: Enlarge the display (zoom in).
F2: Shrink the display (zoom out).
F3: Change from time- to frequency- to cepstral-domain and back again.
F4: Change the signal displayed to either the correlated or the uncorrelated signal.
F5: Change the signal displayed to either the reference signal or the primary signal.
F6: Record 60 maximum values of kurtosis taken over two minutes.
F7: Reset the timer for the A/D converter.
F8: Quit the program.
F10: Pause the Program.

```
#include <stdio.h>
#include <bios.h>
#include <math.h>
#include <dos.h>
#include <dir.h>
#include <string.h>
#include <graphics.h>
#include <time.h>
#include "adapt.h"
```

```
int x[512],                 /* Array displayed on CRT.            */
    y[514],                 /* Array containing TMS data: the     */
                            /*  1st 2 points contain the          */
                            /*  numerator and denominator of the  */
                            /*  recorded kurtosis value.          */
    z[1024],                /* Zeroes for filling the imaginary   */
                            /*  part of the FFT routine.          */
    scale  = 5,             /* Divide y-array by 2^scale.         */
    offset = 100,           /* Display the signal in the middle   */
                            /*  of the screen.                    */
in_or_out = OUTPUTSIG,      /* Determines which type of signal    */
                            /*  is displayed: input or output.    */
    record = -1;            /* Negative if recording of kurtosis  */
                            /*  is not required.                  */

float maxkurt = 0.0,        /* Maximum kurtosis during 2 secs.    */
      ktosis[60];           /* Array of kurtosis values recorded.*/
char  filename[9];          /* File used to record values.        */

unsigned seg320 = 0xD000,   /* Base segment of PC RAM.            */
         io320  = 0x0300;   /* Base address of TMS data memory.   */

int bandwidth,              /* Maximum freq. of the signals.      */
    address,
    lastaddress = 0x0E39;   /* Addresses used for changing the    */
                            /* filter length.                     */

time_t prevclk, *ptr;       /* Seconds since 1 Jan 1970.          */

main()
{
/*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
/*                          THE MAIN PROGRAM                           */
/*                                                                     */
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
    float xaxis[9], endaxis;
    int i;
    int output = CORR,      /* CORR. OR UNCORR. SIGNAL DISPLAYED.*/
        input = SIGNAL,     /* REF OR PRIM. SIGNAL DISPLAYED.    */
        mode = TIME;        /* TIME-DOMAIN IS DISPLAYED.         */

    int gotkey = F1;

    for (i=0; i<514; i++)
        y[i] = 0;
    for (i=0; i<1024; i++)
        z[i] = 0;
    setup();

    while (gotkey != F8)
    {
        while (bioskey(1) == 0) /* WHILE NO KEY IS PRESSED */
        {
            mainloop();
            if (mode == CEPS) cepstrumloop();
            peekwf(0x1fe, seg320, y, 514);   /* TRANSFER THE   */
            pokewf(z, 0x0600, 1024, seg320); /* SIGNAL FROM    */
```

```
        }                                   /* THE TMS320 TO THE PC. */
        gotkey = getkey();
        switch (gotkey)                   /* ACT ON THE KEY PRESSED */
        {
            case F1: ++scale;              break;
            case F2: --scale;              break;
            case F3: timefreq (&mode);     break;
            case F4: corrsig (&output);    break;
            case F5: primref (&input);     break;
            case F6: recorddata();         break;
            case F7: bandwidth = settimer();
                     switch (mode)
                     {
/* RE-LABEL THE */       case TIME: endaxis = 256.0/bandwidth; break;
/* X-AXIS.      */       case FREQ: endaxis = 1.0 * bandwidth; break;
                         case CEPS: endaxis = 128.0/bandwidth;
                     }
                     for (i=0; i<=8; i++)
                     {
                         xaxis[i] = i*endaxis/8;
                         labelaxis (xaxis, i);
                     } break;
            case F10: while (bioskey(1) == 0);
        }
    }
    co80320();          /* RETURN TO TEXT-MODE. */
} /* END OF PROGRAM */

/*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
/*                              SUBROUTINES                                 */
/*                                                                          */
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
void setup (void)
{
/************************************************************************/
/* USED AT THE BEGINNING OF THE PROGRAM TO SET UP THE SCREEN   */
/* AND TO LOAD THE TMS320 PROGRAM.                                      */
/************************************************************************/
    int i;
    int top=1, bottom=24, left=1, right=65;
    float xaxis[9], endaxis;
    int gdriver, gmode;

    clrscr ();
    hlt320 (io320);          /* MAKE SURE THE TMS320 IS STOPPED */

    pokewf (table1,    0x0000, 0x0108, seg320); /* LOAD THE     */
    pokewf (adaptive,  0x0108, 0x00D5, seg320); /* TMS320 PROG. */
    pokewf (fft,       0x0A00, 0x00F6, seg320); /* INTO MEMORY. */
    pokewf (spread,    0x0B28, 0x0038, seg320);
    pokewf (logtable,  0x0B80, 0x0200, seg320);

    for (i=0; i<8; i++)       /* LOAD THE WEIGHT-UPDATE ROUTINE */
        pokewf(update, 0x0E00+48*i, 0x0030, seg320);
                              /* LOAD THE LAST WEIGHT UPDATE.   */
    pokewf (endweight,0x0F7D, 0x0004, seg320);
    bandwidth = settimer (); /* SET THE A/D CONVERSION SPEED.  */
```

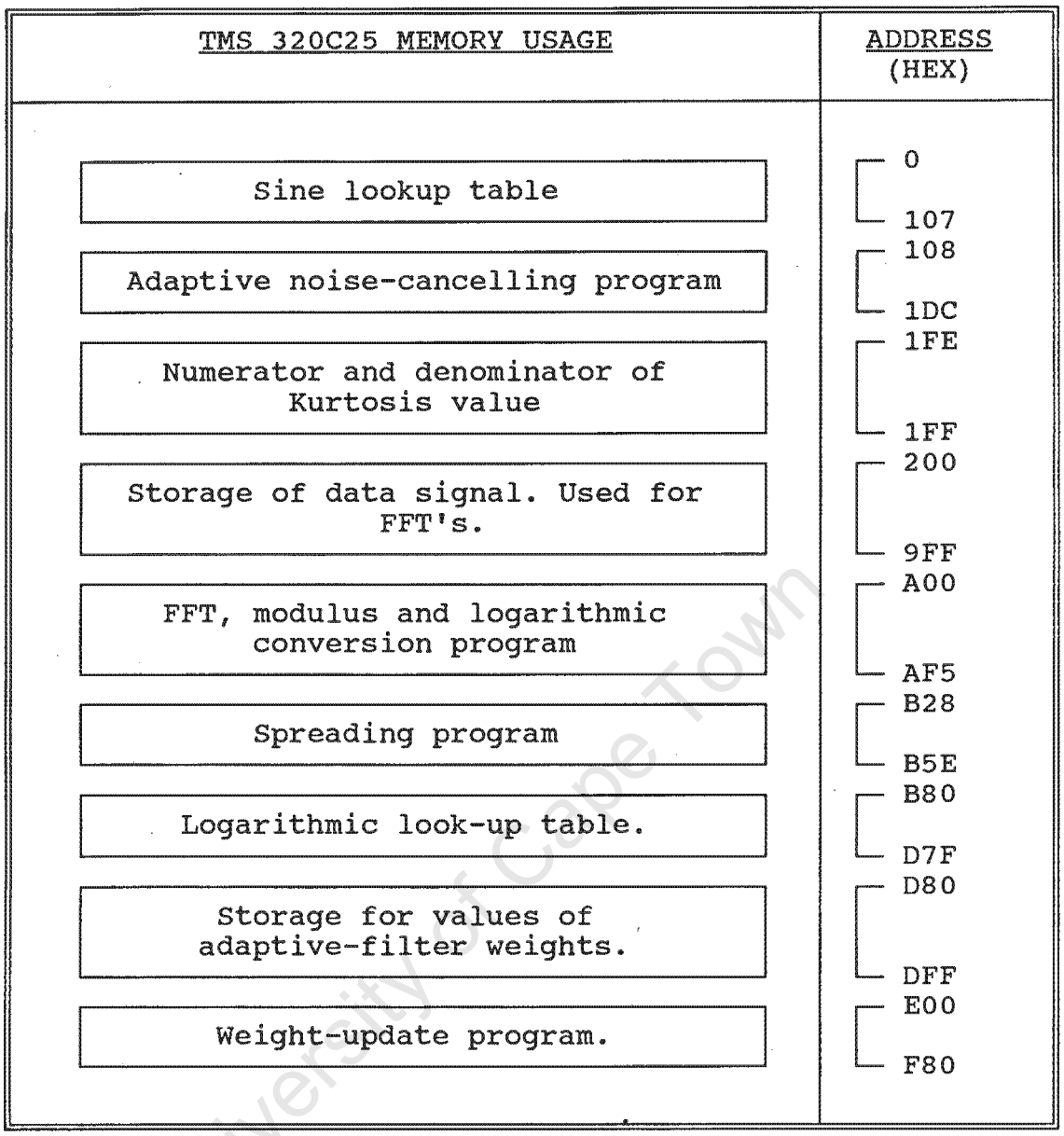


Figure C.6. Allocation of memory in the TMS 320C25

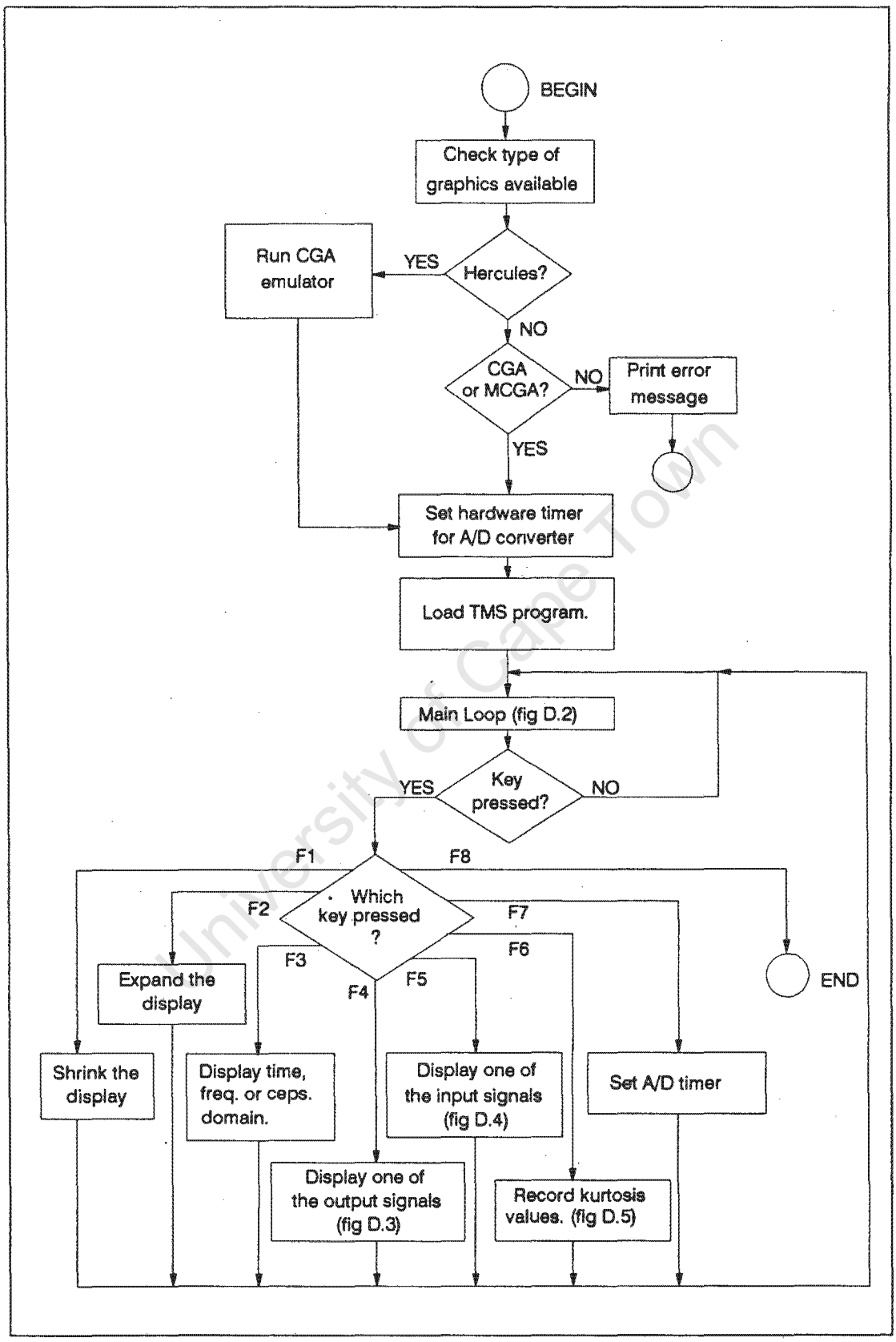


Figure D.1 General Flow Diagram of the Adaptive Noise Cancelling Package

APPENDIX D

THE MANAGING PROGRAM OF THE MACHINE-MONITORING PACKAGE

This program performs the following functions:

1) Display either the time-, frequency- or cepstral-domain signal.
2) The signal displayed can either be the correlated part (filter output), the non-correlated part (filter error), the noise signal (reference signal) or the signal obtained from the component under test (primary signal).
3) The kurtosis of the signal is displayed at all times.
4) The display can be zoomed in or out at the user's will.

These keys perform the following functions:

F1: Enlarge the display (zoom in).
F2: Shrink the display (zoom out).
F3: Change from time- to frequency- to cepstral-domain and back again.
F4: Change the signal displayed to either the correlated or the uncorrelated signal.
F5: Change the signal displayed to either the reference signal or the primary signal.
F6: Record 60 maximum values of kurtosis taken over two minutes.
F7: Reset the timer for the A/D converter.
F8: Quit the program.
F10: Pause the Program.

```
#include <stdio.h>
#include <bios.h>
#include <math.h>
#include <dos.h>
#include <dir.h>
#include <string.h>
#include <graphics.h>
#include <time.h>
#include "adapt.h"
```

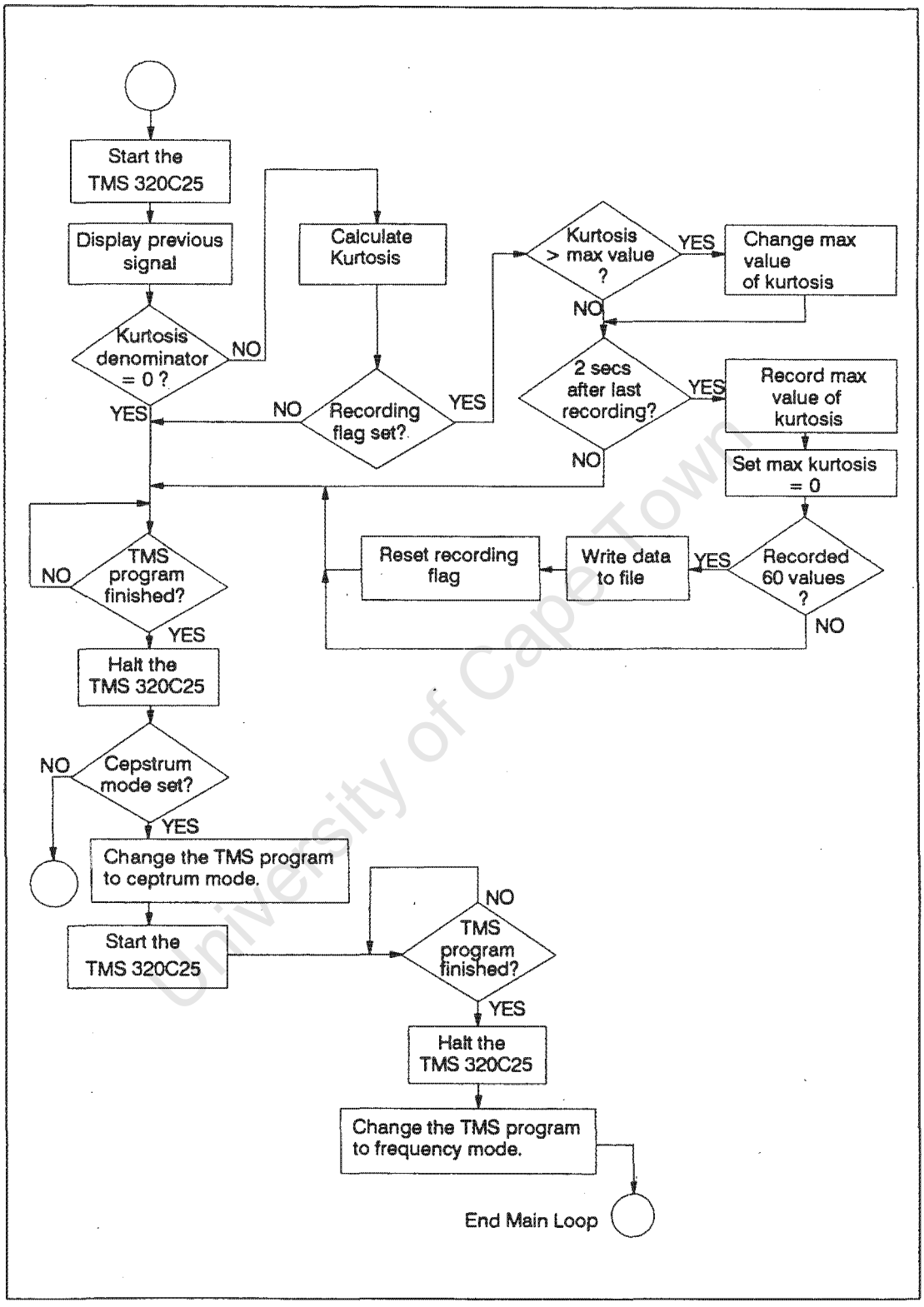


Figure D.2 Flow diagram of the Main Program Loop

```
int x[512],                 /* Array displayed on CRT.              */
    y[514],                 /* Array containing TMS data: the       */
                            /*  1st 2 points contain the            */
                            /*  numerator and denominator of the    */
                            /*  recorded kurtosis value.            */
    z[1024],                /* Zeroes for filling the imaginary     */
                            /*  part of the FFT routine.            */
    scale  = 5,             /* Divide y-array by 2^scale.           */
    offset = 100,           /* Display the signal in the middle     */
                            /*  of the screen.                      */
in_or_out = OUTPUTSIG,      /* Determines which type of signal      */
                            /*  is displayed: input or output.      */
    record = -1;            /* Negative if recording of kurtosis    */
                            /*  is not required.                    */

float maxkurt = 0.0,        /* Maximum kurtosis during 2 secs.      */
      ktosis[60];           /* Array of kurtosis values recorded.   */
char  filename[9];          /* File used to record values.          */

unsigned seg320 = 0xD000,   /* Base segment of PC RAM.              */
         io320  = 0x0300;   /* Base address of TMS data memory.     */

int bandwidth,              /* Maximum freq. of the signals.        */
    address,
    lastaddress = 0x0E39;   /* Addresses used for changing the      */
                            /* filter length.                       */

time_t prevclk, *ptr;       /* Seconds since 1 Jan 1970.            */

main()
{
/*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
/*                          THE MAIN PROGRAM                            */
/*                                                                      */
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
    float xaxis[9], endaxis;
    int i;
    int output = CORR,      /* CORR. OR UNCORR. SIGNAL DISPLAYED.*/
        input = SIGNAL,     /* REF OR PRIM. SIGNAL DISPLAYED.    */
        mode = TIME;        /* TIME-DOMAIN IS DISPLAYED.         */

    int gotkey = F1;

    for (i=0; i<514; i++)
        y[i] = 0;
    for (i=0; i<1024; i++)
        z[i] = 0;
    setup();

    while (gotkey != F8)
    {
        while (bioskey(1) == 0) /* WHILE NO KEY IS PRESSED */
        {
            mainloop();
            if (mode == CEPS) cepstrumloop();
            peekwf(0x1fe, seg320, y, 514);   /* TRANSFER THE   */
            pokewf(z, 0x0600, 1024, seg320); /* SIGNAL FROM    */
```

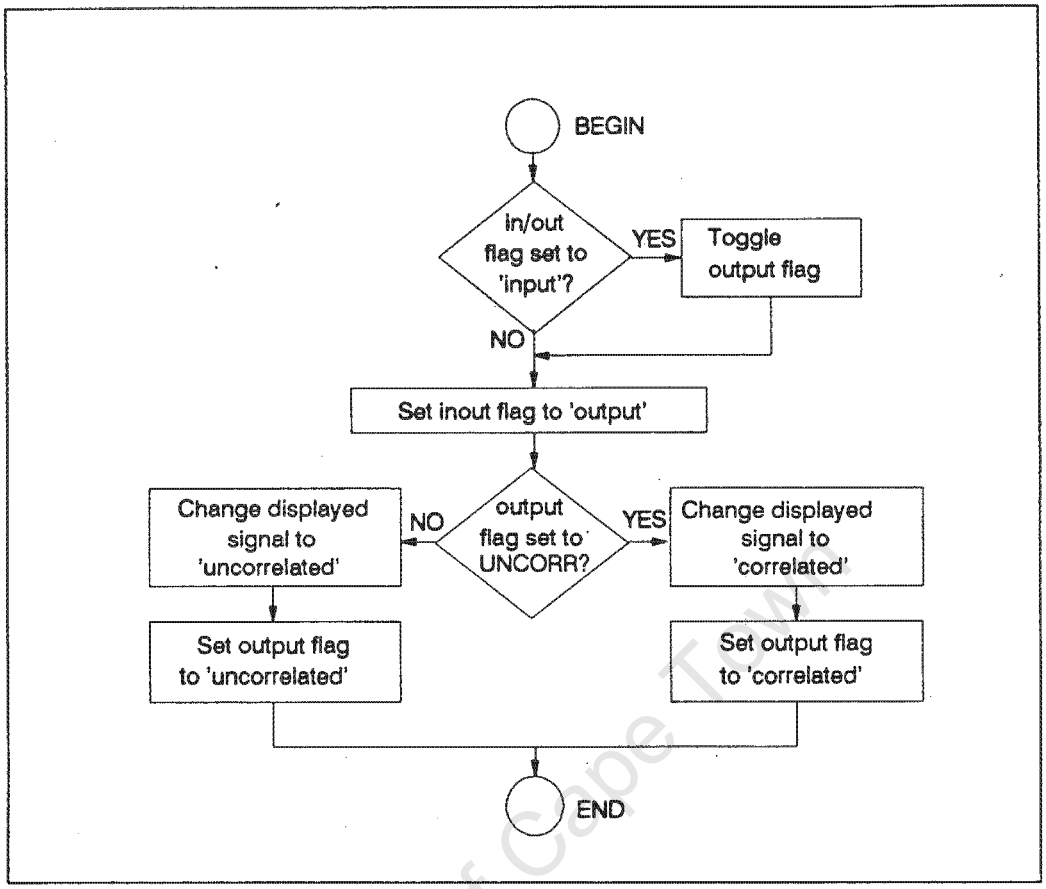


Figure D.3 Display one of the Output Signals

```
        }                                   /* THE TMS320 TO THE PC. */
        gotkey = getkey();
        switch (gotkey)                   /* ACT ON THE KEY PRESSED */
        {
            case F1: ++scale;              break;
            case F2: --scale;              break;
            case F3: timefreq (&mode);     break;
            case F4: corrsig (&output);    break;
            case F5: primref (&input);     break;
            case F6: recorddata();         break;
            case F7: bandwidth = settimer();
                     switch (mode)
                     {
/* RE-LABEL THE */       case TIME: endaxis = 256.0/bandwidth; break;
/* X-AXIS.      */       case FREQ: endaxis = 1.0 * bandwidth; break;
                         case CEPS: endaxis = 128.0/bandwidth;
                     }
                     for (i=0; i<=8; i++)
                     {
                         xaxis[i] = i*endaxis/8;
                         labelaxis (xaxis, i);
                     } break;
            case F10: while (bioskey(1) == 0);
        }
    }
    co80320();        /* RETURN TO TEXT-MODE. */
} /* END OF PROGRAM */

/*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
/*                          SUBROUTINES                                  */
/*                                                                       */
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
void setup (void)
{
/*************************************************************************/
/* USED AT THE BEGINNING OF THE PROGRAM TO SET UP THE SCREEN   */
/* AND TO LOAD THE TMS320 PROGRAM.                                       */
/*************************************************************************/
    int i;
    int top=1, bottom=24, left=1, right=65;
    float xaxis[9], endaxis;
    int gdriver, gmode;

    clrscr ();
    hlt320 (io320);          /* MAKE SURE THE TMS320 IS STOPPED */

    pokewf (table1,   0x0000, 0x0108, seg320); /* LOAD THE     */
    pokewf (adaptive, 0x0108, 0x00D5, seg320); /* TMS320 PROG. */
    pokewf (fft,      0x0A00, 0x00F6, seg320); /* INTO MEMORY. */
    pokewf (spread,   0x0B28, 0x0038, seg320);
    pokewf (logtable, 0x0B80, 0x0200, seg320);

    for (i=0; i<8; i++)       /* LOAD THE WEIGHT-UPDATE ROUTINE */
        pokewf(update, 0x0E00+48*i, 0x0030, seg320);
                              /* LOAD THE LAST WEIGHT UPDATE.   */
    pokewf (endweight,0x0F7D, 0x0004, seg320);
    bandwidth = settimer (); /* SET THE A/D CONVERSION SPEED.  */
```

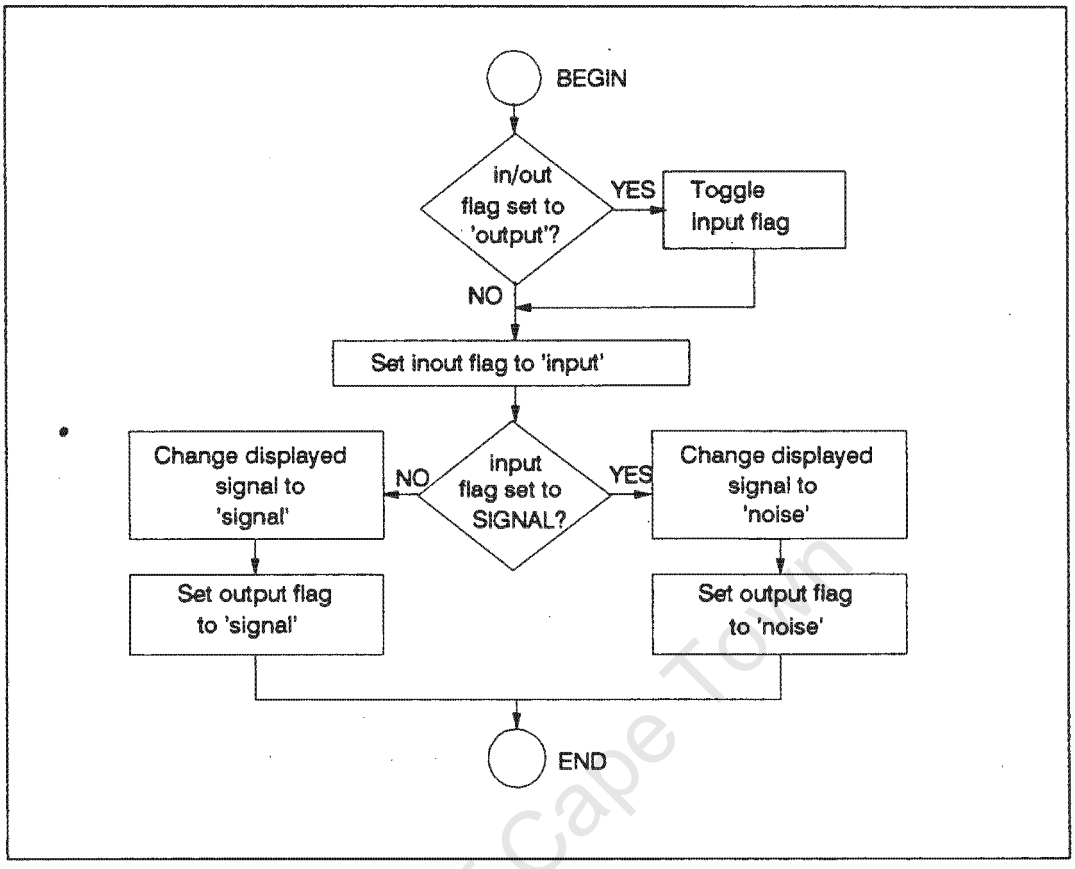


Figure D.4 Display one of the Input Signals

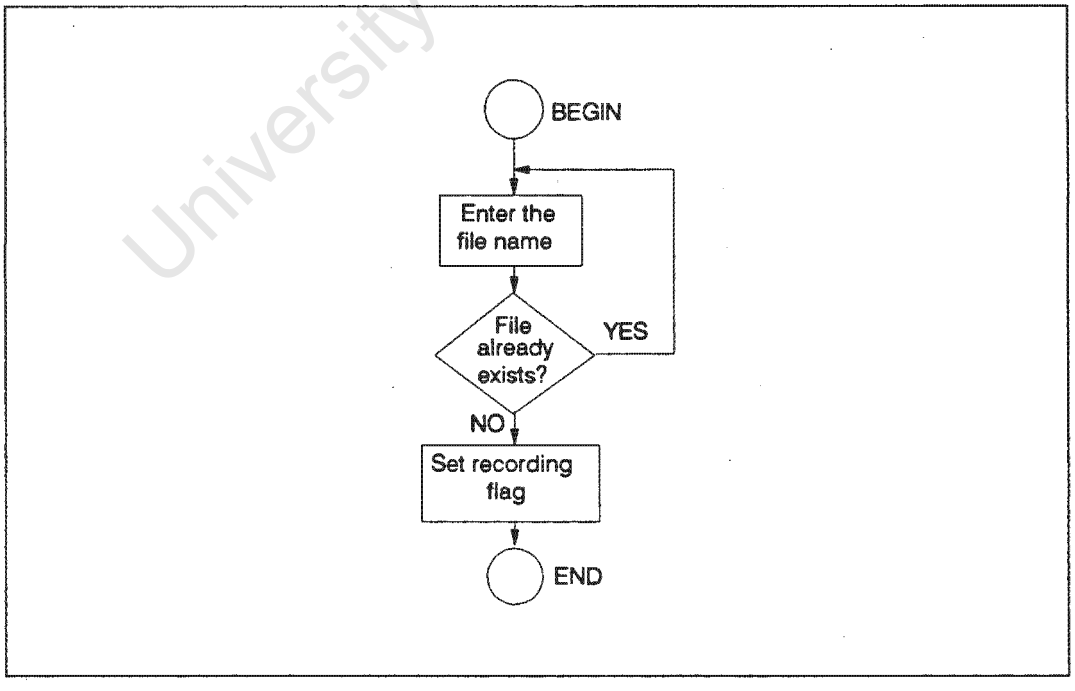


Figure D.5 Record Kurtosis values.

```
    detectgraph (&gdriver, &gmode); /* CHECK FOR CORRECT        */
    if (gdriver < 0)                /* GRAPHICS DRIVER.        */
    {
        puts ("Sorry, no graphics driver detected.");
        exit (0);
    }
    if (gdriver != HERCMONO && gdriver != CGA && gdriver != MCGA)
    {
        puts ("Graphics must be Hercules, CGA or MCGA.");
        exit (0);
    }
    system ("a:");                  /* CHANGE THE DEFAULT DRIVE */

    if (gdriver == HERCMONO)
        system ("fix");             /* EMULATE A CGA DRIVER.    */
    hres320 ();

    /**** DRAW THE COMPUTER SCREEN ****/

    for (i=top; i<=bottom; i++)
    {
        gotoxy (right,i);
        printf ("|");
    }
    gotoxy (left,bottom);
    printf ("_____________________________________");
    printf ("_____________________________________");

    gotoxy (70, 1);  printf ("domain:");
    gotoxy (69, 2);  printf ("  TIME   ");
    gotoxy (70, 5);  printf ("signal:");
    gotoxy (68, 6);  printf (" CORRELATED ");
    gotoxy (68, 9);  puts ("F1: Zoom out");
    gotoxy (68,10);  puts ("F2: Zoom in ");
    gotoxy (68,11);  puts ("F3: Time/Freq");
    gotoxy (68,12);  puts ("    Cepstrum.");
    gotoxy (68,13);  puts ("F4: Corr/Uncr");
    gotoxy (68,14);  puts ("F5: Sig/Noise");
    gotoxy (68,15);  puts ("F6: Rec Ktsis");
    gotoxy (68,16);  puts ("F7: Set timer");
    gotoxy (68,17);  puts ("F8: Quit");
    gotoxy (68,23);  puts ("KURTOSIS");

    endaxis = 256.0/bandwidth;
    for (i=0; i<9; i++)
    {
        xaxis[i] = i*endaxis/8;
        labelaxis (xaxis, i);
    }
    gotoxy (69,25); printf ("mS");
}
/*************************************************************/
```

```
int settimer(void)
{
/*************************************************************/
/*  SET THE HARDWARE TIMER AND ALTER THE LENGTH OF THE FILTER. */
/*  THE DEFAULT VALUE, AND ALSO THE HIGHEST VALUE OF THE      */
/*  BANDWIDTH IS 27 kHz.                                      */
/*************************************************************/
    int gotkey = 0, counter = 0, freq = 0, timer;
    int i, weights, time_array[2] = {0, 0};
    int lowbyte, highbyte, maxfreq = 27;

    unsigned jumpaddr[2] = {0xFF80, 0x0F7D},
             restore[2]  = {0x7B8B, 0x3A9A},

    /** SET THE TMS PROGRAM FOR 19 FILTER WEIGHTS  **/

             filtchange1[2] = {0xCB12, 0x58A0},
             filtchange2[2] = {0xCB12, 0x59A0},
             filtchange3[2] = {0x0313, 0x3C7E},
             filtchange4[3] = {0x0312, 0xCB12, 0x5C90},
             betachange [2] = {0x4000, 0x607A};

    gotoxy (2,2); puts ("Enter the frequency range in kHz: ");
    gotoxy (2,3); puts ("Default value is 27 kHz.");

    /**** THE EDITOR ROUTINE FOR ENTERING THE SAMPLING RATE ****/

    do {
        while (bioskey(1) == 0);  /** HANG AROUND...          **/
        gotkey = getkey();        /** UNTIL A KEY IS PRESSED **/

    /** IF A NUMERAL IS PRESSED... **/
        if (gotkey >= 48 && gotkey <= 57 && counter <= 1)
        {
            gotoxy (36+counter, 2); printf ("%d", gotkey-48);
            time_array [counter] = gotkey-48;
            ++counter;
        }
        if (gotkey == BSPACE)  /** IF BACKSPACE IS PRESSED... **/
        {
            --counter;
            gotoxy (36+counter, 2); puts (" ");
            gotoxy (36+counter, 2);
            time_array [counter] = 0;
        }
    } while (gotkey != ENTER);

    /**** END OF EDITOR ROUTINE ****/

    /** ERASE THE TEXT PLACED ON THE SCREEN BY THIS ROUTINE **/

    gotoxy (2,2); puts ("                                          ");
    gotoxy (2,3); puts ("                          ");
```

```
    /** CONVERT THE CHARACTERS KEYED IN INTO A NUMBER **/

    for (i=0; i<=counter-1; i++)
    {
        freq += pow10 (i) * time_array[counter-i-1];
    }
    if (freq > maxfreq) freq = maxfreq;
    if (freq == 0) freq = maxfreq;

    /** CALCULATE THE TIMER CODE NUMBER **/

    timer = 2500 / freq;
    lowbyte = timer % 256;
    highbyte = timer / 256;

    outportb(0x307, 0xb4);              /* Set the A/D timer    */
    outportb(0x306, lowbyte);
    outportb(0x306, highbyte);

    /** CALCULATE THE NUMBER OF WEIGHTS IN THE FIR FILTER **/

    weights = 800.0/freq - 10;           /* AN EMPIRICAL FORMULA */
    if (weights > 125) weights = 125;
    address = 0x0E00 + weights*3;       /* ADDRESS FOR POKING THE */
                                        /* JUMP COMMAND.          */

    /** RE-WRITE THE TMS 320C25 PROGRAM **/

    filtchange1[0] = filtchange2[0] = filtchange4[1]
                   = 0xCB00 + weights-1;
    filtchange3[0] = 0x0300 + weights;
    filtchange4[0] = 0x0300 + weights-1;

    /** CHANGE THE LMS CONVERGENCE FACTOR **/

    betachange [0] = (16384.0/weights) * 19.0;

    /** ENTER THE CHANGES TO THE TMS 320C25 PROGRAM **/

    pokewf (filtchange1, 0x0125, 0x2, seg320);
    pokewf (filtchange2, 0x017F, 0x2, seg320);
    pokewf (filtchange3, 0x0175, 0x2, seg320);
    pokewf (filtchange4, 0x0150, 0x3, seg320);
    pokewf (betachange,  0x011C, 0x2, seg320);

    pokewf (restore, lastaddress, 0x2, seg320);
    pokewf (jumpaddr, address, 0x2, seg320);
    lastaddress = address;
    return (freq);
}
/*****************************************************************/
```

```
void labelaxis (float xaxis[], int i)
{
/*****************************************************************/
/*  RE-LABEL THE X-AXIS OF THE DISPLAY WHEN THE TIMER HAS BEEN */
/*  RESET OR THE MODE HAS BEEN CHANGED.                          */
/*****************************************************************/
    int a, rem;

    if (i == 0)
    {
        gotoxy (1, 25);
        printf("0");                    /* LABEL THE FIRST POINT */
    }
    else
    {
        gotoxy (8*i, 25);
        a = xaxis[i];
        rem = 100*(xaxis[i] - a);
        if (xaxis[i] >= 10.0)           /* IF THE NUMBER IS      */
        {                               /* GREATER THAN 10 ONLY  */
            rem /= 10;                  /* DISPLAY ONE DECIMAL.  */
            printf ("%2d.%1d", a, rem);
        }
        else
        {
            printf ("%1d.%2d", a, rem);
            if (rem < 10)                /* ELSE DISPLAY TWO      */
            {                            /* DECIMALS.             */
                gotoxy (8*i+2, 25);
                printf ("0");
    }   }   }
}
/*****************************************************************/
void mainloop(void)
{
/*****************************************************************/
/*  START THE TMS320, DISPLAY THE PREVIOUS SIGNAL AND HALT THE */
/*  TMS320 AFTER IT HAS COMPLETED ITS PROGRAM.                   */
/*                                                               */
/*****************************************************************/
    int i;

    go320(io320);
    for (i=0; i<512; i++)                /* SCALE THE POINTS OF */
        x[i] = offset - (y[i+2] >> scale);  /* THE SIGNAL.     */

    sigpts(x, 511);                      /* DISPLAY THE SIGNAL. */
    if (y[1] != 0) kurtrecord(y[0], y[1]);
    while( regin(io320) != 0x0058 );  /* WAIT UNTIL THE TMS 320
                                         HAS FINISHED.        */
    hlt320(io320);                       /* STOP THE TMS 320C25.  */
    pokewf(z, 0x0600, 512, seg320);
}
/*****************************************************************/
```

```
void cepstrumloop (void)
{
/*****************************************************************/
/*  CHANGE THE TMS320 PROGRAM TO EXECUTE THE NECESSARY STEPS     */
/*  TO OBTAIN THE CEPSTRUM OF THE SIGNAL. THEN RUN THE TMS320.   */
/*                                                               */
/*****************************************************************/
unsigned
     ceps_domain[2] = {0xFF80, 0x0B28},    /* BRANCH THE TMS320 */
                                           /* TO THE CEPSTRUM   */
                                           /* PART.             */
     stop_ceps  [2] = {0xC800, 0xE778};    /* REVERT TO THE     */
                                           /* ORIGINAL VERSION. */

    pokewf(ceps_domain, 0x0AF2,  0x2, seg320);
    pokewf(shorttable,  0x0000, 0x88, seg320);
                              /* SINE TABLE FOR 512-POINT FFT. */
    go320(io320);

    while (regin(io320) != 0x0058);  /* WAIT FOR THE TMS320 TO */
    hlt320(io320);                   /* FINISH.                */

    pokewf(stop_ceps, 0x0AF2,  0x2, seg320);
    pokewf(table1,    0x0000, 0x88, seg320);
                             /* SINE TABLE FOR 1024-POINT FFT. */
}
/*****************************************************************/
int getkey(void)
{
/*****************************************************************/
/*  RECEIVES A KEY FROM BIOS AND DETERMINES WHICH IT WAS.        */
/*                                                               */
/*****************************************************************/
    int key, hi, lo;

    key = bioskey(0);
    lo =  key & 0x00FF;
    hi = (key & 0xFF00) >> 8;
    return ((lo == 0) ? hi+256 : lo);
}
/*****************************************************************/
void kurtrecord(unsigned a, unsigned b)
{
/*****************************************************************/
/*  CALCULATES THE KURTOSIS VALUE.                               */
/*                                                               */
/*****************************************************************/
    int c, rem;
    float p, q, kurt;
    time_t timenow;

    p=a;  q=b;
    kurt=p/q;           /* FIND THE TRUE VALUE OF THE DIVISION */
    c=p/q;              /* FIND THE INTEGER PART               */
    rem = 1000*(kurt-c);  /* MULTIPLY THE FRACTION BY 1000.    */
    if (record < 0) kurtdisp (c, rem);
```

```
    /** RECORD KURTOSIS VALUES IF THE OPTION IS CHOSEN **/
    if (record >= 0)
    {
        if (kurt > maxkurt) maxkurt = kurt;
        timenow = time (ptr);
        if (timenow >= prevclk+2) /* IF 2 OR MORE SECONDS PAST */
        {                         /* THE LAST SAMPLE...       */
            c = maxkurt;
            rem = 1000*(maxkurt-c);
            kurtdisp (c, rem);     /* DISPLAY THE KURTOSIS.    */
            ktosis[record] = maxkurt;
            maxkurt = 0.0;        /* RESET THE MAX. KTSIS VALUE */

            ++record;
            gotoxy (72,21); printf ("%d", record);
            prevclk = timenow;    /* UPDATE THE CLOCK.        */

            if (record == 60) /* WRITE THE DATA TO A FILE IF   */
            {                 /* 60 POINTS HAVE BEEN COLLECTED.*/
                writedata();
                maxkurt = 0.0;
    }   }   }
}
/*****************************************************************/
void kurtdisp (int c, int rem)
{
/*****************************************************************/
/*  DISPLAYS THE VALUE OF KURTOSIS IN THE BOTTOM RIGHT-HAND      */
/*  CORNER OF THE SCREEN.                                        */
/*****************************************************************/
    gotoxy(70, 24);
    printf("%2d.%3d  ", c, rem); /* DISPLAY NUMBER AS TWO  */
                                 /* INTEGERS.              */
    if (rem < 10)
    {
        gotoxy(73, 24);  printf("00");  /* FILL ANY SPACES */
    }                                   /* WITH ZEROES.    */
    else if (rem <100)
    {
        gotoxy(73, 24);  printf("0");
    }
}
/*****************************************************************/
void timefreq (int *mode)
{
/*****************************************************************/
/*  CHANGES BETWEEN TIME, FREQUENCY AND CEPSTRAL MODES.          */
/*  ALSO RE-DRAWS THE SCREEN WITH THE APPROPRIATE AXES.          */
/*                                                               */
/*****************************************************************/
    unsigned freq_domain[2]  = {0xFF80, 0x0A00},
             time_domain[2]  = {0xE778, 0xFF80};
    float xaxis[9], endaxis;
    int i;

    ++ *mode;
    if (*mode > CEPS) *mode = TIME;
```

```
switch (*mode)
{
    case TIME:      /** DISPLAY THE SIGNAL'S TIME TRACE    **/
    {
        pokewf(time_domain, 0x01DA, 0x2, seg320);
        gotoxy(70, 1);
        printf("domain:");
        gotoxy(69, 2);
        printf("  TIME   ");
        offset = 100; scale = 5;
        gotoxy(1, 24);
        printf("_____________________________________");
        printf("_____________________________________");
        endaxis = 256.0/bandwidth;
        for (i=0; i<=8; i++)
        {
            xaxis[i] = i*endaxis/8;
            labelaxis (xaxis, i);
        }
        gotoxy (69,25); printf ("mS ");
        break;
    }
    case FREQ:    /**  DISPLAY THE SPECTRUM OF THE SIGNAL  **/
    {
        pokewf(freq_domain, 0x01DA, 0x2, seg320);
        gotoxy(70, 1);
        printf("domain:");
        gotoxy(69, 2);
        printf("FREQUENCY");
        offset = 190; scale = 5;
        gotoxy(1, 24);
        printf("_____________________________________");
        printf("_____________________________________");
        endaxis = 1.0 * bandwidth;
        for (i=0; i<=8; i++)
        {
            xaxis[i] = i*endaxis/8;
            labelaxis (xaxis, i);
        }
        gotoxy (69,25); printf ("kHz");
        break;
    }
    case CEPS:    /**  DISPLAY THE CEPSTRUM OF THE SIGNAL. **/
    {
        gotoxy(70, 1);
        printf("domain");
        gotoxy(69, 2);
        printf("CEPSTRUM ");

        gotoxy(1, 24);
        printf("_____________________________________");
        printf("_____________________________________");
        endaxis = 128.0/bandwidth;
```

```
            for (i=0; i<=8; i++)
            {
                xaxis[i] = i*endaxis/8;
                labelaxis (xaxis, i);
            }
            gotoxy (69,25); printf ("mS ");
            break;
        }
    }
}
/*****************************************************************/
void corrsig (int *output)
{
/*****************************************************************/
/*  CHANGES THE DISPLAY BETWEEN THE CORRELATED AND               */
/*  UNCORRELATED SIGNALS.                                        */
/*                                                               */
/*****************************************************************/
unsigned correlated[2]   = {0x2077, 0x597F},
         uncorrelated[2] = {0x2077, 0x597B};

    if (in_or_out == INPUTSIG) *output = 1- *output;
    in_or_out = OUTPUTSIG;
    switch (*output)
    {
        case UNCORR:       /* DISPLAY THE UNCORRELATED SIGNAL. */
        {
            pokewf(correlated, 0x0130, 0x2, seg320);
            gotoxy(70, 5);
            printf("signal:");
            gotoxy(68, 6);
            printf(" CORRELATED "); *output = CORR;
            break;
        }
        case CORR:           /* DISPLAY THE CORRELATED SIGNAL. */
        {
            pokewf(uncorrelated, 0x0130, 0x2, seg320);
            gotoxy(70, 5);
            printf("signal:");
            gotoxy(68, 6);
            printf("UNCORRELATED"); *output = UNCORR;
            break;
        }
    }
}
/*****************************************************************/
void primref (int *input)
{
/*****************************************************************/
/*  CHANGE THE DISPLAYED SIGNAL TO EITHER THE PRIMARY OR THE     */
/*  REFERENCE SIGNAL.                                            */
/*****************************************************************/
unsigned reference[2] = {0x2077, 0x597C},
         primary[2]   = {0x2077, 0x5967};

    if (in_or_out == OUTPUTSIG) *input = 1- *input;
    in_or_out = INPUTSIG;
```

```
    switch (*input)
    {
        case SIGNAL:      /* CHANGE TO DISPLAYING THE REFERENCE */
        {                 /* SIGNAL.                            */

            *input = NOISE;
            pokewf(reference, 0x0130, 0x2, seg320);
            gotoxy(70, 5);
            printf("signal");
            gotoxy(68, 6);
            printf("   NOISE    ");
            break;
        }
        case NOISE:       /* CHANGE TO DISPLAYING THE PRIMARY   */
        {                 /* SIGNAL.                            */

            *input = SIGNAL;
            pokewf(primary, 0x0130, 0x2, seg320);
            gotoxy(70, 5);
            printf("signal");
            gotoxy(68, 6);
            printf("   SIGNAL   ");
        }
    }
}
/*****************************************************************/
void recorddata (void)
{
/*****************************************************************/
/* PROMPT THE USER FOR A FILE NAME AND SET THE RECORDING FLAG  */
/*                                                               */
/*****************************************************************/
    int  key, xcurs, count, i;
    char *path, search[13];

    do{
        key = 0; xcurs = 46; count = 0;

        for (i=0; i<=8; i++)
            filename[i] = NULL;
        for (i=0; i<=12; i++)
            search[i] = NULL;

        gotoxy (2, 2);
        printf ("ENTER THE FILE NAME "
                "(NO MORE THAN 8 CHARS):          ");
        do{
            while (bioskey(1) == 0);    /* WAIT FOR A KEYPRESS */
            if (count == 0)             /* IF A WRONG FILE NAME HAS    */
            {                           /* ALREADY BEEN ENTERED...     */
                gotoxy (2,4);
                printf("                                        ");
                gotoxy (46, 2);  /* ERASE IT.                      */
            }
            key = getkey();
```

```
      /** IF THE BACKSPACE KEY HAS BEEN PRESSED... **/

            if (key == BSPACE && xcurs > 46)
            {
                --xcurs;
                --count;
                gotoxy(xcurs, 2);
                printf(" ");
                gotoxy(xcurs, 2);
                filename[count] = NULL;
            }

    /** IF AN ALLOWED FILENAME CHARACTER HAS BEEN PRESSED... **/

            if (key > 32 && key < 123 && xcurs <= 53)
            {
                gotoxy (xcurs, 2);
                printf ("%s", &key);
                filename[count] = key;
                ++xcurs;
                ++count;
            }
        } while (key != ENTER);        /* UNTIL ENTER IS KEYED  */

        strcpy (search, "A:\\");       /* SEARCH FOR THE FILE   */
        strcat (search, filename);     /* ON THE A-DRIVE. IF    */
        path = searchpath (search);    /* IT EXISTS THEN TELL   */
                                       /* THE USER TO TRY AGAIN */
        if (path != NULL)
        {
            gotoxy (2,4);
            printf("File already exists: try again.");
        }
    } while (path != NULL); /* UNTIL FILE NAME IS CORRECT. */

    gotoxy (2, 2);
    printf ("Record Kurtosis every 2 secs.");
    printf (" for 1 min.                  ");
    gotoxy (2, 4);
    printf ("                                        ");
    gotoxy (2, 5);
    printf ("Strike any key to start procedure...");

    while (bioskey(1) == 0);
    record = 0;                  /* SET THE RECORDING FLAG.       */
    prevclk = time (ptr);        /* KEEP A COUNT OF THE SECONDS.  */

    gotoxy (2, 2);
    printf ("                                              \n\n");
    printf ("\n                                          ");
    gotoxy (70,18); printf ("Points");
    gotoxy (69,19); printf ("recorded:");
}
/*************************************************************/
```

```
void writedata (void)
{
/*****************************************************************/
/*  WRITE THE KURTOSIS VALUES RECORDED TO THE SPECIFIED FILE.    */
/*                                                               */
/*****************************************************************/
    int i, width=0;
    FILE *stream;

    /** CREATE A FILE FOR WRITING. **/

    stream = fopen (filename, "w");
    getdate (&today);
    gettime (&now);

    fprintf (stream, "The date is: %d/%d/%d.\n",
             today.da_day, today.da_mon, today.da_year);
    fprintf (stream, "The time is: %02dh%02d.\n",
             now.ti_hour, now.ti_min);
    fprintf (stream, "The kurtosis values recorded are:\n\n");

    for (i=0; i<record; i++)
    {
        fprintf (stream, "%7.3f ", ktosis[i]);
        ++width;                          /* FORMATTED OUTPUT */
        if (width == 8)                   /* TO THE FILE.     */
        {
            width = 0;
            fprintf (stream, "\n");  /* START A NEW LINE.     */
        }
    }
    fclose (stream);

   /** REVERT TO NON-RECORDING MODE AND ERASE RECORDING TEXT. **/

    record = -1;
    gotoxy (70,18); printf ("      ");
    gotoxy (69,19); printf ("         ");
    gotoxy (72,21); printf ("  ");
}
/*****************************************************************/
```

Appendix E

The Assembler Utility Flow Diagram

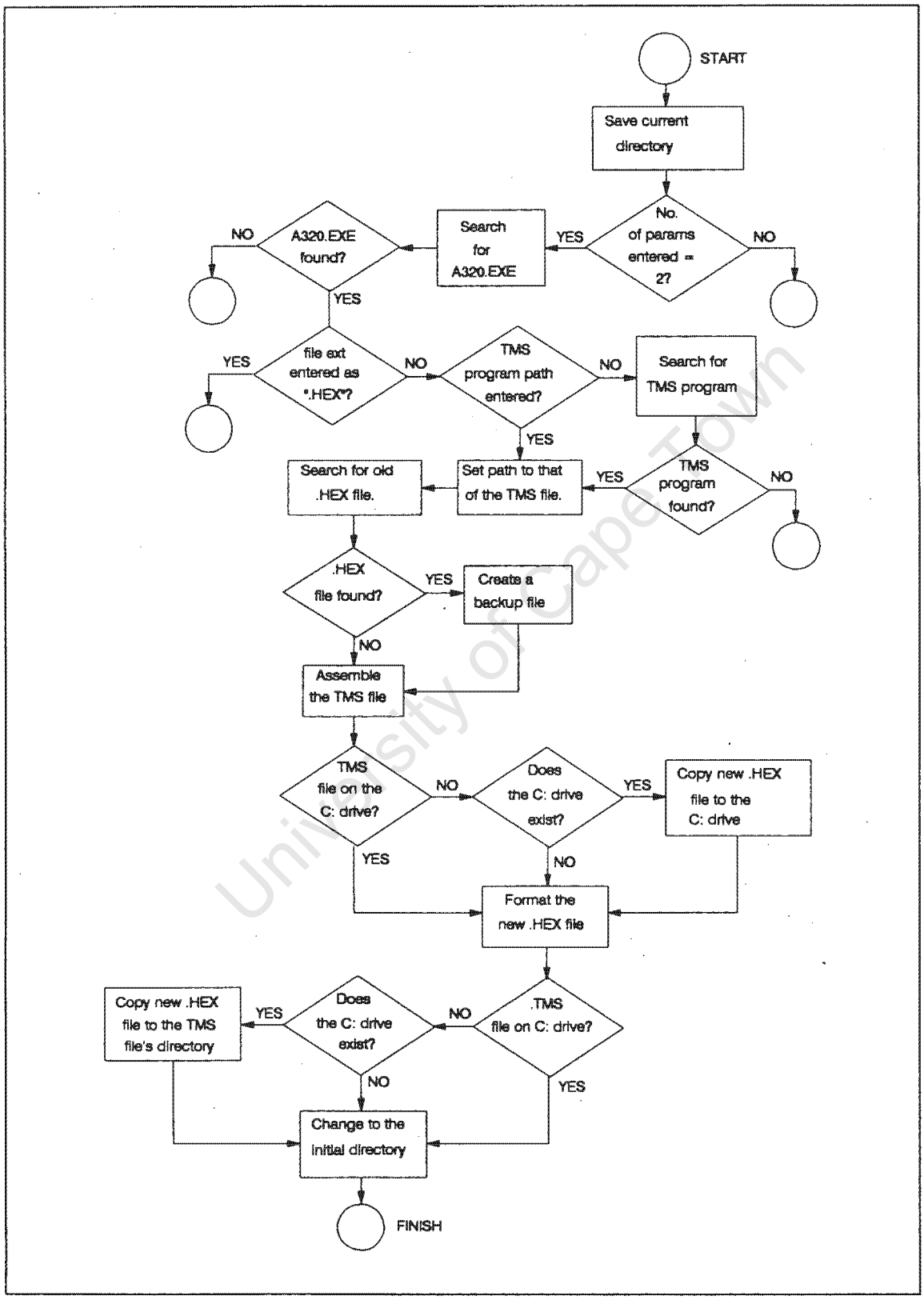


Figure E.1 Flow diagram of the Assembler Utility.

APPENDIX F

THE ANTI-ALIASING FILTER CHARACTERISTICS

The anti-aliasing filters were required to have as sharp a transition band as possible and also as great an attenuation in the stop-band as possible. For this reason it was decided to use elliptical filters, despite the ripple that they produce in both the pass and stop bands.

The filters used were copied out of the tables of elliptical filters that are found in [13]. The filters are seventh-order with a reflection coefficient of eight percent and a modular angle of 60 degrees. For an explanation of these parameters refer to chapter five of this document.

The filter has the shape shown in Figure F.1 and the values of the inductors and capacitors are given in Table F.1. The poles and zeroes of attenuation in the Laplace domain (the s-plane) are given in Figure F.2 [13]. Poles and zeroes of attenuation are exactly opposite to poles and zeroes of transmission: at an attenuation pole there is no signal transmitted.

In the following list of figures, Ω_s is the end of the transition band and the beginning of the stopband of the filter. Ω is the end of the passband and the beginning of the transition band.

Modular angle:	60°
Minimum attenuation in stopband:	40.71 dB
Normalized Ω_s (in radians):	1.1547
Unnormalized Ω_s (in Hz):	26 558
Unnormalized Ω (in Hz):	23 000

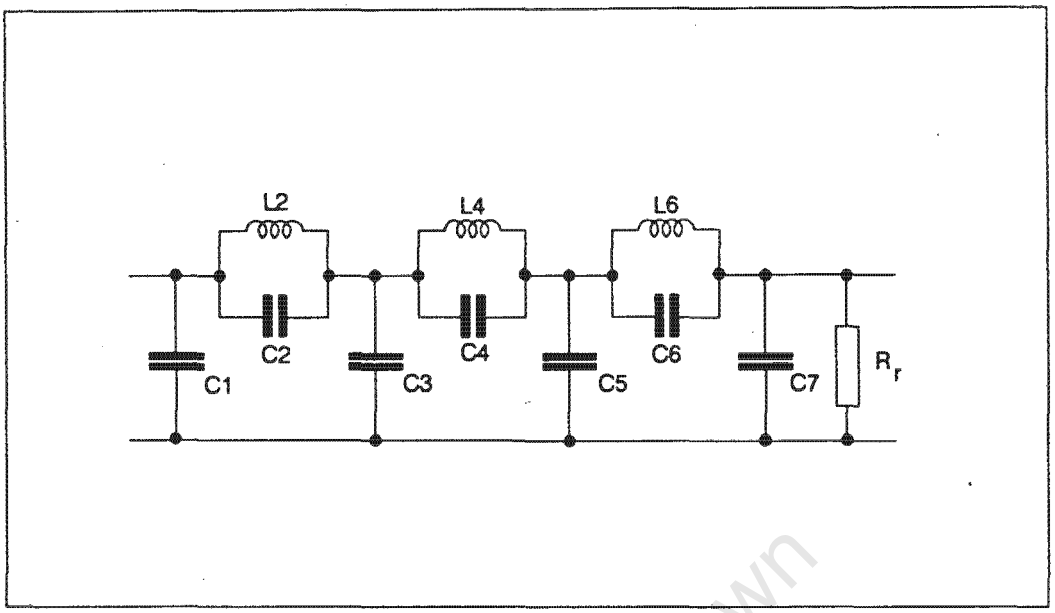


Figure F.1 Circuit Diagram of the Anti-Aliasing Filter

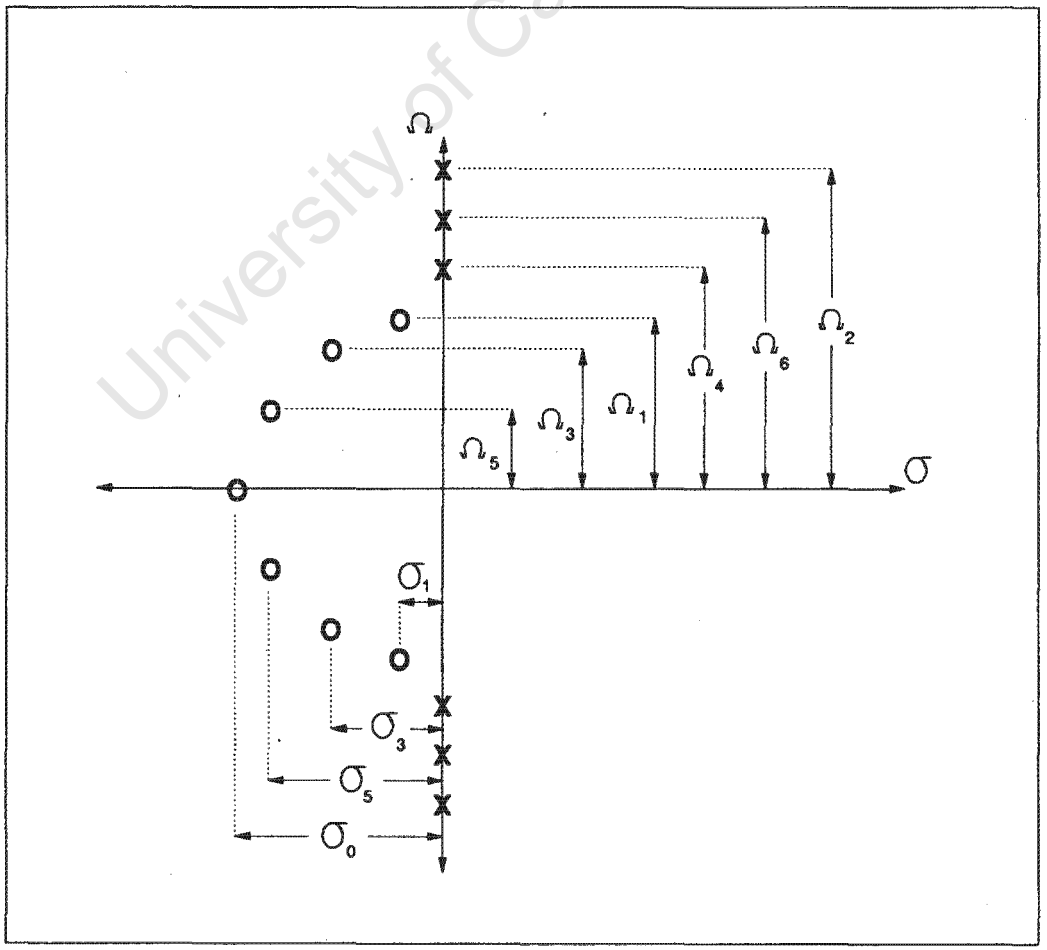


Figure F.2 Positions of the Poles and Zeros of the Filters

Frequency			Decay Curve		
	Normalized (rad.)	Unnorm. (Hz)		Normalized (EXP(σT))	Unnorm. (EXP(σT))
Ω_1	1.0395	23 910	σ_0	-0.71595	-16 466
Ω_2	2.0776	47 785	σ_1	-0.04184	- 962
Ω_3	0.9767	22 465	σ_3	-0.17733	- 4 079
Ω_4	1.1689	26 885	σ_5	-0.46632	-10 725
Ω_5	0.7189	16 535			
Ω_6	1.3235	30 440			

Table F.1. Coordinates of the poles and zeros of attenuation.

Capacitance			Inductance		
	Normalized (F)	Unnorm. (nF)		Normalized (H)	Unnorm. (mH)
C1	0.78583	36.25			
C2	0.19074	8.799	L2	1.21467	1.261
C3	1.23378	56.92			
C4	1.00006	46.14	L4	0.73188	0.7597
C5	1.05571	48.70			
C6	0.76284	35.19	L6	0.74833	0.7768
C7	0.43800	20.21			

Table F.2 The capacitances and inductances used in the filter.

To normalize the inductor and capacitor values use these conversions:

$$C_{norm} = \Omega C_{unnorm} R_r$$

$$L_{norm} = \Omega L_{unnorm} / R_r$$

where R_r is the resistance of the filter load, which is 150Ω.

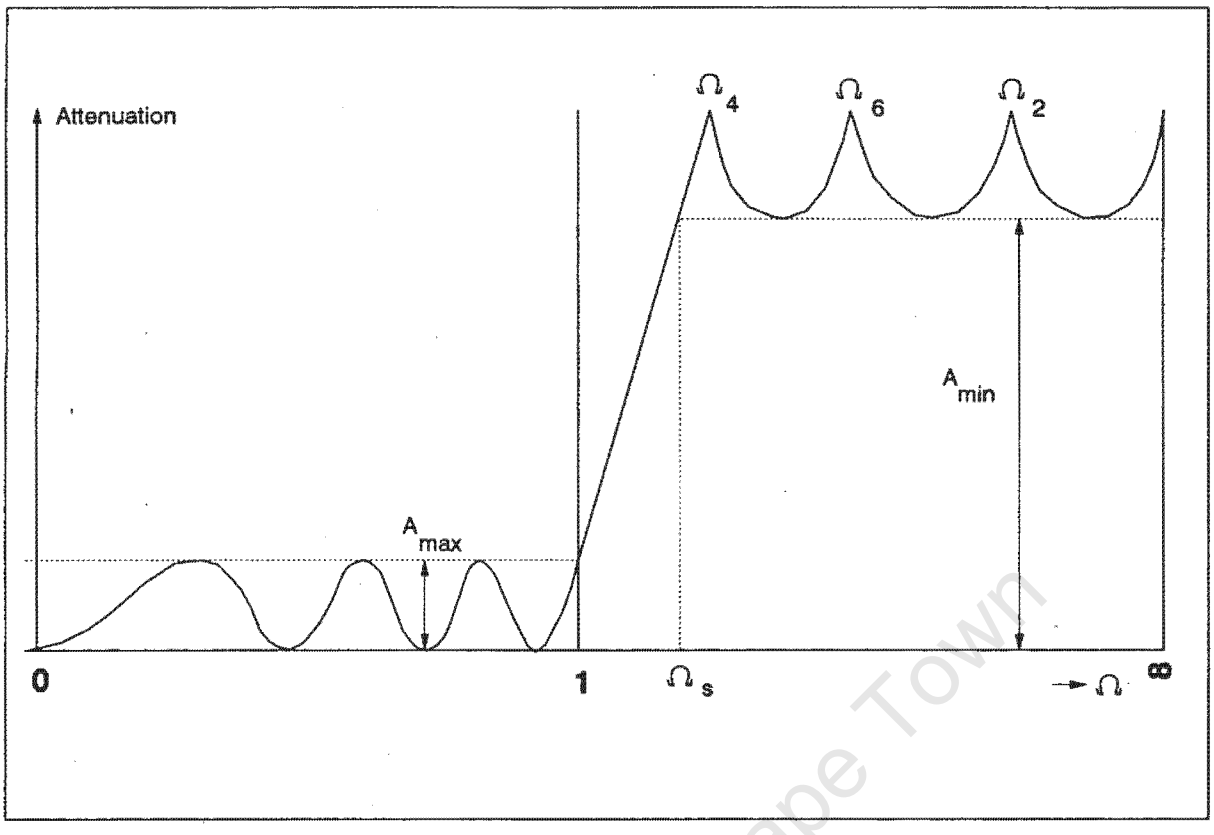


Figure F.3 Attenuation versus Frequency For The Anti-Aliasing Filters